An extension of embedded C for parallel programming

Master-Thesis von Bastian Gorholt

August 2014

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Software Engineering Group

An extension of embedded C for parallel programming

Vorgelegte Master-Thesis von Bastian Gorholt

1. Gutachten: Sebastian Erdweg
2. Gutachten:

Tag der Einreichung:

Abstract

Contents

1 Introduction

Traditionally, the performance of processors is increased by raising their clock rates. Higher frequencies in turn entail even bigger power consumption heightenings. Various techniques are applied by manufacturers to compensate this effect, mainly the reduction of voltages. However, with the advent of the so-called power wall [60], the industry eventually had to and is still facing the problem of not being able to achieve this goal any more. One way to escape this dilemma that is of growing importance is the exchange of higher frequencies for more processor cores on a chip [38]. Instead of executing the instructions of programs continually faster, the multi-core approach is to execute multiple instructions in parallel. While multiple simultaneously running programs can thus be ran in parallel, with rising computational complexity, it is also desirable to distribute the execution of single programs across multiple cores. Yet, programs need to be accordingly designed in order to make use of the new architectures. Parallel programming needs to be applied. The art of this programming technique has proven to be substantially more difficult, since for programs, which run in parallel, program correctness is harder to achieve. This risen difficulty, in turn, influences the productivity and the costs of produced software [36]. As the trend towards multiprocessing has also reached the embedded domain [24][6], this domain is facing the same problems. One tool to "boost the productivity" [2]f for this area is mbeddr. It offers new language-abstractions and integrated solutions for the development process. This is done by a compiler-based extension of the underlying programming language *C*: language abstractions are repeatedly reduced into more basic language concepts until plain C code is obtained. In spite of mbeddr's various abstractions, it still misses native language support for parallel programming. While it is possible for the user to use according existing libraries, it is a laborious task to prepare mbeddr in this way. Even more important, the programmer is faced with the usual problems of such a library-based approach: the danger of data-races and the drawback of unnecessary computational overhead due to suboptimal countermeasures. Hand-based optimization of this overhead, in turn, introduces the potential for compromising the correctness of programs and is a tedious and costly process. It is therefore desirable to have native support for parallel programming in mbeddr. This thesis thus aims at providing the user with language concepts that enable him[1] to write parallel code more easily while simultaneously mitigating the risks of incorrect execution. Additionally, this work will show how to reduce the design-based performance overhead, which the abstractions for data-race safety naturally entail. Due to the complexity of the underlying language C, the presented optimization techniques are prototypically implemented.

The text has the following structure. Chapter 2 will provide the reader with the necessary background to understand the reasoning of the following discussions. It gives short introductions to parallel programming, the character of the embedded domain and the language *C* that is extended. The 3rd chapter will then introduce the concepts of the new language abstraction *ParallelMbeddr*. To this end, the extension's design is presented, accompanied by the translation to the basic concepts that are provided by mbeddr.

[1] In this text, for simplicity reasons, 'he', 'him' and 'his' is used in a generic way and will always denote both sexes.

After this, chapter 4 will show first techniques for compiler-based optimizations of the generated code of ParallelMbeddr. This is done in a declarative algorithmic way. Furthermore the text will depict the difficulties which arose in the prototypical implementation of the algorithms, and propose solution approaches therefor. The evaluation in chapter 5 will show the benefits and shortcomings of the design and implementation. Both the user-written and generated code for general parallelization examples will be investigated. Furthermore, the results of an optimization measurement will be shown and will reveal the performance potential for the compiler-based enhancement of C. The same chapter then gives a short comparison to relevant related work in the parallel-programming field. Finally, chapter 6 will summarize the achievements and results of this work and briefly list promising fields of future research for ParallelMbeddr.

2 Background

This chapter gives a concise introduction into the basic topics of this work. First, the main aspects of parallel programming are explained, in order to give the reader the essential knowledge that is mandatory to understand the problem tackled by this thesis. After that the embedded domain and the tools that are used to create software for it (*mbeddr* and *C*) are presented.

2.1 Parallel Programming

Parallel programming is a technique to speed up the classical sequential execution of programs which are suited for parallelization in the limits of *Amdahl's law* [10][43]. The law gives an upper bound on the possible speed up of an algorithm based on the amount of parallelizeable code.[2] The pitfalls of the often mandatory communication in parallel programming arise from race conditions and countermeasures thereof.

Processes and Threads

Every program in execution which may be delayed or interrupted is represented by a process. A process typically has its own protected data (via virtual memory) and execution state and consists of one or more threads. A thread, also known as lightweight process, shares some of the memory with its process but has its own execution state and thread-local storage as needed [28, p. 20]. In the course of programming language and operating system development, several variants of threads have been devised, among others green threads, fibers and coroutines which mainly differ in how they are managed.

Parallelism and Concurrency

If multiple threads are "in the middle of executing code at the same time" [41, p. 124(?)] they are processed concurrently. They can be executed at the same time on different processors or interleaved on a single processor, which means that they are executed in an alternating way. The former is also called parallelism. Parallelism, generalized to "the quality of occurring at the same time" [21, p. 91], can manifest in at least four different ways: From an application programmer's view there exist bit-level parallelism and instruction-level parallelism at a very low level. Bit-level parallelism is concerned with increasing the word size of processors in order to reduce the amount of cycles that are needed to perform an instruction [19, p. 15]. Instruction-level parallelism, also called pipelining, is the simultaneous utilization of multiple stages of the processor's execution pipeline. Both bit-level parallelism and instruction-level

[2] This work, however, is not about finding appropriate patterns for this purpose but about giving a generic approach. Thus, *Amdahl's law* will not be further considered.

parallelism primarily reside on the hardware level and the operating system level and thus are irrelevant to this work. Data parallelism is present if the same calculation is performed on multiple sets of data. It can be regarded as a specialization of task parallelism, which denotes the simultaneous execution of different calculations "on either the same or different data" [19, p. 125]. This work focuses on the latter form of parallelism, since it is the more general approach to software-level parallelism.

Data Races

When a process consists of multiple concurrently running threads which have access to the shared data of their process, a class of errors that is unique to parallel (and distributed) programming arises. These so-called synchronization errors occur due to data races and are a result of the general non-atomicity of computations and memory references. Data races – also known as race conditions – are defined as at least "two unsynchronized memory references by two processes on one memory location, of which at least one reference is a write access" [22, p. 327].[3] Such data races can result in inconsistent program states and non-deterministic program behavior since the order in which the concerned memory is accessed might change. In order to deal with this issue, three main paradigms have been conceived in parallel programming: shared memory, message passing and transactional memory.

Communication Model: Shared Memory

The memory model that implicitly underlay the former treatment of processes, which share some data with their threads, is formally known as *shared memory* . In this model, communication between entities is realized by shared-memory regions, which are written to and read from [31, p. 138].[4] Data races can be avoided by employing the low-level synchronization primitive called *mutex*.[5] A mutex can only be locked by exactly one thread at a given time. Any other thread that tries to lock the same mutex is blocked until the locking thread releases the mutex [42]. Thus, code regions can be protected by having threads synchronize over mutexes that protect these regions. One of the disadvantages of mutexes is that they are not tightly coupled to the data or computation that they protect. It is the programmer's duty to take care of the sane utilization of a certain mutex. Therefore, various higher-level synchronization measures like monitors in Java [27, p. 42] and synchronized classes in the programming language *D* [8] were developed. These measures are usually built on top of mutexes [37, p. 25]. Another disadvantage of mutexes is their vulnerability for deadlocks. This means that multiple threads are in a state where "each is waiting for release of a resource which is currently held by some other" [50, p. 119] thread. As a result, no thread will ever finish executing [23, p. 2-3].

[3] As the definition implies, data races are not limited to threads and the shared process data alone, e.g. file-based race conditions can even occur between two different processes [62]. Since this work is about parallelization of single processes, other kinds of race conditions are not considered further.

[4] As for race conditions, the model is not limited to intra-process communication via threads or similar approaches to parallelization. Communication between two processes can also be realized via shared memory, however, this topic will not be addressed further in this thesis.

[5] Semaphores, as a second synchronization primitive, are closely related to mutexes. Since they are irrelevant for this paper, they are not further investigated.

Communication Model: Message Passing

Whereas communication in the shared memory paradigm happens rather implicitly, it is done explicitly in the *message passing* paradigm. Message passing originates from Hoare's paper on Communicating Sequential Processes (CSP) [35]. In CSP, messages are sent from one entity to another. "The sender waits until the receiver has accepted the message (*synchronous* message passing)" [53, p. 138] before it continues its execution. Message passing with asynchronous message sends were deployed by the actor model [34] and pi calculus [46]. Although message passing avoids shared data and realizes communication generally via copies of data,[6] it still suffers from potential race conditions [48] when the atomicity of operations is not properly regarded by the implementation.

Communication Model: Transactional Memory

Transactional memory provides a non-blocking[7] memory model, which enables communication via "lightweight, *in-memory* transactions" [30, p. 3]. Transactions are code blocks that, from a programmer's perspective, are executed atomically. The illusion of atomicity is realized by the underlying transaction system, which may execute transactions in parallel and has to take care of conflicting reads and writes in transactions [33].[8] Transactional memory can either be realized in hardware or in software. While the former promises a better performance, it demands specific hardware. Software transactional memory on the other hand seems to suffer from comparatively "poor performance" [13, p. 13].

Coarse- and Fine-grained Synchronization

In order to keep data and computations synchronized, the simplest measure to avoid data races is to use the available instruments like locks or transactions as broadly as possible. E.g. transactions could be widened to hold every operation a thread has to execute. As every synchronization is basically a serialization of otherwise parallel executed code, such coarse-grained synchronization would eliminate the benefits of parallel execution. On the other hand, fine-grained synchronization can introduce race conditions if the programmer misses some locking policy. In addition, the acquisition of every lock takes time, which can become an issue with increasing locking counts. Therefore, a trade-off between locking-overhead and scalability problems has to be found [30, pp. 1-2].

2.2 Embedded Programming with Mbeddr

In the course of this thesis parallelization is introduced into the programming language *C* which is, due to its performance and computational predictability, a natural choice for the embedded progamming

[6] Actually shared data is often used in implementations of message passing models in order to enhance the performance. Furthermore it exists on the language level, for example as monitors, which were developed by Hoare to reduce deadlocks in CSP. The main notion of the message passing concept nevertheless does not use shared data.

[7] TODO: explain

[8] To this end, the corresponding transactions may need to be re-executed as a whole.

domain. *mbeddr* thrives to make the development of software in this domain both easier and safer by giving advanced tool support.

Embedded Programming

“An embedded system is a computerized system that is purpose-built for its application.” [69, p. 1] Due to its narrow scope and monetary constraints induced by the application domain, the hardware of such systems is often constrained to the point that it simply accomplishes the job [69]. As a result, the memory consumption of the resulting program is one main issue to be considered in embedded programming. Additionally, for real-time systems which constitute a subclass of embedded systems not only the correctness of computations but also the consumed time determine their quality and usefulness [18, pp. 1-2]. Therefore, the predictability of the program’s execution time becomes an issue for real-time systems. To this end, predetermined dead-lines of computations are used. In accordance to the consequences of timely missed dead-lines, multiple levels of real-time systems have been conceived. These, however, will not be further discussed.[9]

MPS and Mbeddr

TODO: Visualisieren

“JetBrains MPS[10] is an open source [...] language workbench developed over the last ten years by JetBrains. ” [67] As such it provides a projectional editor which lets the user work directly on the abstract syntax tree (AST) of the program [25]. It supports the development and composition of potentially syntactically ambiguous modular language extensions in combination with the development of integrated development environments (IDEs) or extensions thereof. Mbeddr is an extension of MPS tailored for the embedded software development in C. Every program written in the mbeddr IDE is translated to C99 source code which are then further processed by the gcc tool chain.[11] Every new language construct that is introduced as an extension of the existing language of mbeddr (and any existing construct) is represented by a *concept*. The complete definition of a concept comprises the definition of certain *language aspects*: the structure of a concept is defined in a *structure* aspect which contains the definition of the concept’s abstract syntax in the abstract syntax tree (AST) of the program. The visual representation of the concept, its concrete syntax, is defined in the *editor* aspect. The *type* aspect of a concept may contain a type inference rule for it and further *non-typesystem rules*, which restrict the way it may be used. In a formal meta-language, the latter are usually part of the inference rules but are kept separately in MPS to diminish the complexity of inference rules and facilitate an easy extension of them. The *generator*

[9] The reason for this decision is that this work tries to cover a rather broad domain, instead of focusing on any particular embedded sub-domain. Furthermore, mbeddr does not have first-class support for the quantification of related parameters like the worst-case execution time (WCET) [18, p. 8], yet. In consequence, predictability is a lesser concern of this thesis and will be reflected primarily in the careful consideration of the CPU consumption and processing time of the implementation.

[10] http://jetbrains.com/mps, accessed: 2014-07-18

[11] http://gcc.gnu.org/, accessed: 2014-07-18

“defines the denotational semantics for the concepts in the language” [61]. Thus, this aspect describes the translation of concepts into concepts of the base language. The implementation of C in mbeddr does not only provide extensions to the core of C but also offers a few differences to the basis of the C99 standard. They will be introduced as needed.

C

The semantics of C differ from modern object-oriented languages like Java in various ways. In order to clarify some of the choices that were made for the design and implementation of ParallelMbeddr, in the following, the most relevant differences shall be outlined. Like Java, C leverages pass-by-value semantics for function parameters. In contrast to Java though, which copies the references themselves, it copies the referenced values into the memory that is allocated to function calls. Thus, a change to a field of a struct instance that was copied in such a way does not affect the original struct instance.[12] On the other hand, arrays are treated like pointers, which becomes evident when they are passed to functions. Hence, a change of an entry of an array argument actually changes the array that is referred to by a variable on the caller site. For an array to be copied entirely into a function (not just by its address), it can be declared as a struct field, which, due to the copy semantics for structs, ensures that like any other field of the struct instance the array’s value is copied into the newly created struct instance. The copy semantics are not restricted to function arguments but also extend to function return values and variable assignments. The ‘pass-by-pointer-value’ semantics for arrays in C implies that arrays cannot be returned from functions as is done in Java. Instead corresponding pointers are returned. This means that it is not safe to return an array, respectively a pointer thereof, from a function if the array resides in the area of the stack that was allocated for this function.[13] A peculiarity of C is that global variables may only be initialized with constant expressions, which makes it impossible to initialize a global variable with an arbitrary function call of a proper type [16, p. 48]. In C, the type of a pointer to a value of type **`t`** is written **`t*`**.

TODO: Maybe add section for explicit vs. implicit parallel programming, look at Programming Distributed Systems by H. E. Bal, pp. 113-114

TODO: Implementation: coarse- vs. fine-grained synchronization, problems => implicit synchronization not exhaustive => optimization for safe lock avoidance helpful

[12] The only way to avoid this behavior is to copy the memory addresses of values as pointers into functions.

[13] Otherwise the pointed-to memory of the returned array pointer would become deallocated after the return of the called function. This again would cause the return of a dangling pointer into the receiver of the returned value, i.e. a pointer that does not point to a valid memory address.

3 Design and Translation

In this chapter, the extension of mbeddr for parallel programming, called ParallelMbeddr, is introduced. To this end, the new language features for C are explained, each in terms of their design and the translation to plain mbeddr C code.[14] In order to illustrate the presented features, a running example is incrementally built. Further examples are depicted whenever the running example does not provide the right structure to clarify a feature. At the end of this chapter, the measures implemented to make the extension sufficiently thread-safe[15] are explained.

Notation

In the following sections, the concepts of the language of mbeddr are iteratively extended with new elements. For this purpose, based on the notation presented by Pierce [51], the syntax of new concepts is described by notations like the following:

```
e ::= ...| e'
```

This exemplary line extends the set of expressions of mbeddr by the expression **e'** which may contain arbitrary meta-variables like **e**. In order to fit into the type system of mbeddr, these concepts are equipped with *type inference* rules, each of which, given a list of premises, derives the type of some concept in the conclusion:

$$\text{NewConcept}\ \frac{premise_1, ..., premise_n}{e' \vdash t'}$$

The premises of the concepts are kept minimal but not exhaustive, which means that complex premises are given as informal explanations and are mostly explained in section 3.4. This way, the type inference rules are kept comprehensible and the explanations for the safety measures of ParallelMbeddr are kept closely together. Additionally, this separation resembles MPS' separation of type inference rules and non-typesystem rules.

The translation of a new concept in ParallelMbeddr to a base concept of mbeddr will be shown informally by listing the resulting code that is generated by the IDE after it was given some input code. To this end, if not expressed in the text, the symbol $\Longrightarrow$ will denote the translation of code c_1 to some other code c_2 in $c_1 \Longrightarrow c_2$. The translation of basic code will often entail the generation of additional code somewhere else in the program as a side effect, e.g. the translation of an expression to a function call may force the generator of mbeddr to generate the declaration of the called function first. These side effects will also be demonstrated by listings of the generated code.

[14] For the sake of legibility, the syntax of the generated mbeddr code is depicted in a simplified manner where deemed necessary.

[15] Data-races should be prevented, as well as it possible. However, the user always has the opportunity to introduce data races if his requested synchronizations for the modifications of data, which is used by multiple units of execution, do not take account of the implicit data dependencies of variables.

3.1 Tasks

The basic parallelization element is a *task*. It denotes a parallel unit of execution and, as the name suggests, aims at task parallelism. As the implementation of the underlying parallelization technique might change in the future it is reasonable to abstract the terminology from it. The most basic task which always exists executes the code of the entry function of the program. A task can also be regarded as a closure of the expression that shall be run in parallel. The reader should distinguish this 'execution template' from the actual running instance of a task. The latter will further on be addressed as a *running task*.

3.1.1 Design

The syntax e of expressions in mbeddr is extended by
$e ::= ...|\ |e|$
When executed, a task term yields a handle to a parallel unit of execution. This way, the initialization of the task and the actual execution are decoupled and can happen independently. When a task is run the embraced expression is executed and its value is returned. If the type of the expression is *void* then no value will be returned. The type of a task reflects its return value:
$t ::= ...|\ Task{<}t{>}$
Due to implementation reasons (see 3.2.2 for details), the embraced return type of a task must be either *void* or a pointer to the type of the embraced expression:

$$\text{VoidTask}\frac{e \vdash void}{|e| \vdash Task{<}void{>}} \qquad \text{NonVoidTask}\frac{e \vdash t \qquad t \neq void}{|e| \vdash Task{<}void{*}{>}}$$

When a task is not used anymore to produce running instances of itself, it should be cleared in order to free the memory that it implicitly occupies on the heap:
$e ::= ...|\ e.clear$

$$\text{VoidTask}\ \frac{|e| \vdash Task{<}void{>}}{e.clear \vdash void}$$

If a task is copied by the pass-by-value semantics of C, the copied task will share the heap-managed data, i.e. the reference environment of its free variables, with the original task. Therefore, a task needs to be cleared only once in order to avoid memory leaks. It has to be kept in mind that a running instance of a task will not be affected by the clearance of its task template. The clearance of a task is only necessary if the task is stored somewhere. Thus, a task that is directly run via **`|e|.run`** need not be cleared, which makes such expressions memory-safe.

3.1.2 Translation

The POSIX Threads standard and library (pthreads) was chosen as a means to realize concurrency in the translation. It supports all necessary parallelization features and provides a more direct control of the generated code when compared to frameworks like OpenMP.[16] Every task in ParallelMbeddr is represented by a thread as provided by the POSIX threads standard, a so-called *pthread*.[17] As the thread initialization function of pthreads takes a function pointer of type **void* -> void***, the computation of the translated task is represented by an according function:

```
void* parFun_X(void* voidArgs) {...}
```

The **X** in the name symbolizes that for every task a unique adaptee of this function with the prefix **parFun_** and some unique suffix chosen by the framework is generated.[18] As the function signature indicates, a pthread and therefore a task which is to be run can be parameterized with values and can return a value which will be explained in the following paragraphs.

If a task contains any references to local variables or function arguments, they need to be bound to capture the variable states at the time of the task initiliazation. Such states are represented by an 'argument' struct:

```
struct Args_X {
  t_1 v_1;
  ...
  t_n v_n;
}
```

where in the task expression every **v_i** represents an equally named reference to a variable of type **t_i**. The generated function **parFun_X** is then given an instance of **Args_X**, which it uses to bind the references of the task expression to. The full function definition of a task **e** of type **Task<t*>** is, thus:

```
void* parFun_X(void* voidArgs) {
  t* result = malloc(sizeof(t));
  Args_X* args = (Args_X*) voidArgs;
  *result = e';
  return result;
}
```

where **e'** is the expression obtained when every local variable reference and function argument reference **r** in **e** is substituted by a reference to an equally named and typed field in **args**:

$r \,/\, args{-}{>}r$

If the embraced expression of a task does not contain any reference of this kind (e.g. only references to global variables) the **args** definition line is omitted as is clearly the – otherwise empty – declaration of **struct Args_X**. In this case **e'** equals **e** except for other reductions of **e** that might occur in the translation process of mbeddr.

The generated function of a task of type **Task<void>** omits the result-related statements:

```
void* parFun_X(void* voidArgs) {
```

[16] http://openmp.org/

[17] In the following sections *pthreads* will denote both the library and multiple threads as they are provided by the library. The context should always clarify which one is currently meant.

[18] In the following explanations, **X** will always denote some arbitrary suffix. It has to be kept in mind, though, that these suffixes do not necessarily coincide with each other for different kinds of components.

```
  t* result = malloc(sizeof(t));
  Args_X* args = (Args_X*) voidArgs;
  e';
}
```

Again, any argument-related code is generated as needed.

The aforementioned handle that a task yields is represented by an instance of a corresponding struct which captures both the initialization state and the computation of the embraced expression.[19] The *void* pointer of the arguments **voidArgs** does not keep the arguments' type information and with it their byte size. Therefore, an additional field **argsSize** is needed in order to be able to create copies of the arguments later on (see 3.2.2 for details).

```
exported struct Task {
  void* args;
  (void*) => (void*) fun;
  size_t argsSize;
}
```

As opposed to the unique definitions of other elements that need to be defined for every occurrence of a task (the ones with the **X** suffixes), **struct Task** is generic and is reused for every task. Generic declarations are kept in fixed, separately generated modules and are imported into the user-defined modules. With these components in mind, the actual translation of a task expression **|.|** that contains references **v_1** to **v_n**, which need to be bound, becomes an mbeddr block expression:[20]

```
{
  Args_X* args_X = malloc(sizeof(Args_X));
  args_X->v_1 = v_1;
  ...
  args_X->v_n = v_n;
  yield (Task){ args_X, parFun_X,
                sizeof(Args_X) };
}
```

$\Longrightarrow$

```
taskInit(v_1, ..., v_n)
```

with function declaration

```
inline Args_X taskInit(t_1 v_1, ..., t_n v_n) {
  Args_X* args_X = malloc(sizeof(Args_X));
  args_X - >v_1 = v_1;
  ...
  args_X->v_n = v_n;
  return (Task){ args_X, parFun_X,
                 sizeof(Args_X) };
}
```

The expression of the **yield** statement is a compound literal, which on evaluation creates an instance of the aforementioned **struct Task**. The block expression is then further reduced by mbeddr to a call of a newly generated inline function.[21]

Without any references to bind, a task is reduced to the compound literal:

[19] **(void*) => (void*) fun** is mbeddr syntax for the not easily edible function pointer **void *(*fun) (void *)** in standard C99.

[20] A block expression contains a list of statements of which the mandatory yield statement returns the result value.

[21] Whereas in C for every struct type **T** a typedef has to be manually defined by the programmer, this definition is done implicitly in mbeddr, in order to reference this type directly with **T** instead of **struct T**.

```
(Task_X){ null, parFun_X, 0 }
```

The reduction of a task is accomplished differently if the task is immediately run via **`|e|.run`**. Section 3.2.2 will show how this is done.

By the above definition of **`parFun_X`**, it becomes clear that the arguments of a task – its environment – are stored on the heap before execution. This approach was chosen mainly in order to simplify the generation of the resulting code. On the other hand, both the result and the arguments of a task have to be deleted by the programmer by hand. Since, for reasons which are explained later, the result is returned via a pointer onto the heap as well, it becomes apparent that the return type of a task must either be a void type or a pointer type, as was mentioned in the design section. Concerning task arguments, one advantage of this implementation is that a task may be passed by value, e.g. when using a builder function to create tasks, without the possible need to copy multiple arguments. Instead just the pointer to the heap-managed data is copied. As will be shown in the (FUTURE WORK) chapter, a stack-based implementation of task (I/O) is conceivable.

In the translated code, the clearance **`e.clear`** of a task becomes a call of the **`free`** function of C, parameterized with the arguments of the translated task **`e'`**:

```
free(e'.args)
```

3.1.3 Example Code

The running example concerns itself with the calculation of π, based on the definition given in the concurrent-pi example for the programming language *Go*.[22] π is approximated by the summation of a certain number n of terms where n determines the deviation of the result from the actual value of π: $\pi_{\text{approx}} = \sum_{i=0}^{n} 4 * \frac{-1^i}{2i+1}$.

In the first scenario, the amount of work is distributed among a certain number of tasks, each of which calculates the contribution of summands for a range of indices i. The calculation of such a partial sum for a range $[\text{start}, \text{end}[$ of indices is done by the functions:

```
long double calcPiRange(uint32 start, uint32 end) {
  long double partialSum = 0;
  for (uint32 i = start; i < end; ++i) {
    partialSum += calcPiItem(i);
  }
  return partialSum;
}

long double calcPiItem(uint32 index) {
  return 4.0 * (pow (-1.0, index) / (2.0 * index + 1.0));
}
```

The work can be distributed among e.g. 4 tasks, where each task calculates a partial sum for an equally long range, which is given by:

[22] https://github.com/foamdino/learning-go/blob/master/concurrent-pi/concurrent-pi.go

```
#constant BLOCKSIZE = 300000000;
#constant BLOCKCOUNT = 4;
#constant THRESHOLD = BLOCKSIZE * BLOCKCOUNT;
```

These values are then used to initialize an array of tasks:

```
int32 main(int32 argc, string[] argv) {
...
  Task<long double*>[BLOCKCOUNT] calculators;
  for (i ++ in [0..BLOCKCOUNT[) {
    uint32 start = i * BLOCKSIZE;
    uint32 end = start + BLOCKSIZE;
    calculators[i] = |calcPiBlock(start, end)|;
  }
...
}
```

The final reduction of the calculated values will be shown after the presentation of futures in section 3.2.3. The code is translated[23] to the building blocks that were introduced in 3.1.2:

```

int32 main(int32 argc, string[] argv) {
...
  Task[BLOCKCOUNT] calculators;
  for (int8 __i = 0; __i < BLOCKCOUNT; __i++) {
    uint32 start = __i * BLOCKSIZE;
    uint32 end = start + BLOCKSIZE;
    calculators[__i] = taskInit_0(start, end);
  }
...
}

struct Args_0 {
  uint32 start;
  uint32 end;
};

inline Task taskInit_0(uint32 start, uint32 end) {
  Args_0* args_0 = malloc(sizeof(Args_0));
  Args_0->start = start;
  Args_0->end = end;
  return (Task){ args_0 , :parFun_0 , sizeof (Args_0)};
}

```

[23] For legibility reasons, the code shown in the following listing and any other mbeddr code presented is a simplified version of the intermediate code that mbeddr actually generates. The performed changes are restricted to renaming and negligible syntax changes.

```
void* parFun_0(void* voidArgs) {
  long double* result = malloc(sizeof(long double));
  Args_0* args = ((Args_0*) voidArgs);
  *result = calcPiBlock((args)->start, (args)->end);
  free(voidArgs);
  return result;
}
```

The type of **calculators** is translated into an array type of the generic **Task** struct type such that the type specification **long double***, which is not needed in the translated code, gets lost in this process. The task expression **|calcPiBlock(start, end)|** is translated into a function call of the generated inline function **taskInit_0**. This function stores the values of the referenced local variables **start** and **end** in a structure which will later be used as the input for the parallel executed function **parFun_0**. This function, the wrapped arguments and their size are stored in a generic **Task** structure instance which is the handle that will later be used to initiate the task. **parFun_0** takes its arguments generically (as is required by the POSIX threads standard) and also returns its result generically via the heap. As the arguments reside on the heap and are uniquely allocated, for this function they must be freed before **parFun_0** returns. The calculation of the result is straightforwardly given by the execution of the expression of the original task sub-expression **calcPiBlock(start, end)** except that the two variable references are substituted by references to the according fields in the argument struct instance **voidArgs** which is cast to the appropriate type **Args_0**.

3.2 Futures

Whenever a task **t** is run, a *future* is generated. Futures in ParallelMbeddr are based on Halstead's definition of a future [32]. A future is a handle to a running task that can be used to retrieve the result of this task from within some other task **u**. As soon as this happens, the formerly in parallel running task **u** joins **t**, which means that it waits for **t** to finish execution in order to get its result value. The asynchronous execution is, thus, synchronized.

3.2.1 Design

The syntax e of expressions in mbeddr is extended by:[24]

e ::= ...| *e.run* | *e.result* | *e.join*

Like with tasks, the type of a future is parameterized by its return type:

t ::= ...| *Future<t>*

e.run denotes the launch of task **e** whereas **e.result** joins a running task that is represented by a future handle **e**, i.e. halts the execution of the calling task until **e**'s execution is finished, and returns its result. The last expression **e.join** can be used to join tasks that return nothing. These properties are reflected in the typing rules:

[24] If the expression e has a pointer type the dots (.) are replaced by arrows (->).

$$\text{Future}\frac{e \vdash Task{<}t{>}}{e.run \vdash Future{<}t{>}} \qquad \text{FutureResult}\frac{e \vdash Task{<}t^*{>}}{e.result \vdash t^*} \qquad \text{FutureJoin}\frac{e \vdash Task{<}void{>}}{e.join \vdash void}$$

As was already depicted in the previous section, **result** returns a pointer to a heap-managed value. Hence the programmer has to free the value eventually.

3.2.2 Translation

A future type **Future<t*>** is translated to a generic **struct** that contains a handle of the thread, a storage for the result value – which is dropped for futures of type **Future<void>** – and a flag that indicates whether the thread is already finished:

```
exported struct Future {
  pthread_t pth;
  boolean finished;
  void* result;
};
```

For every task and future expression shown above, a generic function reflects the semantics in the translation. The **run** of a task involves taking a task, creating a pthread with the task's function pointer, and arguments and generating a future[25] with the initialized thread handle:[26]

```
Future runTaskAndGetFuture(Task task) {
  pthread_t pth;
  if ( task.argsSize == 0 ) {
      pthread_create(&pth, 0, task.fun, 0);
  } else {
    void* args = malloc(task.argsSize);
    memcpy(args, task.args, task.argsSize);
    pthread_create(&pth, 0, task.fun, args);
  }
  return ( Future ){ .pth = pth };
}
```

The code shows that the arguments to be provided for the thread are copied onto a new location in the heap, although they already reside in the heap as was shown in section 3.1.2. It is necessary to do so in order to avoid dangling pointers. These could arise when a task is cleared so that its arguments get deleted while one or more running instances (pthreads) of this task are not finished yet. Furthermore, generally every thread needs its own copy of the argument data in case it modifies it. A function

[25] In the following a future will always denote either the according concept or an instance thereof. A **Future**, on the other hand, will either denote the struct that is used for futures, an instance thereof or the according struct type – the respective meaning will be explicitly clarified if necessary.

[26] Obviously the thread handle is copied into the **Future**. This is safe as can be seen when looking at the POSIX function **pthread_t pthread_self(void)**, which also returns a copy of a thread handle. This useful property is worth mentioning since it does not hold for all POSIX related data structures as is explained in footnote 39.

corresponding to the previous function, called **runTaskAndGetVoidFuture**, is generated for futures that return nothing.

The signature of **pthread_create** indicates how the result of a threaded function can be received. As was already suggested **pthread_create** expects a function pointer of type **void* -> void***, which is the reason why the function generated for a task is equally typed. The result is thus a generic **void** pointer. This implies that the threaded function could generally return the address of a stack-managed value, i.e. a local variable. Since the existence of the value after thread termination cannot be guaranteed in such a case, a dangling pointer [55] could emerge, which resembles the aforementioned problem for thread arguments. The only safe alternative that fits the task-future structure well is to allocate memory from the heap and return the address of this memory (see section 3.1.2). The translation of the result retrieval is a call of the following function:

```
void* getFutureResult(Future* future) {
  if (!future->finished) {
    pthread_join (future->pth, &(future->result));
    future->finished = true;
  }
  return future->result;
}
```

First the future is used to join the according thread which blocks the execution until the thread is finished. Additionally, the result is copied into the designated slot of the future struct instance. Finally, the result is returned. In POSIX, a thread can only be joined once; every subsequent call causes a runtime error. In order to allow the user to request the result multiple times nevertheless the **finished** flag is used to determine whether a join should happen. The same basic structure can be found in the translation of the **join** function for a future of type **Future<void>**. The main difference is the missing result-related code:

```
void joinVoidFuture(VoidFuture* future) {
  if (!future->finished) {
    pthread_join (future->pth, null);
    future->finished = true;
  }
}
```

Both aforementioned generated functions take their future parameters by address. This is necessary to make the setting of the future data work. If struct instances of futures would be passed to these functions as is, then due to C's pass-by-value semantics only copies of the provided future arguments would be filled with data. Thus, the result of a task would never arrive in the original future struct instance. Furthermore, subsequent calls to these functions would always work with false **finished** flags and ultimately trigger runtime errors. The necessity for future pointers in turn does not assort well with chained future expressions like:

```
Task<int32*> task = |(int32)23|;
int32* result23 = task.run.result;
```

In this sample code, the result of the future is requested without being stored previously and accessed via address. Hence, the code conflicts with the definition of the translation of **result** given beforehand. A first approach to solve this problem would be to change the line to:

```
int32* result23 = (&(task.run))->result;
```

This, however, is not allowed because **task.run** is no lvalue [52, pp. 147-148] which prohibits the utilization of the address operator on this expression. Instead, in order to allow for chainings like **task.run.result**, two wrapper functions, one for each **join** and **result**, are provided. These functions each take a future by argument, thus binding it to an addressable location, and call the generated functions that correspond to **join**, respectively **result**, in turn:

```
void* saveFutureAndGetResult(Future future) {
  return getFutureResult(&future);
}

void saveAndJoinVoidFuture(VoidFuture future) {
  joinVoidFuture(&future);
}
```

By making use of the presented functions, the reductions of **e.run**, **e.join** and **e.result** (where **e'** is the reduced value of **e**) directly become function calls thereof:

```
runTaskAndGetFuture(e')
```

```
runTaskAndGetVoidFuture(e')
```

```
((t)getFutureResult(&e'))
```

```
joinFuture(&e')
```

The type cast of the result returned by **getFutureResult(&e')** is necessary since it returns a generic pointer of type **void*** which may not be compatible with the receiver of the value. In consequence, the result is cast to the result type of the future for which **e.result** was type checked. For expressions of the kind **|e|.run**, the reduction to a call of **runTaskAndGetFuture()** is not applied. Instead, a call to a specific function **futureInit_X** that combines both the logic of the task initialization and the future initialization is created. If **e** contains references to local variables or arguments, then similar to the **taskInit_X()** expression block from section 3.1.2, the future initialization function first allocates memory from the heap for the values of the referred variables. To this end, it uses an instance of the **Args_X** struct that was created for the task to store the values. Afterwards, instead of wrapping the arguments struct inside a **Task** struct instance, the function directly declares a pthread and initializes it with the arguments and a pointer to the function **parFun_X** that was created for the task:

```
Future futureInit_X(t_1 v_1, ..., t_n v_n) {
  Args_X* args_X = malloc(sizeof(Args_X));
  args_X->v_1 = v_1;
  ...
  args_X->v_n = v_n;
  pthread_t pth;
  pthread_create (&pth, null, :parFun_X, args_X);
  return (Future){ .pth = pth };
}
```

If **e** does not contain any references to local variables or function arguments the arguments-related code is omitted and **null** is given as argument parameter to **pthread_create()**:

```
Future futureInit_X() {
  pthread_t pth;
  pthread_create(&pth, null, :parFun_X, null);
  return (Future){ .pth = pth };
}
```

If the type of **e** is **void**, the struct type **Future** is replaced by **VoidFuture** in either declaration of **futureInit_X**. The code shows why no clearance of the arguments for a task in the case of a direct run of the task is required, as was mentioned in section 3.1.1: Since the struct instance **args_X** is only used for exactly one running task, no further copies of the arguments are needed. Hence, the freeing of the heap-allocated memory can be left to the function **parFun_X**. **futureInit_X** is called in the reduction of **|e|.run**:

```
futureInit_X(v_1, ..., v_n)
```

Whereas the following section will show the generation of expressions **e.run** to calls of **runTaskAndGetFuture** the reduction to a call of **futureInit_X** is depicted in section 3.3.4.

3.2.3 Example Code

The running example concerning the π approximation from section 3.1.3 can now be extended with the result-related code. The tasks that were previously declared – and can be seen as templates for according running instances of task – are used to initiate a running task instance of each:

```
int32 main(int32 argc, string[] argv) {
  Task<long double*>[RANGECOUNT] calculators;
  Future<long double*>[RANGECOUNT] partialResults;

  ... // task declarations

  for (i ++ in [0..RANGECOUNT[) {
    partialResults[i] = calculators[i].run;
    calculators[i].clear;
  }

  for (i ++ in [0..RANGECOUNT[) {
    result += *(partialResults[i].result);
    free(partialResults[i].result);
  }
}
```

For every task that is run, a future of the same type is created. After the initialization, the tasks (i.e. the task templates, not the running instances thereof) are cleared in order to avoid memory leaks. The programmer is free to choose whether he is willing to do so. If only a very limited amount of memory is used by the tasks, the clearance might not be deemed necessary. In the second loop, the futures of all running tasks are used to retrieve and accumulate all partial π results. In the end their memory is freed since they are located in the heap. Like with tasks, the freeing of future results is up to the programmer and, at the end of the program, might not be even useful. The translation of the new code becomes:

```
int32 main(int32 argc, string[] argv) {
  Task[RANGECOUNT] calculators;
  Future[RANGECOUNT] partialResults;

  ... // task declarations

  for (int8 __i = 0; __i < RANGECOUNT; __i++) {
    partialResults[__i] = runTaskAndGetFuture(calculators[__i]);
    free (calculators[__i].args);
  }

  for (int8 __i = 0; __i < RANGECOUNT; __i++) {
    result += *(((long double*) getFutureResult(&partialResults[__i])));
    free((long double*) getFutureResult(&partialResults[__i]));
  }
```

As the code shows, the future type becomes the type of the according generic **Future** struct. Like the translation of the task type it loses the type parameterization **long double***, which is not necessary any more. The task running expression **calculators[i].run** and result retrieval expression **partialResults[i].result** are translated to calls of the newly generated generic functions **runTaskAndGetFuture()**, respectively **runTaskAndGetFuture()**. Their definitions can be found in the previous section 3.1.2. Due to its genericity **runTaskAndGetFuture()** returns a result of type **void***. In consequence, its result must be cast by the compiler to an appropriate pointer type which in this case is **long double*** like in the original task and future definitions. Finally, the task clearance is simply reduced to a call of C's free function, which is given the pointer to the argument struct instance that the task **calculators[i]** holds.

3.3 Shared Memory

The previous chapters introduced the means to enable parallel execution of code in terms of tasks and futures. Still missing from a discussion of ParallelMbeddr's building blocks is the communication between tasks. The communication model of choice for ParallelMbeddr is shared memory. The reason for this choice follows from the objectives for a communication model: it should offer a reasonable performance, considering that it is supposed to be used in embedded systems; it should be reasonably safe by design in order to avoid the trip hazards that are involved with low-level synchronization approaches like mutexes. For performance reasons, transactional memory seems not to be ready for the embedded domain for performance reasons. By following argumentation, message passing does not offer profound advantages in comparison to shared memory if the access to the shared memory is controlled in a sound way. Usually message passing forbids shared memory between two parallel units of execution. Instead, communication is realized via message sends. A strict separation of memory does not fit the usual C work-flow concerning pointer arithmetic. Therefore, in order to reduce performance loss, some form of memory sharing would have to be introduced into a message passing model. As this would lead to the same problems that already arise with the general shared memory model, the opposite way is chosen: Instead of introducing shared memory in a message passing model, a shared memory model is designed, on top of which message passing can be attached in order to simplify the communication between tasks.[27]

Memory that is to be shared between two tasks must be explicitly declared by an according type. A variable of this type denotes a *shared resource*. Thus, a shared resource can be regarded as a wrapper of data that is to be shared. In order to make use of a shared resource it has to be synchronized first. Specific language elements are used to access and change the value of a shared resource. The chosen approach enables the programmer to use shared data both globally and locally and, like with any other data, nest it inside structs and arrays. Additionally, the new data type enables the IDE to ensure – despite the arbitrary structuring of shared resources – that data is shared in a sound[28] way.

3.3.1 Design

The resources to be shared are typed with the shared resource type:

$t ::= ... \mid shared{<}u{>} \mid shared{<}shared{<}u{>}{*}{>}$

$u ::= t \qquad void \neq u \neq t*$

The type parameterization denotes the base type of a shared type, i.e. the type of the data that is wrapped by a shared resource. The base type of a shared resource can be either of mbeddr's C-types, with two exceptions. It must not be **`void`** because a shared resource has to wrap an actual value. Furthermore due to reasons that will be explained in 3.4, the base type may not denote a pointer to a value that is not shared.[29] Further restrictions apply to shared types, which are also discussed in the aforementioned section. The same applies to the **`.set`** expression by which the value of a shared resource can be modified; the value can be retrieved via **`.get`**:

$e ::= e.get \mid e.set(e)$

$$\text{SharedGet}\ \frac{e \vdash shared{<}t{>}}{e.get \vdash t} \qquad \text{SharedSet}\ \frac{e \vdash shared{<}t{>} \quad e' \vdash t' \quad t' <: t}{e.set(e') \vdash void}$$

In order to access (read or write) the value via **`.get`** or **`.set`** it must be synchronized first. For this purpose the syntax `stmts` of statements in mbeddr is extended by the synchronization statement **`sync`**.

[27] Such an extension would be a future concern and is not implemented in this work.

[28] 'Sound' in this context means 'atomic'. Only one task at a time should be able to access a shared datum.

[29] In other words: A pointer wrapped in a shared resource must point to a shared resource itself. Due to the comprehensive type system that mbeddr is equipped with and that is only partially existent in C99, the compatibility of base types of shared types was mainly tested with the primitive types of C99 as well as pointer types, array types, struct types and type definitions (typedefs). Future work will have to be done to demonstrate and establish full compatibility with the rest of mbeddr's types.

sync contains a *synchronization list* of shared resources to synchronize and a block of statements whose referenced shared resources may be synchronized by a surrounding synchronization statement:

stmt ::= ...| *sync*(*res*, ..., *res*){ *stmt*...*stmt* }

`res` denotes the syntax of possible *synchronization resources*. Each synchronization resource **res** wraps an expression **e** which evaluates to a shared resource. **e** can be either of type **shared<t>** or of type **shared<t>***. A shared resource can be synchronized as it is (as a synchronization resource) or be named (the synchronization resource then becomes a *named resource*):

res ::= *e* | *e as* [*resName*]

The latter allows the programmer inside the **sync** statement to refer to the result of an arbitrary complex expression, which evaluates to a shared resource, inside the synchronization statement. Hence, a *named resource* (i.e. a synchronization resource with a name) can be seen as syntactic sugar for a local variable declaration, initialized with a shared resource, and a synchronization resource with a reference thereof.[30] The type of such a reference is given by the shared resource that the expression of the named resource evaluates to. Due to the copy semantics this type is restricted to **shared<t>***. This enforcement ensures that the named resource actually synchronizes the original shared resource and not a copy thereof, of which a synchronization would be useless. The scope of a named resource is restricted to the according synchronization statement. More precisely, a named resource *n* of a synchronization statement *s* can be referenced from anywhere inside the abstract syntax tree (AST) of the statement list of *s*. Furthermore it can be referenced from within the expression of any synchronization resource that follows *n* in the synchronization list of *s*, e.g.:

```
shared<shared<int32>> v;
// vContent is declared before it is used in the list => valid
sync(v, &(v.get) as vContent, &vContent->get as vContentContent) {
  vContentContent->set(0);
}
// vContentContent is not in scope, here => invalid
vContentContent->set(1);
```

This restriction results from the lexical scoping of synchronization resources: In order to determine whether the target of **.get** or **.set** is synchronized, ParallelMbeddr checks whether the target is a reference to a visible and synchronized variable or to a named resource. This implies that shared resource expressions other than variable references (e.g. paths and function calls) must be bound by named resources, in order to be able to access their wrapped values via **.get** or **.set**.

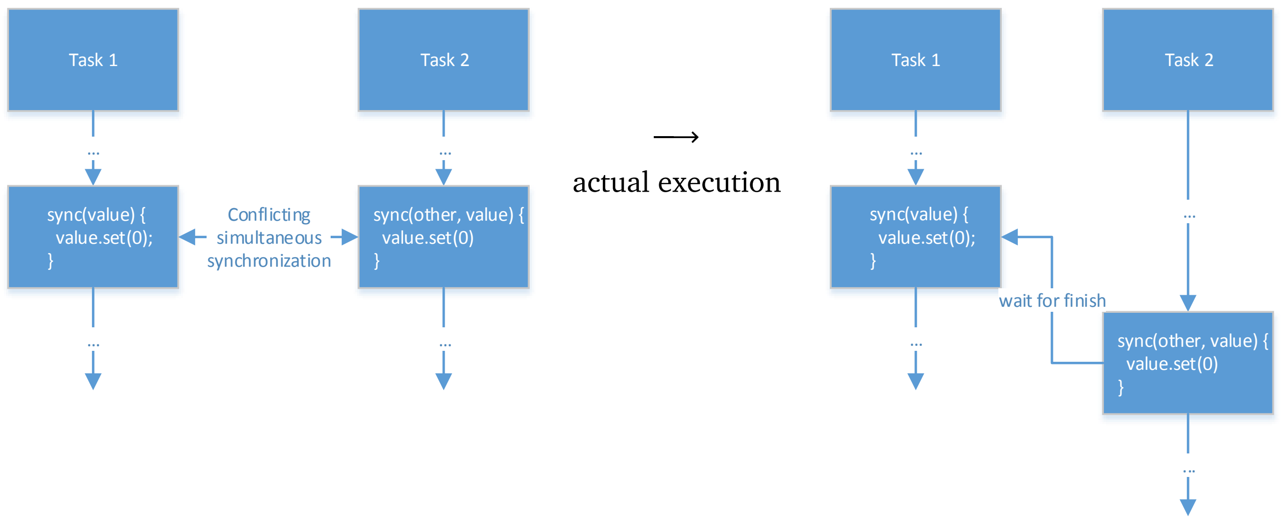
In contrast to Java's synchronization blocks and methods [59, p. 279], the synchronization of tasks is not computation oriented, but data oriented. The crucial difference is that a synchronized block **A** in Java is only protected against simultaneous executions by multiple threads. Thus, it is valid to access the data that is involved in **A** by some other computation whose protecting block (if any) is completely unrelated to **A**. Since low-level data races can obviously not be guaranteed to be excluded with this scheme, ParallelMbeddr ties the protection to the data that is to be shared. Every shared resource, i.e.

[30] Actually, due to eager evaluation (i.e. expressions are evaluated as soon as they are encountered) a named resource provides different semantics than a local variable as will become clear later on.

block of shareable data, is therefore protected separately and application-wide. ParallelMbeddr does not yet face higher-level data races, which involve multiple shared resources. Dependencies between shared resources must be resolved by the programmer who as to wrap the dependent data inside a single shared resource, e.g. as fields of a struct. Concluding, if two synchronization blocks, which are about to be executed in parallel, contain synchronization references that overlap in terms of their referenced shared resources, their executions will be serialized. For instance let *t1* and *t2* be two tasks that have access to the same shared resource which is referenced by a global variable **value**. *t1* wants to synchronize **value**, simultaneously *t2* wants to synchronize **valuePointer** which points to **value**'s shared resource and some other shared ressource that is available via the variable **other**:

```
shared<int32> value;
shared<int32>* valuePointer = &value;
shared<double> other;
```

The synchronization semantics will then cause one thread to wait for the other to finish the execution of the blocking synchronization statement before it starts the execution of its own synchronization statement. The execution order might therefore be changed in the following manner:

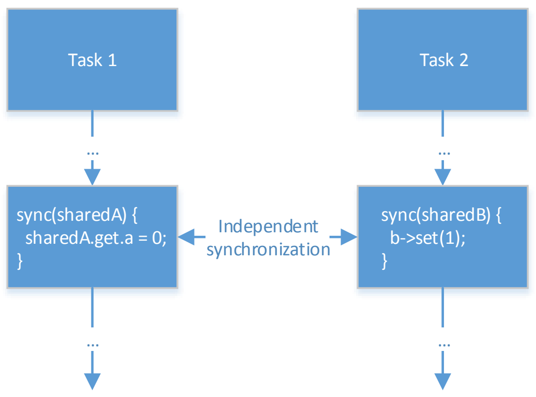

Task 1
Task 2
sync(value) { value.set(0); }
Conflicting simultaneous synchronization
sync(other, value) { value.set(0) }
actual execution
Task 1
Task 2
sync(value) { value.set(0); }
wait for finish
sync(other, value) { value.set(0) }

The possibility to refer to multiple shared resources in the synchronization list of a synchronization statement is not mere syntactic sugar for nested synchronization statements. Instead, the semantics of a synchronization list are that all referenced shared resources are synchronized at once, but with a possible time delay. Due to the design of the underlying implementation, deadlocks, as a result of competing synchronization statements, are thus avoided.[31]

As a result of the fact that generally the access to shared resources is resource-centric, a value wrapped in a shared resource which in turn contains nested shared resources is independently protected from the latter. Therefore, a shared resource of a struct with a shared member **b** is independently synchronized from **b**:

[31] Nevertheless, deadlocks can obviously still occur if nested synchronization statements compete for the same resources in an unsorted order.

```
struct A {
  int32 a;
  shared<int32> b;
}
shared<A> sharedA;
shared<int32>* sharedB;
sync(sharedA) { sharedB = &(sharedA.get.b); }
```



The translation of shared resources is currently only supported for executable programs. Therefore, it is not safe to use shared types and synchronization statements in libraries that are written with mbeddr. The explanations of the following section are thus limited to according scenarios, although the translation for library code would only change in details.

3.3.2 Translation of Shared Types

In order to fully understand the translation of synchronization statements, the translation of shared types is given first. For the implementation of shared types in C, two main solutions are conceivable, which differ in the coupling that they exhibit between the data that is to be shared and the additional data required for access restriction, i.e. synchronization. In any case, a solution must make use of additional data that can be used to synchronize two threads which try to read or write the shared data. To this end, the most basic synchronization primitive was chosen: each protected data item is assigned exactly one mutex.

In the first solution, the data to be shared is stored as if no protection scheme existed at all. Additionally, all mutexes that are created by the application are stored in one global map, which indexes each mutex by the memory address of its corresponding shared datum. This approach offers the advantage that access to the data itself is not influenced by the mutex protection: Every reference to the value of a shared resource **e.get** can directly be translated into a reference to the wrapped value. Additionally, since the mutexes are globally managed, all data that is returned by a library can be easily made (pseudo-) synchronization safe. For instance, if a pointer to an arbitrary memory location `loc` is returned, the pointed-to address can be used to create a new mutex and add a mapping to the global mutex map. However, this on-the-fly protection of memory locations can incur synchronization leaks: The compiler cannot guarantee that addresses returning functions with unknown implementation will not leak their returned values to some other computation which accesses the according memory unsynchronized. This implies that such protection would only be safe if any reference to `loc` was wrapped in some shared resource which, in this scenario, is not feasible. Hence, a design was chosen that does not allow for such protection. Consequently, a global map would not be beneficial in this regard. A map solution would entail a space-time trade-off. For instance, Google's C++ *dense_hash_map*, [32] provides comparatively fast access to its members but imposes additional memory requirements in comparison to slower hash map

[32] http://goog-sparsehash.sourceforge.net/doc/dense_hash_map.html

implementations[33]. A disadvantage of hash maps is the non-deterministic performance[34] and increased access time that may be induced by hash collisions [39].

The second solution for the implementation of shared types keeps each shareable datum and its mutex together. An instance of a struct with member fields for both components is used in place of the bare datum to be shared. In contrast to the aforementioned solution, a reference to the value **`e.get`** needs one level of indirection via the struct instance. However, the access to a mutex is simplified. As with the value, it can be retrieved by a member access to the corresponding struct field, whereas the map solution requires a map lookup to get the mutex (plus additional delay to make any modifying access to the map thread-safe). The computational overhead imposed by a struct member lookup is deemed negligible in the overall application performance. On the other hand, the space required for the struct equals that of the individual fields (value and mutex) plus additional padding [44, pp. 303 ff.] that is required to arrange the struct fields along valid memory addresses. The latter depends on the size of the data to be stored. In order to keep the padding as small as possible, smart member ordering should be applied.

For this work, the struct solution was chosen in order to keep the access time of mutex lookups small and deterministic while not imposing too much space and computation overhead for datum lookups. For each kind of shared type **`shared<t>`** with the translated base type **`t'`** a separate struct is generated:

```
struct SharedOf_t {
  pthread_mutex_t mutex;
  t' value;
}
```

Any nested shared types are, hence, translated first. For implementation reasons nested *typedef*s and constants used in array types are also resolved in this process.[35] Depending on the base type of the shared type, the generated struct declaration is stored either in a generic module or in a newly generated other module. The following illustration depicts the tree for a type **`shared<t>`** whose nodes are made of types and whose edges are formed by the base type relationship of shared types, array types and pointer types,[36] e.g:

[33] See http://incise.org/hash-table-benchmarks.html for details. This examplary library should be considered as an illustrative example for the general space-time trade-off that is inherent with maps.

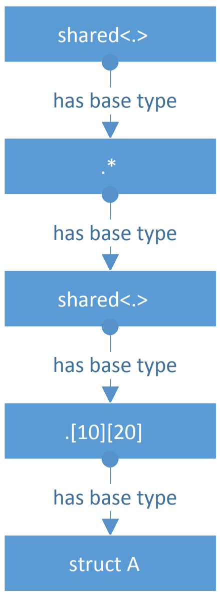
[34] Although real-time applications with their time-wise constraints are not a primary concern of this work, it should nevertheless be kept in mind that they are part of the embedded domain. Therefore, a solution that considers the characteristics of this domain is at least regarded advantageous.

[35] Although the resolution of typedefs and constants impedes corresponding (test-wise) adjustments made by the programmer in the translated C code, this is not deemed an issue since generally, changes should always be made from within mbeddr, i.e. on the original code.

[36] Clearly, this tree will have only one branch.

```
struct A { int32 val; }
shared<shared<A[10][20]>*>
```

$\longrightarrow$ the simplified type AST



If the leave of the tree is a primitive C type the struct declaration is stored in a generic module. In any other case the leave must be some struct type **s**. In order to preserve the visibility of the corresponding struct in the newly generated struct **SharedOf_t**, the latter is stored in the same module. Since the generation of code related to shared types can diminish the legibility of the resulting code profoundly for every such module, a specific *SharedTypes_X* module is created and imported into the module that declares **s**. *SharedTypes_X* is used to store struct declarations like **SharedOf_t**. Furthermore **s** is lifted into it in order to make it visible in the member declaration **value** of **SharedOf_t**. For any field of **s** whose type tree contains another user-defined struct type, the corresponding struct declaration is either lifted as well (and recursively treated in the same way), or imported by its module. Should this separation of generated code used for shared type declarations and other user-defined code not proof well in praxis, it could easily be deactivated. With the former generation of the struct declaration in mind a type **shared<t>** is reduced to:

```
SharedOf_t
```

The reduction of an expression **e.get** and **e.set(f)** make use of the **value** field of the **SharedOf_t** struct. They are basically reduced to a retrieval of the field, respectively an assignment to the field:

```
e'.value
```

respectively

```
e'.value = f'
```

If **e'** has a pointer type, the expressions **e->get** and **e->set(f)** are reduced to:

```
e'->value
```

respectively

```
e'->value = f'
```

The mutex of the struct **SharedOf_t** in the equally named struct field **mutex**, which is used to synchronize one variable of an according type, must be initialized prior to any usage. This is done implicitly by generated code, in order to free the programmer from this task. Accordingly, mutexes must be released before they get out of scope to prevent memory leaks. In the generated code, both functionalities make use of corresponding functions, i.e. for every type who is one of the following types a pair of **mutexInit**–**mutexDestroy** functions is generated:

- shared types whose base types are shared types or for whom mutex functions are recursively generated;
- array types which are not base types of array types themselves and whose base types are either shared types or struct types for whom mutex functions are recursively generated;
- struct types whose structs contain at least one field with a type for which the same relation holds as for the aforementioned types.

For example, a type **shared<int32>[42][24]** would enforce the generation of one mutex initialization and one mutex destruction function. **shared<int32>*** on the other hand would not. Any variable **v** of the latter type would point to a shared ressource which must be referenced directly by another variable **v'** of type **shared<int32>** or be contained in the memory-addressable value of some variable **v"** of a more complex type. The declaration of **v'**, respectively **v"**, would then trigger the initialization of the mutex of the shared resource that **v** points to. The resulting mutex initialization functions for types of the aforementioned kind are declared as follows:

```
// for a proper shared type shared<t> and the type t' that t is reduced to
void mutexInit_X(SharedOf_t'* var) {
  pthread_mutex_init(&var->mutex, &mutexAttribute);

  // either if t is a shared type or a struct type:
  mutexInit_X'(&var->value);

  // or if t is an array type t[i_1]...[i_n] of 1 to n dimensions:
  mutexInit_X'((SharedOf_t'*...*)var->value, i_1, ..., i_n);
}
```

For a shared resource, first the mutex of the corresponding translated struct is initialized by a call to the POSIX function **pthread_mutex_init()**, which takes a mutex pointer and a mutex attribute pointer.[37] Then – since by aforementioned conditions the shared resource contains another shared resource – a call to the appropriate mutex function for the contained value is triggered. Depending on whether the base type of the current shared type is an array additionally the dimension sizes for this base type may need to be provided as well (see below for details).

```
// for a proper array type t[]...[] of 1 to n dimensions where '...' denotes the occurrence of an accor
// and t' denotes the reduced-to type
void mutexInit_X(t'*...* var, int32 size_0, ..., int32 size_n) {
  for (int32 __i_0 = 0; __i_0 < size_0; __i_0++) {
    ...
      for (int32 __i_n = 0; __i_n < size_n; __i_n++) {
        // in case t is a struct type
        mutexInit_X'(&var[__i_0]...[__i_n]);

```

[37] The meaning of the mutex attribute will be explained later on.

```
    // or, in case t is a shared type with generic C base type:
    pthread_mutex_init(&var[__i_0]...[__i_n].mutex, &mutexAttribute);
   }
  ...
 }
}
```

For shared ressoures that are nested in arrays, a nested iteration over all elements of the corresponding possibly multidimensional array with calls to either the generated mutex functions or the function defined by the POSIX standard are triggered.
For structs with nested shared resources for each field that is or contains a shared resource, either the **`pthread_mutex_init()`** function is called directly or the mutex initialization is done by a call to the already generated function that is type compatible with this field (by possibly providing additional array dimensions):

```
// for a proper struct type t of a struct t { u_1 f_1; ...; u_n f_n } and according reduced field types
void mutexInit_X(SharedOf_t* var) {
 ...
 // in case u_i demands further initialization and it is a struct type or a shared type
 mutexInit_X'(&var->f_i);

 // in case u_i demands further initialization and is an array type u_i[j_1]...[j_n] of 1 to n dimensi
 mutexInit_X'((SharedOf_u_i'*...*)var->f_i, j_1, ..., j_n);

 // or, in case u_i is a shared type with a generic C base type:
 pthread_mutex_init(&var->f_i.mutex, &mutexAttribute);
 ...
}
```

The signature of **`mutexInit_X()`** for arrays shows that those are not passed as arrays to the mutex functions, but as pointers. This is due to the necessity of declaring multidimensional arrays at least partially with the size for each dimension (e.g. **`int[][]`** would be missing at least one dimension size). Nevertheless it would not make sense to declare one mutex function for each shape of dimension size. Since arrays are treated like pointers internally when they are passed as function arguments, it is completely safe to cast them to appropriate pointer types and to use equal types for the according function parameters.
The deletion of mutexes is defined quite similar to the initialization, with the main difference that the utilized according pthreads function only takes one mutex. Therefore, only the deletion of mutexes nested in resources of shared types is shown:

```
void mutexDestroy_X(SharedOf_t'* var) {
 pthread_mutex_destroy (&var->mutex);
 // ... further call to a mutexDestroy_X function equivalent to mutexInit_X shown above
}
```

The presented functions are used to initialize mutexes at the beginning of their life span and delete them right before the corresponding end. For mutexes refered to by global variables this means that they must be initialized at the beginning of the entry function of the program.[38] As already forced for executable programs by mbeddr, the programmer thus has to specify a main function. Similarly, mutexes of local variables are initialized right after their declaration, whereas mutexes of function arguments are declared at the beginning of the related function.[39] The deletion of mutexes for shared resources must be accomplished before they get out of scope which, again, depends on the kind of variables they are referred to.

Mutexes for global variables need not be destroyed at all in order to avoid memory leaks. Their lifetimes span the whole program execution so that the memory allocated to these mutexes will be cleaned up by the underlying operating system. On the other hand, the mutexes of local variables must be destroyed before they get out of scope. Hence, mutex destruction calls are added at the end of the surrounding scopes of mutexes. If control flow breaking statements (*return*, *break*, *continue*, *goto*) are present, additional calls are inserted according to the following rules. Let c denote a control-flow-breaking statement that occurs in the AST of the same function as the declaration l of some local variable, which refers to a (nested) shared resource. If c is part of the AST of any statement that follows l[40] and

- c is a *return* statement and refers to a function or a closure whose AST contains l or
- c is a *break* statement and refers to a loop or a *switch* statement case whose AST contains l or
- c is a *continue* statement and refers to a loop whose AST contains l or
- c is a *goto* statement and refers to a label outside the AST of any statement that follows l,[41]

c must be preceded with a destruction call of the mutex of the shared resource of l (compare with the synchronization stopping rules below). Allocated memory that is not freed at the end of the program is automatically released by the operating system. Hence, a *return* statement that refers to the entry function of the program need not be preceded with destruction calls of its local shared resources. The proper destruction of a mutex of an argument of function f just requires according function calls at the end of f and before any return statement that refers to f.

[38] Due to the way mutexes are used in ParallelMbeddr (recursively and nested in structs – as will be shown in the following paragraphs) and the peculiarities of C and the POSIX standard, there is no way to combine the definition and the initialization of mutexes.

[39] It is not an obvious choice to enable the programmer to use function arguments which contain or are shared resources and: C's pass-by-value semantics of function parameters causes parameters to be copied into functions. Therefore, a shared resource which is not passed by its addresses but by its actual value will provoke the generation of another, equal shared resource at the beginning of the function execution. This copy is synchronization-wise completely unrelated to the original shared resource, since mutex copies cannot be used to lock with their origins [5]. Furthermore, they have to be initialized and destroyed separately. The use of shared resources in such a manner can confuse programmers who are not aware of this fact. Nevertheless ParallelMbeddr allows this kind of utilization of shared resources in order to not burden the programmer with having to copy large structs that contain shared resources component-wise if the shared resource data is not of relevance. Depending on the feedback of future users, it should be considered whether warnings for unintended misuse of shared resources in this way might be helpful.

[40] In other words, only those control-flow-breaking statements cs that are either one of the statements *stmts* which have the same AST parent p and follow l in the statement list of p or are contained in the AST of some stmt are considered.

[41] Since *goto* statements can considerably disorganize the program flow they should be only used with great care in ParallelMbeddr. Specifically the initialization of mutexes is not protected against misuse of these statements.

As inside the declarations of the mutex destruction functions explained above, the actual function to call for a variable or argument is either one of the **mutexDestroy_X()** functions or a direct call of **pthread_mutex_destroy()** for 'simple' shared resources of generic C types, e.g.:

```
// simple shared resource
shared<int32> v1;

// complex shared resource
shared<int32>[2][3] v2;
```

$\Longrightarrow$

```
SharedOf_int32_0 v1;
pthread_mutex_init(&v1.mutex, &mutexAttribute);

SharedOf_int32_0[2][3] v2;
mutexInit_0((SharedOf_int32_0**)v2, 2, 3);
```

To recap: The mutex of a shared resource is either directly initialized and destroyed via appropriate pthreads functions or it is indirectly handled via functions that are based on the types of the values that shared resources are nested in. This approach was chosen in order to reduce the amount of code duplication that would occur if mutexes of shared resources would be handled inline for every according variable. As a result the generated code's readability is enhanced. The additional computational overhead due to function calls and returns should be regarded as an optimization concern of a further compilation step by a compiler like *gcc*.

ParallelMbeddr does not prevent the programmer from structuring the synchronization statements in such a way that a task will synchronize a shared variable multiple times (*recursive synchronization*). The following code depicts such behavior:

```
shared<int32> sharedValue;
sync(sharedValue) {
  sync(sharedValue) {
    sharedValue.set(42);
  }
}
```

Since each synchronization statement locks the mutexes of the refereed shared resources (see below for details), a recursive synchronization results in a *recursive lock* of the corresponding mutex. Mutexes as defined by the POSIX standard must be specifically initialized in order to allow for this behavior:[42] A mutex attribute that specifies the recursiveness must be defined and initialized first. It can then be used by any number of mutexes. For this purpose, an application-wide attribute is defined in a generic module that is imported by all user-defined modules. It is initialized at the beginning of the main function:

```
// inside the generic module:
pthread_mutexattr_t mutexAttribute
// at the beginning of main:
pthread_mutexattr_init(&mutexAttribute);
pthread_mutexattr_settype(&mutexAttribute, PTHREAD_MUTEX_RECURSIVE);
```

[42] By default a recursive lock results in undefined behaviour because a default mutex does not have a lock count which is required to make recursive locks work: http://linux.die.net/man/3/pthread_mutex_trylock.

3.3.3 Translation of Synchronization Statements

Every synchronization statement is reduced to its statement list – as a block –, surrounded with calls to functions that control the synchronization of the mutexes. The reduction of such a statement is given either by

```
sync(e) stmt_list                  ⟹        startSyncFor1Mutex(&e.mutex);
                                            stmt_list'
                                            stopSyncFor1Mutex(&e.mutex);
```

in case it contains only one synchronized resource, or else by

```
sync(e_1, ..., e_n) stmt_list      ⟹        startSyncForNMutexes(&e_1.mutex, ..., &e_n.mutex);
                                            stmt_list'
                                            stopSyncForNMutexes(&e_1.mutex, ..., &e_n.mutex);
```

The statements are kept inside their statement list block in order to preserve the scope of local variables inside synchronization statements. A synchronization statement list block is reduced to another block where statements that break the program flow structure may be preceded by an identical call of the **stopSyncForNMutexes()** function as is present after the list: Let *s* be a synchronization statement and *c* be a control-flow-breaking statement which is nested on some level in the AST of *s*' statement list. Then *c* is preceded with a call to **stopSyncForNMutexes()** if one of the following cases holds:

- *c* is a *return* statement and refers to a function or a closure whose AST contains *s*;
- *c* is a *break* statement and refers to a loop or a *switch* statement case whose AST contains *s*;
- *c* is a *continue* statement and refers to a loop whose AST contains *s*;
- *c* is a *goto* statement and refers to a label outside the AST of *s*.[43]

In this manner, inconsistent synchronization states of shared resources due to a control flow break by the aforementioned statements are omitted. Since the occurrence of such a statement may also force ParallelMbeddr to insert **mutexDestroy_X()** calls, a careless mixture of mutex unlocking and destruction calls can cause runtime errors [5]. The generator therefore ensures that any destruction calls are put behind the generated unlocking calls.
For each arity of synchronization resources, separate versions of the **start**- and **stopSyncForNMutexes()** functions are declared inside a generic C module. A **stopSyncForNMutexes()** function straightforwardly redirects its mutex parameters to calls of the **pthread_mutex_unlock** function:

[43] Similarly to the aforementioned problem with mutex management functions, *goto* statements can impair synchronization states and should therefore be used with great care. This holds particularly for *goto* statements which cause jumps into synchronization statements from outside.

```
// the corresponding function 'stopSyncFor1Mutex()' for exactly one mutex is skipped here
void stopSyncForNMutexes(pthread_mutex_t* mutex_1, ..., pthread_mutex_t* mutex_n) {
  pthread_mutex_unlock (mutex_1);
  ...
  pthread_mutex_unlock (mutex_n);
}
```

Abstracted from the details of the actual implementation, synchronization statements synchronize their ressources atomically, as was mentioned in the preceding design section. Since one or more mutexes can be tentatively locked by multiple threads simultaneously, specific contention management has to be conducted. The illusion of atomic synchronization is realized by an implementation of the obstruction-free[44] busy-wait protocol *Polite*. In order to resolve conflicts, *Polite* uses exponential back-off. The according back-off function is explained further down. The synchronization function tries to lock every mutex as given by its arguments. On failure, it releases every mutex that was locked so far, uses the back-off function to delay its execution for a randomized amount of time, and repeats afterwards. This scheme enables competing threads to (partially) proceed and avoid deadlocks due to unordered overlapping mutex locks.[45]

```
// again, the equivalent function declaration for one mutex is skipped
void startSyncForNMutexes(pthread_mutex_t* mutex_0, ..., pthread_mutex_t* mutex_m, pthread_mutex_t* mut
  uint8 waitingCounter = 0;
  uint16 mask = 16;
  uint32 seed = (uint32)(uintptr_t) &waitingCounter;

  while (true) {
    if ([| pthread_mutex_trylock (mutex_0) |] != 0) {
      backoffExponentially(&waitingCounter, &mask, &seed);
    }
    else if ([| pthread_mutex_trylock (mutex_1) |] != 0) {
      [| pthread_mutex_unlock (mutex_0) |];
      backoffExponentially(&waitingCounter, &mask, &seed);
    } ...
    else if ([| pthread_mutex_trylock (mutex_n) |] != 0) {
      [| pthread_mutex_unlock (mutex_m) |];
```

[44] Busy-waiting means that the thread will repeatedly test a condition until it is met, without doing actual useful work [49, p. 166]. Thus, it is an alternative to suspending a thread and revoking it later on when some condition is met (which can, e.g., be realized by *condition variables* as provided by POSIX threads [17, p. 77]). Obstruction-free means that the execution of any thread will progress when at some time the thread is run in isolation, i.e. when the execution of obstructing other threads is interrupted meanwhile. The existence of obstruction-freedom guarantees that no deadlocks will occur [12]. However, livelocks and starvation are not necessarily avoided. Stronger degrees of non-blocking algorithms like lock-freedom and wait-freedom tackle these problems (partially), but are not relevant for this work. Further information on the latter is for instance provided by http://preshing.com/20120612/an-introduction-to-lock-free-programming/.

[45] Again, the presented scheme does not prevent the programmer from nesting the synchronization statements in such a manner that deadlocks in nested synchronization statements occur. It is rather a prevention of deadlocks that are caused solely by synchronization statements on the same nesting level.

```
      ...
      [| pthread_mutex_unlock (mutex_0) |];
      backoffExponentially(&waitingCounter, &mask, &seed);
    }
    else {
      break;
    }
  }
}
```

The back-off realized by Polite delays the execution by less than $limit = 2^{n+k}$ ns [58] in a randomized fashion. n denotes the retry counter and k denotes some constant offset which can be machine-tuned. The randomized wait time of the exponential back-off is used to avoid livelocks which could happen if two threads would repeatedly compete for the same resources and delay their execution for equal amounts of time. In the current implementation **backoffExponentially()** of the contention management, the offset k is set to 4 and a threshold m of 17 denotes the number of rounds after which k is reset.[46] Thus, maximum delays of about 100 ms (specifically 131 ms) are allowed.[47]

```
inline void backoffExponentially(uint8* waitingCounter, uint16* mask, uint32* seed) {
  *mask |= 1 << *waitingCounter;
  randomWithXorShift(seed);
  struct timespec sleepingTime = (struct timespec){ .tv_nsec = *seed & *mask };
  nanosleep(&sleepingTime, null);
  *waitingCounter = (*waitingCounter + 1) % 13;
}
```

It has to be noted that **backoffExponentially()** keeps its main state inside the **startSyncForNMutexes()** function. The state will therefore be re-initialized before the execution of every synchronization block. The generation of the pseudo-randomized delay is realized via utilization of the Marsaglia's Xorshift random number generator [45]:

```
void randomWithXorShift(uint32* seed) {
  *seed ^= *seed << 13;
  *seed ^= *seed >> 17;
  *seed ^= *seed << 5;
}
```

The generator was chosen for its high performance, low memory consumption, and thread safety due to the utilization of the stack-managed **seed** parameter as opposed to the global state usage of the standard C random generator **rand()**. The fact that after a certain number – which may be smaller than in other generators – of repeated calls with the same seed value (i.e. the same memory address of a seed)

46 k's value is reflected in the initial value ($2^4 = 16$) of **mask** whereas m's value is composed of mask's base and the divisor (13) in the calculation of the next **waitingCounter**.

47 The search for machine- or application-specific optimal offsets and thresholds is a task for future enhancements of ParallelMbeddr.

repetitions of the sequence of calculated numbers will occur is not of relevance for the purpose of this work.

3.3.4 Example Code

In the previous sections 3.1.3 and 3.2.3 the running example approximated the number π by using tasks which calculate exactly one fraction of the result each and by retrieving their results via futures. The amount of work was therefore partitioned in advance. In this section, a more dynamic approach is chosen instead: The work of every task comprises major and minor rounds. A minor round is equivalent to the full calculation loop in the previous π solution. In every step of a major round, a minor round is initiated by first coordinating with the other tasks which range of π the current task should calculate. After having calculated the sum in a minor round, the task then uses a queue to store its next partial result. A dedicated task is used to collect these partial results from the queue and accumulate them to a complete sum, which eventually becomes the result of the over-all approximated π. The communication-based solution can be seen as a map-reduce implementation where partial results are mapped onto the queue by a certain number of tasks and from there reduced to a final result by a separate task (compare with [20]). In order to understand the new implementation, the following example is given: A thread-safe queue of a certain size and slots of type **long double** is to be viewed as a black box. Further, it has to be supposed that functions for the initialization, for adding a value and getting the next value (or wait for the next value), exist:

```
struct Queue {...}
void queueInit(shared<Queue>* queue);
void queueSafeAdd(shared<Queue>* queue, long double item);
void queueSafeGet(shared<Queue>* queue, long double* result);
```

As in the previous approach, the amount of work to be done is defined by a range size (number of minor task rounds) and the number of ranges altogether. Additionally, the number of mapper tasks is set to a certain value that should be in the order of the number of processors:

```
#constant BLOCKSIZE = 300000000;
#constant BLOCKCOUNT = 4;
#constant THRESHOLD = BLOCKSIZE * BLOCKCOUNT;
#constant MAPPERCOUNT = 2;
```

These constants are used to initialize the mappers and the reducer in the main function appropriately:

```
exported int32 main(int32 argc, string[] argv) {
  shared<Queue> queue;
  queueInit(&queue);
  shared<Queue>* queuePointer = &queue;

  shared<uint32> counter;
  shared<uint32>* counterPointer = &counter;
  sync(counter) { counter.set(0); }

  Task<void> mapperTask = |map(THRESHOLD, counterPointer, queuePointer)|;
  Future<void>[MAPPERCOUNT] mappers;
  for (i ++ in [0..MAPPERCOUNT[) {
    mappers[i] = mapperTask.run;
  }
  mapperTask.clear;

  shared<long double> result;
  shared<long double>* resultPointer = &result;

  |reduce(RANGECOUNT, resultPointer, queuePointer)|.run.join;

  return 0;
}
```

First the queue is defined as being shared in order to be accessible by all tasks. After the initialization, a pointer to the queue is created, which is necessary since any other reference inside a task expression to the queue variable would otherwise cause a copy of the queue struct instance into the task, regardless of whether later on the address of the referred value is retrieved via the **&** operator. This ‘shortcoming’ of the current semantics could be addressed by the introduction of a new address operator that creates a temporary variable of the addressed value and should be considered for future extensions of ParallelMbeddr. Similar to the queue, a **counter** variable is introduced which will be used by the tasks to check and communicate how many ranges have been processed so far. Then, one mapper task is defined and initialized with a task expression of a call to the **map()** function. For communication purposes, the mapper gets the queue pointer and the counter pointer. Additionally, although due to the use of constants not necessary in the current example, the mapper task is told the maximum number of items that need to be calculated. This single task template[48] is used to create multiple running tasks and store their handles as futures in a **mappers** array. The reducer uses a pointer to a result value memory location to safe its result. Further, it gets access to the queue via a pointer thereof and is told how many items (**RANGECOUNT**) it shall read from the queue before termination. The code shows the first example of a task expression chain: First the task is declared, then an instance of it is run in parallel and immediately the main task joins the reducer. Since nothing is done in the main task between the run and the join, the serialized execution could also be realized by a simple function call to **reduce()**. For demonstrative purposes, the code was chosen this way nevertheless. The main task solely joins the reducer task, since after its termination every mapper task will also be finished. The **map()** function iteratively calculates complete ranges of fractions of π until the maximum number of items as given by **threshold** is reached:

```
void map(uint32 threshold, shared<uint32>* counter, shared<Queue>* resultQueue) {
  while (true) {
    uint32 start;
    uint32 end;

```

[48] **mapperTask** can be seen as a template for actual running mapper task instances.

```
    sync(counter) {
      start = counter->get;
      if (start == threshold) {
        break;
      }
      uint32 possibleEnd = start + BLOCKSIZE;
      end = (possibleEnd <= threshold)?(possibleEnd):(threshold);
      counter->set(end);
    }

    queueSafeAdd(resultQueue, calcPiBlock(start, end));
  }
}
```

In every iteration, the function synchronizes the shared resources that **counter** points to in order to retrieve its value and increment it by the number of items that **map()** is going to calculate in the current round. It uses the **calcPiBlock()** function that was presented in section 3.1.3 to calculate a partial sum. Afterwards, the result is added to the queue. The **reduce()** function uses the queue to iteratively read all partial results and update the value of the shared resource of the final result accordingly:

```
void reduce(uint32 numberOfItems, shared<long double>* finalResult, shared<Queue>* partialResultQueue)
  sync(finalResult) {
    for (uint32 i = 0; i < numberOfItems; ++i) {
      long double item;
      queueSafeGet(partialResultQueue, &item);
      finalResult->set(item + finalResult->get);
    }
  }
}
```

During the whole calculation, **reduce()** synchronizes the result variable in order to keep the synchronization overhead small. It is able to do this because no other task will try to access the result before the termination of the single reducer task.

In the beginning of the translated main function, the global mutex attribute is initialized, which will be reused for every mutex. Afterwards, the declared queue is initialized:

```
pthread_mutexattr_settype(&mutexAttribute_0, PTHREAD_MUTEX_RECURSIVE);
pthread_mutexattr_init(&mutexAttribute_0);
initAllGlobalMutexes_0();
SharedOf_Queue_0 queue;
mutexInit_2(&queue);
queueInit(&queue);
SharedOf_Queue_0* queuePointer = &queue;
```

The translated struct type **SharedOf_Queue_0** of the original type **shared<Queue>** refers to a struct that contains a field for the protected queue and a mutex field:

```
struct SharedOf_Queue_0 {
  pthread_mutex_t mutex;
  Queue value;
};
```

The mutex initialization function **mutexInit_2()** initializes the mutex of the shared queue and calls another initialization function which in turn initializes any nested mutexes of the **Queue** struct. Similarly, a destruction function is generated:

```
void mutexInit_2(SharedOf_Queue_0* var) {
  pthread_mutex_init(&var->mutex, &mutexAttribute_0);
  mutexInit_1(&var->value);
}
void mutexDestroy_2(SharedOf_Queue_0* var) {
  pthread_mutex_destroy(&var->mutex);
  mutexDestroy_1(&var->value);
}
```

Similar to the declarations for the queue, the mbeddr generator creates declarations of a struct for the **result** variable. The initialization and destruction of **result** is accomplished inline by calls to the pthreads functions, since no nested mutexes for the fields of the struct exist:

```
struct SharedOf_long_double_0 {
  pthread_mutex_t mutex;
  long double value;
};
... // in main()
SharedOf_long_double_0 result;
pthread_mutex_init(&result.mutex, &mutexAttribute_0);
SharedOf_long_double_0* resultPointer = &result;
```

Lastly, the **counter** variable shows how translated shared resources can be synchronized.[49] For the duration of the setting of **counter**'s value, its mutex is locked. The setting of the value is done by an assignment to the **value** field of the generated struct that keeps the value of the shared resource:

```
SharedOf_uint32_0 counter;
pthread_mutex_init(&counter.mutex, &mutexAttribute_0);
SharedOf_uint32_0* counterPointer = &counter;

startSyncFor1Mutex(&counter.mutex);
{ counter.value = 0; }
stopSyncFor1Mutex(&counter.mutex);
```

The mutexes of all shared resources are destroyed at the end of the main function. Although this should not be necessary for local variables of the entry function of the program, the compiler currently does not distinguish between the main function and any other function for which such calls would be necessary

[49] The declarations of the struct and the mutex functions for the **counter** variable are skipped.

TODO: maybe fix this

:

```
mutexDestroy_2(&queue);
pthread_mutex_destroy(&counter.mutex);
pthread_mutex_destroy(&result.mutex);
```

The initializations of the tasks and the declarations of the futures is quite similar to those in section 3.1.3:

```
 1  Task mapperTask = taskInit_0(queuePointer, counterPointer);
 2  // mappers:
 3  VoidFuture[MAPPERCOUNT] mappers;
 4  for (int8 __i = 0; __i < MAPPERCOUNT; __i++) {
 5    mappers[__i] = runTaskAndGetVoidFuture(mapperTask);
 6  }
 7  free(mapperTask.args);
 8
 9  // reducer:
10  saveAndJoinVoidFuture(futureInit_0(resultPointer, queuePointer));
```

Since in the new solution the tasks do not return any results directly, the **VoidFuture** struct and respective functions are used.[50] The reduction of the original code **|reduce(RANGECOUNT, resultPointer, queuePointer)|.run.join** to the code in line 10 in the previous listing is done in the following manner. As was shown in section 3.2.2 the **run** call and the task declaration are reduced to a call of a function which combines the initialization of a task with the one of a future of a parallel running instance of this task. This call is reflected by **futureInit_0(resultPointer, queuePointer)**. Furthermore the join of the task must first bind the created future handle to some addressable location before it can use this handle by its address. For this reason **saveAndJoinVoidFuture** is used in place of **joinVoidFuture**. The declaration of **futureInit_0** follows:

```
VoidFuture futureInit_0(SharedOf_long_double_0* resultPointer, SharedOf_Queue_0* queuePointer) {
  Args_1* args_1 = malloc(sizeof(Args_1));
  args_1->resultPointer = resultPointer;
  args_1->queuePointer = queuePointer;
  pthread_t pth;
  pthread_create (&pth, null, :parFun_1, args_1) |];
  return (VoidFuture){ .pth = pth };
}
```

The original declaration of the reducer task contains references to the local variables **resultPointer** and **queuePointer**, which is why they are bound to equally named fields in the argument struct. An instance of the **VoidFuture** struct is returned as the parallel task does not return a value and, thus, has the type **task<void>**.

[50] Since the declarations of the **taskInit_0()** function and the **Args_X** structs resemble equivalent declarations of previously discussed code examples they are skipped here.

Inside the helper function **map()**, the synchronization statement of **counter** is replaced by its statement list, which is surrounded by calls to appropriate functions:

```
while (true) {
  uint32 start;
  uint32 end;

  startSyncFor1Mutex(&counter->mutex);
  {
    start = counter->value;
    if (start == threshold) {
      stopSyncFor1Mutex(&counter->mutex);
      break;
    }
    uint32 possibleEnd = start + BLOCKSIZE;
    end = (possibleEnd <= threshold)?(possibleEnd):(threshold);
    counter->value = end;
  }
  stopSyncFor1Mutex(&counter->mutex);

  queueSafeAdd(resultQueue, calcPiBlock(start, end));
}
```

The break statement is preceded by another call to the synchronization stop function, as the function would otherwise return in a state where **counter** would still be locked, which would ultimately cause the program to fail. The former expressions to get and set the value of **counter** are translated into accesses of the translated **value** field of the **counter** struct instance. The synchronization statement of **reduce()** is likewise translated into its statement list with surrounding synchronization calls. The translation of the expression **finalResult->set(item + finalResult->get)** contains two accesses to the aforementioned **value** field, one for **.set** and one for **.get**:

```
startSyncFor1Mutex(&result->mutex);
{
  for (uint32 i = 0; i < numberOfItems; ++i) {
    long double item;
    queueSafeGet(resultQueue, &item);
    result->value = item + result->value;
  }
}
stopSyncFor1Mutex(&result->mutex);
```

3.4 Safety Measures

So far the basic blocks that constitute parallel code execution and shared data synchronization, namely tasks, shared resources and synchronization thereof, were introduced. Still missing are most of the rules which ensure that only shared resources may be shared and that these can only be used in a sane way. The current section fills this gap by giving an informal overview of the rules that were implemented in ParallelMbeddr, categorized by their objectives. In the following paragraphs **t** denotes some arbitrary type.

3.4.1 Avoidance of Implicitly Shared Unprotected Data

Global variables can be accessed by any function for which they are visible. Therefore, they must have a type **shared<t>** in order to restrict any modifications of their values to synchronized contexts. This restriction can be too strong if a global variable is only accessed by exactly one thread. Nevertheless, in this paper, the conservative approach was chosen in order to establish a safe foundation. Future static code analysis should be leveraged to reliably detect the cases where restrictions can be loosened. Another class of data that is inherently vulnerable for unsafe data sharing arises from static variables. In C, local variables that are declared static have a "global lifetime" [47, p. 439], which means that as with global variables, the addresses of their allocated memory does not change. Thus, they keep their values from one function call to the next. The main difference between static local variables and global variables is the respective visibility. Consequently, static variables must have a type **shared<t>** as well. Finally, a base type **t** of a shared type may never be a pointer type with a base type other than a shared type. Otherwise the value of a shared resource would point to data that is not synchronized and would enable unprotected inter-task communication. For instance, in the following example the functions **foo()** and **bar()** do not block one another since they synchronize over different shared resources. Nevertheless they both write to the same location in memory, which causes a data race.

```
// global variables:
shared<int32*> v1;
shared<int32*> v2;

int32 main(int32 argc, string[] argv) {
  int32 sharedValue;
  sync(v1, v2) {
    v1.set(&sharedValue);
    v2.set(&sharedValue);
  }
  |foo()|.run;
  |bar()|.run;
}

foo() {
  sync(v1) { *v1.get = 0; }
}
bar() {
  sync(v2) { *v2.get = 1; }
}
```

3.4.2 Copying Pointers to Unshared Data into Tasks

The pass-by-value semantics of C generally already ensure that any local data which is refereed from within a task expression is safely copied into the task. On execution, the task will not access the original data, but a copy thereof. On the other hand, this approach becomes unsafe as soon as local variables are copied whose values are plain pointers (pointers to something else than shared resources). When such a copied pointer is used inside a task to access a pointed-to memory location in an unsynchronized manner, it accesses data that might simultaneously be accessed by another task, e.g. the task by which this task was created, which knows the address of the data. To avoid this behavior, every pointer that might be copied into a task by accessing a local variable or a function argument from within a task expression must point to a shared resource, i.e. must be of type **`shared<t>*`**. It has to be noted that this does not only hold for the variables themselves, but also for nested fields of struct instances and array elements. Furthermore, arrays must not be copied into tasks unless they are wrapped in a struct field. Due to the internal treatment of pointers in C (see section 2.2), the access to an array holding local variable would cause a copy of the address of the array into the task as a pointer. In consequence, references of local variables and arguments with type **`t[]...[]`** inside task expressions are not allowed. On the other hand, it is safe to have a struct with a an array field be copied into a task. In contrast to the former case the array would then be entirely copied along its surrounding struct instance.

3.4.3 Unsynchronized Access to Synchronizable Data

As was already mentioned in section 3.3 the value of a shared resource can only be accessed (retrieved or rewritten) from within a proper synchronization context. This approach ensures that no write to shared data invalidates any other write or read of the data. The according rule is that an expression **`e.get`** or **`e.set`** is only allowed if **`e`** is either a reference to a named resource in scope, i.e. a shared resource which is synchronized in a surrounding synchronization statement and bound to a new name; or if **`e`** is a reference to a variable with a shared resource as value which is also refereed to by a synchronization reference of a surrounding synchronization statement. By this restriction the following code would trigger an error message in ParallelMbeddr:

```
shared<shared<int32>> v;
sync(v) {
  sync(v.get) {
    // error: e.get seems to be unsynchronized
    e.get.set(0);
  }
}
```

Although the previous code would not produce any synchronization gap, ParallelMbeddr does not recognize this since the expression **`e.get`** of **`.set`** does not refer to a named resource or a variable with a synchronized shared resource. Instead, for exactly this purpose, named resources were implemented which allow to rewrite the code in the following valid manner:

```
shared<shared<int32>> v;
sync(v) {
  sync(&(v.get) as w) {
    w->set(0);
  }
}
```

The reasoning behind this was to simplify the implementation of the safety checking analysis. Again, the chosen approach can in certain cases be overly conservative. If write conflicts can never happen for a shared resource and, in consequence, data races thereof are impossible, it would be safe to access the variable outside any synchronization context despite the error message that is generated by the IDE. Moreover, by the applied lexical scoping ParallelMbeddr is not able to detect whether a shared resource is recursively synchronized across multiple function calls:

```
  shared<int32> v;
  sync(v) { foo(&v); }
  ...

void foo(shared<int32>* v) {
  sync(v) { v->set(0); }
}
```

Both problems should be addressed by static analysis in order to (partially) detect such cases. The second problem could further be addressed by the introduction of a *synced* type for shared resources that are already synchronized in the current scope.

Kommentar von Bastian
Maybe move to future work chapter.

3.4.4 Address Leakage of Shared Resource Values

In order to restrict any writes or reads of the values of shared resources to synchronization contexts, it is crucial to not leak the memory addresses of these values outside the protected synchronization context from where they could be accessed via the address operator (**&**). The measures to keep the addresses encapsulated constrain the use of the address operator and the use of arrays: The first rule forbids any expressions **&e** where **e** contains a sub path **eSub.get** and **e** does not evaluate to a shared resource. The latter condition allows the programmer to get the address of an encapsulated shared resource, which is unproblematic since shared resources may not be overwritten as is explained in the next section 3.4.5. The second rule states that an expression **e** of some array type is forbidden, if **e** contains a sub path **eSub.get** and the parent of **e** does not access a specific element of **e**. Thus, any access to an (multidimensional) array that is encapsulated in a shared resource must actually be extended to an access of the innermost element of the array. Otherwise it would be possible to assign the array itself or, in the case of a multidimensional array, take an element of the array which itself is an array and assign it to an unprotected pointer variable. Hence, the address of the array or of an element of this array would be leaked. For instance in the following example, ParallelMbeddr would complain about an address leakage of the first element of the array wrapped inside **v**:

```
shared<int32[5][10]> v;
int32* pointer;
// address leakage!
sync(v) { pointer = v.get[0]; }
```

3.4.5 Overwriting Shared Resources

Shared resources may never be overwritten. The reason for this regulation results from the following consideration. If a shared resource **r** shall be overwritten, e.g. by a direct assignment or by a **set** if **r** is nested inside another shared resource, it must be synchronized first since the overwriting could overlap with another access to **r** from within another task. Before the resource is rewritten, the mutex of **r** must be destroyed in order to prevent memory leaks. Furthermore, after the rewriting is done, the newly created mutex for **r** must be initialized prior to any usage. Hence, in the time between the destruction and the re-initialization, the mutex of **r** (respectively, **r'** after the rewrite is done, because the variable will refer to a new value) cannot be accessed in the synchronization attempt of any other simultaneously executed synchronization statement. Any such task would thus have to be blocked, which would complicate the compiler and decrease the performance of the executed code without offering any worthy advantage. In addition to this problem, the overwriting of a shared resource of a struct instance that itself contains a shared resource field **f** would invalidate any pointer **p** to **f**. **p** could therefore not be used anymore afterwards, which on the hand complies with the usual C semantics, but does not fit the safety-first approach of ParallelMbeddr. Nevertheless, it is safe to copy a shared resource into the memory of a local variable declaration or of a function argument, i.e. a pass of a shared resource to a function or an initialization of a newly created local variable with a shared resource is valid. In these cases, the shared resources can only be used after their initializations with shared resource copies. The safety enforcing rules are as follows: A variable that refers to a shared resource or to a value that contains a nested shared resource may not be assigned a new value (the initialization of a declaration is not a classical assignment). Additionally an expression **e.set(e')** is not allowed if **e'** is a shared resource or contains a shared resource, because **e.set(e')** is translated into an assignment (see section 3.3.2).

3.4.6 Overwriting Pointers to Shared Resources

Not only references to variables of shared resources can be used for the definition of synchronization references, but also expressions that evaluate to pointers of shared resources. While providing more flexibility this approach facilitates the introduction of data races via the following two techniques. For the first problem the following example is given:

```
struct Container {
  shared<int32> value;
  shared<int32>* pointer;
}
// somewhere in the code
```

```
shared<Container> c;
// somewhere else
sync(c, &c.get.value as valueP, c.get.pointer as pointerP) {
  // make use of valueP and pointerP
}
```

Generally, the semantics of the synchronization statement would be that **c**, **valueP** and **pointerP** would be synchronized altogether by repeatedly trying to lock their mutexes. The translation of the named resources would introduce local variables which bind the values of the expressions **&c.get.value** and **c.get.pointer**:

```
shared<int32>* valueP = &c.get.value;
shared<int32>* pointerP = c.get.pointer;
sync(c, valueP, pointerP) {...}
```

The intermediary translated code reveals the problem: The initialization values of the pointers to the shared resources are, due to eager evaluation, evaluated as soon as they are declared. In the case of **valueP** this does not matter, since the **value** field may never be overwritten. **pointerP** on the other hand is a copy of the field **pointer** which may be overwritten by any task that has access to **c**. This means that between the declaration of **pointer** and its use in the synchronization statement its value may become out of date. Hence, the atomicity semantics of the synchronization of **sync** would be violated. The translation could be fixed to safely support such cases. Yet, due to the possibility to make use of control-flow-breaking statements inside synchronization statements, the translated code would become considerably more complicated. For this reason the opposite path was chosen: Cases like the depicted one are forbidden by the following rule. If inside the synchronization list **l** of a synchronization statement the expression of a well-typed named resource **n** – which must have the type **shared<t>*** – is no address reference expression **&e** but contains either a reference to a previous named resource in **l** or a reference to a variable that is synchronized by a previous synchronization resource in **l**, then **n** is invalid.

The second problem is illustrated by the following example:

```
shared<int32> value1;
shared<int32> value2;
shared<int32>* p = &value1;
shared<int32>* q = &value2;
// somewhere else
sync(p) {
  p = q;
  p.set(42);
}
```

Since both **p** and **q** are pointers to shared resources they can be used for synchronization resources. Furthermore, according to the previous rules of this chapter, they may be arbitrarily assigned new values. Thus, the line **p = q** should be fine. Nevertheless this assignment introduces the potential for data races, since **q**'s target **value2** and thus its modification via **p.set(42)** is not synchronized. In addition, at the end of the synchronization statement the program will try to release the lock of **`value2`**'s mutex although only **`value1`**'s mutex was previously locked. To solve this problem, pointers to shared resources should never be overwritten if they are simultaneously used for synchronization purposes. For this purpose, a synchronization resource whose type is **`shared<t>*`** must be named. Additionally, a named resources may not be treated like an *lvalue* which means that neither it may be assigned any value nor its address may be retrieved (to prevent overwritings via address dereferences).

4 Optimization

The previous chapter introduced the means to enable parallel execution of code via tasks and establish communication between tasks via shared resources. In order to make the communication thread-safe, synchronization statements were presented which provide synchronization contexts of atomic thread-safe blocks for the shared resources that they synchronize. For the purpose of both simplicity of the design and thread-safety of the user-code, conservative restrictions were made: in the variability of the code and the scopes of synchronization contexts. While this strategy simplifies the construction of correct programs for the user, it may induce unnecessary serialization during program execution if potential data hazards will actually never manifest at runtime [54]. While the synchronization overhead reflects the optimization potential in the time-wise dimension, there is also a space-dimension which originates from the way mutexes are used in the implementation.

4.1 Space Optimization

As was briefly addressed in 3.3.2, the runtime memory consumption of the translated code is extended by the amount of memory that is occupied by the mutex maintenance. In addition to the obvious memory consumption of the handle and some internal management data, the realization of the choice to make mutexes robust against recursive locks requires a counter field to be maintained throughout the lifetime of each mutex. This property impairs both the space and computation time overhead of the implementation unnecessarily if shared resources are never recursively synchronized. If such cases are detected (which they are, as will be shown in the next section), the generator could decide to declare the according mutexes as non-recursive. This optimization is left for future extensions of ParallelMbeddr. Another starting point for the optimization of space consumption would be the reduction of padding, i.e. unused data, that is automatically introduced by the compiler into the struct instances of shared resources. Padding is added into structs in order to retrieve data from memory more efficiently by aligning it along proper addresses [26, p. 27]. The amount of padding that is inserted depends on the difference among the byte sizes of the individual fields, as well as the order of these fields [26]. Since the “art of C structure packing”[51] is no trivial task and space optimization is no primary concern of this paper, according work is left to future research.

4.2 Time Optimization

Various optimizations for lock-based synchronizations have been conceived. The general goal of all approaches is to minimize the overhead of synchronization measures. Among others, this can be accomplished in two different ways: First, the amount of synchronization management, i.e. the time spent for

[51] See http://www.catb.org/esr/structure-packing/ for details.

acquiring and releasing locks, can be reduced by diminishing the number of performed locks. This technique is called *lock elision*. Secondly, the waiting time of threads for the release of locks which they try to acquire, also known as *lock contention*, can be diminished. While the classic approaches try to optimize the code at compile-time, increasing effort is performed towards optimizations which occur at run-time. The latter kind primarily resides in the domain of the transactional memory model [54][56]. Due to the complexity and overhead of such techniques this thesis focuses on compile-time optimizations.
The applicable measures for optimizations differ in their coverage of 'perceived potential' and their performance. Superior techniques often require more comprehensive information, which, in turn, demands more sophisticated analyses. Such analyses, however, usually come with an increase of complexity. Typical candidates are *pointer analysis* and *data-flow analysis*.

4.2.1 Pointer Analysis

Pointer analysis, also known as *points-to analysis*, tries to find the set of memory locations to which a pointer may point[57]. The information delivered by the pointer-analysis will be heavily used in the static analysis of this chapter. The following example illustrates the basic principle of pointer analyses. At the end of the if-else statement **p** may point to the location of either **v** or of **w**. Due to the copy-semantics of assignments in C, **q** will have the same value as **p** after the assignment in line 5. **p** can therefore be regarded as an *alias* of **q**.

```
1  int32 v, w;
2  int32* p;
3  if (condition)  p = &v;
4  else            p = &w;
5  int32* q = p;
```

The quality of a pointer analysis is reflected by its precision. Two properties – that are relevant for this work – influence the precision: whether the analysis is *flow-sensitive* or *-insensitive* and whether it is *inter-procedural* or *intra-procedural* [11].[52] The first property distinguishes whether the particular flow of data and with it the order of statements is taken into account of the analysis. For an illustration, the previous code listing is reconsidered. In a flow-insensitive analysis, the set of locations for **p** would contain the locations of both **v** and **w** in either branch. A flow-sensitive analysis, on the other hand, would precisely assign **v** or **w** to the points-to set (in the following also called *alias set*) of **p**. The second property distinguishes how precisely the calculation of points-to sets regards effects across functions, i.e. the context flow across function calls. The following example shall illustrate the difference between the two shapes. According to Andersen, an intra-procedural analysis would merge the calls of **bar()** so that the points-to set of **p** would contain both **v** and **w**. Likewise, the sets of **vP** and **wP** would contain the same elements. On the other hand, an inter-procedural analysis would distinguish the calls so that, in the end, the sets of **vP** and **wP** would contain exactly **v**, respectively **w**.

```
void foo() {
  int32 v, w;
```

[52] The latter property is also called context-sensitivity [68].

```
  int32* vP = bar(&v);
  int32* wP = bar(&w);
}
void bar(int32* p) {
  return p;
}
```

At times it is necessary to not compute the aliases that a pointer can have in the course of a program run. Instead the set of locations that a pointer will point to on every possible path through the program might be needed.[53] While the former is called a *may point-to* analysis, the latter is called a *must point-to* analysis [11]. Currently, mbeddr does not provide any form of pointer analysis.

4.2.2 Optimization Opportunities

Static analysis offers a variety of optimization opportunities. These differ both in their optimization potential and the information needed for their realizations. Due to the similarity of shared resources to the synchronization concept of Java's monitors and the ongoing research in this area, the optimization ideas were (mainly) influenced by the according literature [7][29][15]. The opportunities are arranged in the oder in which they are applied in ParallelMbeddr. The first three of the following paragraphs concern themselves with the *elision* of unnecessary locks and, hence, the computational overhead of synchronization management. The fourth focuses on the reduction of lock contention.

Single-Task Locks

The most obvious case in which locks for shared resources can be removed is when synchronized data is only accessed by one task. This may happen for a limited amount of time, for instance in the time span from the declaration of a local variable of a shared resource to its first sharing with other tasks:

```
void foo() {
  shared<int32> v;
  shared<int32>* vP = &v;
  init(vP);
  |task(vP)|;
}

void init(shared<int32>* var) {
  sync(var as varToSet) {
    varToSet->set(0);
  }
}

void task(shared<int32>* var) {
```

[53] An application of this demand will be shown later on.

```
  sync(var as varToGet) {
    int32 val = varToGet->get;
  }
}
```

Whereas **varToSet** in **init()** does not need to be synchronized, since it is not yet shared with another task, **varToGet** obviously needs to be synchronized inside **task()**.
Furthermore, during the whole run of a program, a shared resource could be accessed by one task only. This can happen if the programmer does not work attentively. More importantly, the re-use of existing data structures and according functions for single-task data can cause the same effect. For instance, a thread-safe queue and functions to manage this queue could be re-used by the user for data that resides in only one task. It would then be helpful to distinguish the necessity of locks (i.e. sync resources) depending on the use of queue.

Read-only Locks

In ParallelMbeddr shared resources may actually never be set. For primitive data, e.g. of the type **shared<int32>** this should seldom be the case, if the user creates the code carefully. However, he might decide to use structs in order to pack independently synchronizable data:

```
 1  struct QueueContainer {
 2    shared<Queue>   queue;     // Queue is given as a black-box
 3    shared<boolean> isFull;
 4    shared<int32>   itemCount;
 5  }
 6
 7  void foo() {
 8    shared<QueueContainer> queueC;
 9    shared<QueueContainer>* queueCP = &queueC;
10    // ...
11    sync(queueC) {
12      sync(&queueC.get.isFull as isFull) {
13        isFull->set(true);
14      }
15    }
16  }
```

Because of the semantics for variables of shared resources, **QueueContainer** can never be overwritten by another value. This is accomplished by mbeddr's non-typesystem rules. Therefore, any synchronization of **queueC** and any of its aliases is not necessary. However, the user still needs to pack the data into a shared resource in order to be able to safely share it with other tasks. Although the IDE could infer that **queueC** never needs to be synchronized, the user has to annotate the synchronization in line 11. If, on the other hand, such synchronization would not be required for **queueC**, additional synchronizations would have to be added if **QueueContainer** is eventually equipped with non-shared data. The current approach omits this change.[54] Nevertheless, the compiler should take care of eliminating locks for such data. If functions are called with both read-only shared resources and written shared resources as argument data, the necessity of synchronizing them might depend on the respective calls (equivalent to single-task locks). This could for instance be the case for logging functions which only read the data of their arguments that shall be logged.

Recursive Locks

ParallelMbeddr does not prevent the user from acquiring locks for shared resources recursively. Instead, due to the scoping rules of synchronization resources and their contexts the programmer might be forced to synchronize shared resources recursively:

```
void sort(shared<int32[1000]>* array, int32 start, int32 end)
int32 middle = ...
sort(array, start, middle);
sort(array, middle + 1, end);
sync(array) { /* merge the sorted sub arrays */ }
```

The example depicts a simplified version of a sort algorithm like merge sort. The function **sort()** recursively divides the array into smaller sub arrays and uses the sorted sub arrays to calculate a sorted version of the current array interval $[start, end]$. In the first call of **sort()** from outside, **array** might not yet be synchronized. In any other recursive call, however, **array** will definitely be already synchronized. Thus, at least in every sub-call of **sort()** the corresponding lock should be omitted.

Lock Contention

Besides the removal of locks, an important optimization opportunity is the reduction of lock contention. In order to accomplish this goal, the synchronization lists of synchronization statements should be shrunk to the absolute minimum. In the following, this technique is called *lock narrowing*. For instance, the user might decide to apply coarse-grained synchronization by defining one big synchronization context inside a function:

```
void calculate(shared<double>* result) {
  sync(result as myResult) {
    // do something that is expensive and unrelated to the argument
    double pi = calculatePi();
    // now use myArg
    myResult->set(pi);
    // again, something unrelated
    log(pi);
  }
}

```

[54] Of course, it is discussable which approach would be better.

```
12 double calculatePi() {...}
13 void log(double arg) {...}
```

The statements in 4 and 8 do not make any use of the synchronized argument **result**. Hence, it would be safe to move these statements out of the synchronization context:

```
void calculate(shared<double>* result) {
  double pi = calculatePi();
  sync(result as myResult) { myResult->set(pi); }
  log(pi);
}
```

One could argue that synchronization lists could even be split into multiple parts in order to separate statements whose evaluations access the current synchronization resources from those that do not:

```
void increment(shared<int32>* c) {
  int32 current, next;
  sync(c as myC) {
    current = myC->get;
    next = current + 1;
    myC->set(next);
  }
}
```

$$\underset{\text{split}}{\longrightarrow}$$

```
void increment(shared<int32>* c) {
  int32 current, next;
  sync(c as myC) { current = myC->get; }

  next = current + 1;

  sync(c as myC) { myC->set(next); }
}
```

Yet with such an aggressive strategy, the optimizer might split the code across data dependencies which were formerly reflected in the code by the scope of a synchronization statement. For instance, in the previous example the split does not take into consideration the data dependencies between **current**, **next** and **myC**. Therefore, when another call of **increment()** is executed in an interleaved fashion, the resulting code introduces data races for the shared resource that **c** points to.

4.2.3 Performed Optimizations

The optimizations that were performed in this work are direct realizations of the aforementioned optimization opportunities. For the lack of supporting analyses in mbeddr at the time that this thesis was written, the optimizations assume that all threads are executed simultaneously in order to fit the scope of this work. Thus, since mbeddr was missing a pointer-analysis, at first a simplified pointer analysis for the application in the optimization algorithms was conceived. In this analysis, differing from the usual approach, a separate alias set is computed for each local variable, argument and reference of both kinds.[55] Additionally, divergent from the usual terminology, an alias in this context is a variable or argument of type **shared<t>**. These differences result from the following facts

- only shared resources are considered;
- shared resources are bound to memory locations, as they may not be rewritten;
- shared resources inside structured data like arrays and structs are not considered.

[55] The analysis is therefore *inclusion-based*, i.e. alias sets may overlap [68].

Also, it is assumed that everything which has a type **`shared<t>`** or **`shared<t>*`** (i.e. variables, arguments and expressions) has an alias set. For the former type, this is clearly the location of the resource itself.

Pointer analysis

As was mentioned before, the analysis that was implemented in the course of this thesis makes several simplifications in order to fit the scope of this work. The analysis is intra-procedural but flow-sensitive for the regarded language concepts. It can either compute *must point-to* or *may point-to* alias sets. As a starting point, a simplified directed data-flow graph is constructed. The nodes of this graph consist of local variable declarations, arguments[56] and references to either kind. For each reference r to a local variable or an argument x an arc (x, r) is added to the graph. Furthermore, for each local variable v of type **`shared<u>`** or **`shared<u>*`**, whose initialization expression is a reference r' to a local variable or argument of the same type, an arc (r', v) is added. The same is done for local variables of type **`shared<u>*`** whose initialization expressions reference local variables or arguments of type **`shared<u>`** by address. Equivalently to local variables and initialization expressions, arcs (r, a) are added to the graph for according pairs of argument values r and arguments a. **TODO:** Visualize
The data-flow graph is used to perform a pointer analysis. The output of the analysis is a directed alias graph whose nodes comprise the nodes of the data-flow graph. Each arc (u, v) of the alias graph connects a variable, argument or reference u with a variable or argument v. In doing so, either v is an alias for u or u refers to a variable $v2$ for which v is an alias.[57] Trivially, loops (u, u) for a variable or argument u are contained, since every node is an alias for itself. With this initial setting the algorithm works as follows:

```
function ALIASES($g$, $strict$)                                ▷ g is an inverse data-flow graph
    for all $v \leftarrow$ VARIABLES AND ARGUMENTS($g$) of type shared<t> do
        ADD($a$, $(v, v)$)                                           ▷ a is the new alias graph
    end for
    repeat
        find some $(n, m) \leftarrow$ ARCS($g$) where $a[n]$ does not contain all $a[m]$
      ▷ in strict mode, only must-aliases are considered ⇒ all in-nodes must then have the same aliases
        if strict, and $n$ is an argument and there are $i, j \in g[n]$ where $a[i] \neq a[j]$ then
            skip $(n, m)$
        else
            $a[n]$.ADD ALL($a[m]$)
            if $n$ is a local variable then
                for all $r \leftarrow$ FOLLOWING REFERENCES($n$) whose targets are in $a[n]$ do
                    $a[r]$.ADD ALL($a[n]$)
                end for
```

56 Although generally, arguments are the values that are passed to function parameters, in the course of this work, no such distinction is made. Instead, the formal function parameters are called arguments and the values, which are passed to functions, are called argument values. This terminology is closer to the one established in mbeddr.

57 Thus, due to the inclusion of references, the alias graph is not one in the classical sense.

```
            end if
        end if
    until no more changes possible
end function
```

aliases propagates alias information through the alias graph as long as changes are possible. It repeatedly tries to find a node n which does not contain all aliases of a node m for which an arc (m,n) resides in the original data-flow graph. In such a case, n gets connected to the missing aliases. If *must point-to* aliases need to be calculated (which is currently necessary for the recursive-lock elision algorithm), all nodes l for which an arc (l,n) exists must have equal aliases in order to be applicable for an alias augmentation of n. Due to the simplifications of the analysis, this can only be the case for value-to-argument arcs, since no other branches are considered.

Removal of Single-Task Locks

In order to remove synchronization resources of variables which are only used in one task, the aforementioned alias analysis is performed. It is fed with the inverse of a data-flow graph, exactly as it is delivered by the aforementioned function.

```
function REMOVE SINGLES(g, a, d)        ▷ g = inverse data-flow graph, a = aliases, d = additional data
    for all $v \leftarrow d.variables$ do
        if there is no $(n,m)$ in $g$ where $a[n]$ contains $v$ and $n$ and $m$ reside in different tasks then
            $\bar{c}$.ADD($v$)                                  ▷ $\bar{c}$ = single (clean) task variables
        end if
    end for
    REMOVE CLEAN LOCKS(c, a, d)
end function
```

Remove Singles gathers all variables which never leave their defining task contexts. This is accomplished by investigating for each variable v all references whose alias set contains v. If any of these references leave the task context of the variable or argument x that they reference – which may happen if they reside in a task expression but x does not –, then v is no single task variable. The sought variables are thus gathered. The actual lock removal is accomplished by **`Remove Clean Locks`**, which is also used for the removal of read-only locks. The following pseudo-code depicts the general approach of the function:

```
function REMOVE CLEAN LOCKS($\bar{c}$, $a$, $d$)        ▷ $\bar{c}$ = clean variables, $a$ = aliases, $d$ = additional data
    for all $s \leftarrow$ SYNC RESOURCES(data) do
        if $\bar{c}$ contains all $a[s.expr]$ then        ▷ $s.expr$ evaluates to a shared resources or pointer thereof
            $s$.REMOVE
        else
            $f\_to\_\bar{s}[function\ of s]$.ADD($s$)    ▷ $f\_to\_\bar{s}$ contains functions with partially clean sync resources
        end if
    end for
    for all $(f, \bar{s} \leftarrow f\_to\_\bar{s}$ do
        TRY TO DUPLICATE($f, \bar{s}, \bar{c}, a, d$)
    end for
end function
```

Every synchronization resource s is either directly removed or deferred. If all aliases of the expression of s are clean (e.g. they are all single task variables), s can clearly be removed, since its synchronization is useless. Otherwise at least some of its aliases might be clean. In case of single-task locks, these aliases would ideally originate from an argument like those in the following example:

```
void shareXButNotY() {
  shared<int32> x;
  shared<int32>* xP = &x;
  shared<int32> y;
  |xP|;                    // important: for |x| or |&x|, x would not be shared but copied
  syncXOrY(&x);
  syncXOrY(&y);
}

void syncXOrY(shared<int32>* xOrY) {
  sync(xOrY as val) { val.set(0); }
}
```

In this example, **x** is shared, but **y** is not. Therefore, the set of clean variables c in **Remove Clean Locks** would only contain **y**. On the other hand, the set of aliases $a[$**xOrY**$]$ would contain both variables, as they would be forwarded to **xOrY** via the calls **syncXOrY(&x)** and **syncXOrY(&y)** in the aliasing analysis.[58] In this case the function *Try To Duplicate* would distinguish the respective calls and learn that for **syncXOrY(&y)** no synchronization is needed for **xOrY**. In such a case, the function could be inlined for the safe call and the clean synchronization resource could be removed. Alternatively, as is done by *Try To Duplicate*, the function can be duplicated and accordingly optimized:[59]

```
  //...
  syncXOrY(&x);
  syncXOrY_1(&y);
}
void syncXOrY(shared<int32>* xOrY) {
  sync(xOrY as val) { val.set(0); }
}
void syncXOrY_1(shared<int32>* xOrY) {
  shared<int32>* val = xOrY;
  sync() { val.set(0); }           // the empty sync will be removed
}
```

[58] This merge is caused by the inter-procedural property of the current alias analysis. In case of an intra-procedural pointer analysis, the following analysis would be facilitated. Of course, this simplification, in turn, requires a more complex algorithm for the conduction of the performance analysis.

[59] Function duplication instead of inlining is done since this approach also works for recursive functions.

However, if the aliases of the synchronization resource's expression e are not received via paths to the arguments of the surrounding function, function inlining (or duplication) will not help. This may for instance be the case if e refers to a local variable of type **`shared<t>`** that resides in the same function. Another possibility is that it refers to a global variable whose value was set inside another function. In order to match the simplified pointer analysis, currently only synchronization resources, whose expressions directly refer to one of the arguments of the surrounding function (like in the previous example), are considered. The *Try To Duplicate* function works as follows:

```
    ▷ $f$ = function, $\overline{ds}$ = partially clean sync resources, $\overline{c}$ = clean variables, $a$ = aliases, $d$ = add. data
function TRY TO DUPLICATE($f$, $\overline{ds}$, $\overline{c}$, $a$, $d$)
    for all $ds \leftarrow \overline{ds}$ do                ▷ for each call find the sync resources that are clean
        $\overline{da} = a[ds.expression]$ which is not in $\overline{c}$        ▷ $\overline{da}$ = dirty aliases for $ds$
        for all $l \leftarrow$ CLEAN CALLS FOR($ds$, $\overline{da}$, $f$, $a$) do        ▷ $l$ = clean calls for $ds$
            $l\_to\_\overline{cs}[l]$.ADD($ds$)        ▷ $l\_to\_\overline{cs}$ = clean syncs for call $l$
        end for
    end for
    for all $l, \overline{cs} \leftarrow l\_to\_\overline{cs}$ do        ▷ pack the calls by equal sets of clean sync resources
        $\overline{cs}\_to\_\overline{l}[\overline{cs}]$.ADD($l$)
    end for
    for all $\overline{cs}, \overline{l} \leftarrow \overline{cs}\_to\_\overline{l}$ do    ▷ duplicate the function for calls of equal sets of clean sync resources
        DUPLICATE FUNCTION($f$, $\overline{cs}$, $\overline{l}$)
    end for
end function
```

Try To Duplicate first considers all partially clean synchronization resources $\overline{ds}$, i.e. synchronization resources whose expression aliases are clean for at least one function call. For each ds of these $\overline{ds}$ it uses the function *Clean Calls For* to determine the function calls $\overline{l}$, which do not need ds. This means that the ds actually needs to get its aliases from one of the arguments of its function (otherwise function inlining would be useless). Furthermore, every call in $\overline{l}$ must not contain any of the dirty aliases of ds. Hence, they can only originate from some other call. In case of an inter-procedural pointer analysis this information would certainly be easier to gather. For each synchronization resource ds with a non-empty set $\overline{l}$ for each call, a mapping to ds is established. These mappings are then used to cluster calls which have equal sets of clean synchronization resources. These clusters, in turn, are used by *Duplicate Function* to generate optimized versions of the current functions. For the lack of valuable insight, the definitions of *Clean Calls For* and *Duplicate Function* are skipped.

Removal of Read-only Locks

Read-only locks are removed equivalently to single-task locks. It differs in the condition that it uses to determine whether locks for a specific variable may be removed:

```
function REMOVE READONLYS(g, a, d)    ▷ g = inverse data-flow graph, a = aliases, d = additional data
    for all $v \leftarrow d.variables$ do
```

```
        **if** $\exists$ *e.set*(_) in *d.sharedSets* where $a[e]$ contains $v$ **then**
            skip $v$
        **else if** $\exists$ *e.get* in *d.sharedGets* where $a[e]$ contains $v$ and $\exists\ e' = \_$ where $e'$ contains $e$ **then**
            skip $v$
        **end if**
        $\overline{c}$.ADD($v$)                         ▷ $\overline{c}$ = readonly (clean) task variables
    **end for**
    REMOVE CLEAN LOCKS(c, a, d)
**end function**
```

A few things should be noted for the last two optimizations. First, in the actual implementation, the gathering of 'clean variables' is separated from the actual call of *Remove Clean Locks*. Instead, variants of the two algorithms, called *Get Singles* and *Get Readonlys*, are used to first gather all variables for which locks can be removed. Only then *Remove Clean Locks* is applied to the union of both variable sets. This way, a function may be duplicated only once if it is called multiple times with (a) shared resources that are actually shared and written, (b) read-only shared resources and (c) single-task shared resources. The second thing to consider is that these removal functions are actually called as often as some optimization by their application is possible. Otherwise, optimizations might not be possible if an argument value is itself 'partially clean', like **var** in the following example:

```
void send() {
  shared<int32> a;
  shared<int32> b;          // b is never shared or written
  shared<int32>* aP = &a;
  |aP|;
  sync(a) { a.set(0); }

  forward(&a);
  forward(&b);
}

void forward(shared<int32>* var) {
  sync(var as myVar) { myVar.get; }
  receive(var);
}

void receive(shared<int32>* var) {
  sync(var as myVar) { myVar.get; }
}
```

Since **b** is never shared or written, its synchronizations inside **forward()** and **receive()** are useless. **a**, on the other hand, must always be synchronized. In the first optimization round, the values of the calls of **forward** can strictly be separated into the 'clean' value **b** and the 'dirty' value **a**. Therefore, an optimized variant for **forward(&b)** can be generated. On the other hand, there is only one call for **receive()**

and the aliases of its value are currently merged, because the pointer-analysis is intra-procedural. Thus, another round of optimizations has to be applied where the two calls of the now separated functions **forward()** (for **a**) and **forward_0()** (for **b**) can be distinguished. An intra-procedural pointer-analysis could directly deliver this information. The third point to consider is that function duplication does not work across functions which do not split the aliases of their arguments themselves, as was done by **forward()**. If the synchronization statement was removed in this function, the algorithm would not detect that a duplication of **forward()** might help. In consequence, **receive()** would not be duplicated as well. For this reason, in the future, cross-function duplications should also be made available. On the other hand, like function inlining, duplication can lead to a 'code explosion' if it is used too aggressively. Thus, it should be controllable by the programmer.

Removal of Recursive Locks

In contrast to the previous optimization techniques, in this section, the property, which must be proven in order to be able to remove a lock, does not hold for entire variables. Instead, the recursiveness of a lock has to be shown separately for every synchronization resource. For the basic idea of a recursive-lock removal algorithm, an arbitrary synchronization resource *r* of an expression *e* is considered. If it can be shown that for all aliases of the pointer or shared resource, which *e* evaluates to, *e* must already be synchronized, then *r* can be removed. In order to facilitate the analysis, the data-flow graph, which is necessarily used in such an algorithm, should be preprocessed in the following way: First, edges caused by recursive function calls should be removed. The rationale behind this constraint is that recursive functions should not be able to synchronize their arguments on their own. In the following code listing, an analysis of **recursive()** could otherwise detect that **value1** and **value2** are already synchronized at the beginning of the function.

```
void recursive(shared<int32>* value1, shared<int32>* value2) {
  sync(value2 as myValue2) {
    myValue2.set(2);
  }
  sync(value1 as myValue1) {
    myValue1.set(5);
    recursive(myValue1, myValue1);
  }
}
```

Additionally, if there is another call to **recursive()** from outside the cyclic call structure the algorithm need not first check whether any of the calls to **recursive()** originate from a recursive call path in order to treat it differently. Furthermore, while in the previous example recursions would be easy to detect, indirect recursions across multiple functions are more complicated to handle. For these reasons, the directed data-flow graph should be created by starting in the main function and avoiding edges that close cycles in the corresponding call graph. Secondly, edges between nodes of different task contexts in the data-flow graph should be ignored. As a result, false recursive locks across tasks are omitted:

```
void task1(shared<int32>* value) {
  sync(value as myValue) { value.set(1);
    |task2(value)|.run                    // here, value is not synced anymore
  }
}

void task2(shared<int32>* value) {
  sync(value as myValue) { value.set(2); }
}
```

Since every task needs to synchronize its shared resources on its own, the synchronization context of **sync** in **task1** cannot be forwarded to **task2**. Additionally, it should be noted that synchronization contexts may not be forwarded to other tasks via the wrapped values of shared resources. For instance, in the following example, the reference to **myValue** is a reference to a synchronized value. Yet, this synchronization information may not be forwarded into **myContainer**. Otherwise a different task which accesses the value of **myContainer** would not have to synchronize the value any more.

```
void task1(shared<int32>* var, shared<shared<int32>*>* container) {
  sync(var as myVar, container as myContainer) {
    myContainer->set(myVar);
  }
}
```

The construction of the alias graph should, however, only omit cyclic edges, since aliases across tasks are completely valid. In the following, two algorithms for the recursive-lock removal are presented.[60] The first algorithm uses the alias graph to locally mark variable references inside synchronization statements whose aliases are all synchronized. It uses the data-flow graph to push the synchronization states forward across function contexts:

```
▷ $g$ = data-flow graph (dfg) w/o recursions or task-crossings, $i$ = inv. dfg, $a$ = aliases, $d$ = add. data
function REMOVE RECURSIVE LOCKS 1($g, i, a, d$)
    for all $s \leftarrow d.syncResources$ do                         ▷ spread the sync contexts locally
        for all $r \leftarrow$ REFERENCES IN($s$) do
            if $r \neq s.expr$ and TASK($r$) = TASK($s$) and $a[s.expr] \supseteq a[r] \neq \emptyset$ then
                $\bar{s}$.ADD($r$)                ▷ $\bar{s}$ = references to synchronized shared resources
            end if
        end for
    end for
    repeat                                                         ▷ spread the sync contextes globally
        for all $s \leftarrow \bar{s}$ do
            if there is some $n$ in $d[s]$ which is not in $\bar{s}$ and $\bar{s}$.CONTAINSALL($i[n]$) then    ▷ $n$ = next node
                $\bar{s}$.ADD($n$)
```

[60] The former algorithm was implemented for its simplicity in light of the simplified scenarios. The second one, on the other hand, avoids the incremental flow of synchronization contexts through the data flow graph and might be better suited for more complicated scenarios with concepts like assignments, shared resources nested in structures and arrays and multidimensional pointers (e.g. **`shared<int32>**`**).

```
            end if
        end for
    until no more change is possible
    for all $s \leftarrow d.syncResources$ do                    ▷ remove recursive locks
        if $\bar{s}$.CONTAINS($s.expr$) then
            remove $s$
        end if
    end for
end function
```

After the synchronization contexts have been flowed through the data-flow graph, the function can determine whether the expression of any synchronization resource s evaluates to an already synchronized shared resource. In such a case s can safely be removed. It should be noted that code resulting from the implementation of the algorithm is currently not able to remove recursive locks across functions reliably. This limitation results from the fact that the applied primitive pointer analysis works intra-procedural. In consequence, the aliases for a function argument would usually be merged for different calls of a function. For an illustration, the following example is given:

```
void foo() {
  shared<int32> p1;
  shared<int32>* p1P = &p1;
  shared<int32> p2;
  shared<int32>* p2P = &p2;
  |p1P|;
  |p2P|;
  bar(p1P, p1P);
  bar(p2P, p1P);
}

void bar(shared<int32>* v1, shared<int32>* v2) {
  sync(v2 as myV2, v1 as myV1) {
    myV1->set(1);
    myV2->set(2);
  }
}
```

Due to the merge, the alias set of **`myV1`** becomes {**`p1`**, **`p2`**} and dominates the alias set {**`p1`**} of **`myV2`**. Hence, in the recursive lock algorithm the synchronization resource **`myV2`** would removed. The optimized code would therefore become:

```
...
void bar(shared<int32>* v1, shared<int32>* v2) {
  shared<int32>* myV2 = v2;
  shared<int32>* myV1 = v1;
  sync(myV1) {
    myV1->set(1);
    myV2->set(2);
  }
}
```

Thus, for both calls **bar(p1P, p1P)** and **bar(p2P, p1P)** only **myV1** would be synchronized. In consequence, in the second call only the shared resource of **p2** would be synchronized, although both **p2** and **p1** would be modified via **myV1->set(1)**, respectively **myV2->set(2)**. This problem is currently faced by applying a *must point-to* pointer-analysis for this optimization technique. As a result, the alias sets in the example would become empty. Hence, for the algorithm to be able to decide that the alias set of a synchronization resource s is superfluous, the condition $a[s.expr] \supseteq a[r] \neq \emptyset$ in the algorithm requires that the alias set of s is not empty. For testing purposes, this restriction can be deactivated (which happens in the evaluation chapter). In the future, an inter-procedural pointer analysis should be applied in order to check whether for every function call a specific synchronization resource s is covered by another synchronization resource t (t's alias set is a superset of s' alias set). If that is the case, it may be removed. The non-emptyness condition can then be omitted. As in the previous algorithms, it would further make sense to apply function inlining/duplication to remove synchronization resources for arguments – of a function f–, which are used recursively only for a strict subset of calls of f.

Another problem results from the restricted range of expressions and types that are currently supported in the optimization algorithm. If for instance a shared resource is nested inside an array **a** of type **shared<int32>[100]** and accessed via an array-access expression **&a[5]**, the optimizer is not able to identify the accessed shared resource. Thus, if this expression is used as an initialization for a local variable **v** of type **shared<int32>***, the alias-set of **v** will become empty. The same property holds for structs. For this reason, the data-flow construction and alias-analysis can be optionally used with support for *pseudo aliases*. A pseudo alias denotes a shared resource that is not directly supported by the algorithm. Thus, whenever a pointer to a shared resource is accessed via an array-access or struct-access expression and assigned to a local variable or an argument x, then a pseudo-alias (currently represented by an local variable without a name) is added to the alias-set of x. This way, the optimizer can generally also detect recursive locks for not supported data structures, which is illustrated by the following example:

```
struct Container { shared<int32> value; }
void foo() {
  Container c;
  // ... share c with some other task
  sync(&c.value as myValue) {
    myValue->set(0);
    bar(value);
  }
}

void bar(shared<int32>* value) {
  sync(value as myValue) { myValue->set(1); }
}
```

The alias set of the local variable **`myValue`**, which the named resource in line 5 is reduced to, will be added a pseudo-alias l. l will be forwarded to **`value`** in **`bar()`** and reach the local variable **`myValue`** in the same function. Since the synchronization context of **`myValue`** in **`foo()`** will take the same path, the compiler will find that the synchronization of **`myValue`** in **`bar()`** can be removed. However, this approach does not enable the detection of equal pseudo-aliases if they result from different expressions. Therefore, more extensive data structure support should be aimed for in the future.

The second algorithm uses the alias graph and the call graph, which connects the main function with all other functions that are induced by the data-flow graph. It considers every synchronization resource s exactly once by determining whether the respective expression e is synchronized for all aliased shared resources that e might evaluate to. Every alias x must be covered by a synchronization statement, either in the same function f that s resides in or in one of the functions that lie on the paths from the main function f in the call graph. Hence, either s must have an ancestor synchronization statement in the abstract syntax tree of f that has another synchronization statement s', whose aliases contain x. Or on every path from f to *main* there must be at least one call that itself is nested in a synchronization statement with a synchronization resource whose alias set contains x. If for a synchronization resource this property holds, it can be removed. The algorithm expects an inter-procedural analysis. Hence, in order to handle synchronization resources with references to function arguments correctly, it checks the alias-coverage for every call of f separately.

```
                                   ▷ $a$ = aliases, $d$ = add. data, $f\_to\_\bar{\bar{c}}$ = function to main paths' calls
function REMOVE RECURSIVE LOCKS 2($a, d, f\_to\_\bar{\bar{c}}$)
    for all $l \leftarrow$ CALLS OF FUNCTION OF($s$) do
        for all $s \leftarrow d.syncResources$ in $l$ do
            $r$ := true                                  ▷ indicates whether $s$ can be removed
            for all $x \leftarrow a[s.expr]$ do
                $r := r \wedge$ (ALIAS COVERED IN SYNC($x, s, a, f\_to\_\bar{\bar{c}}$) $\vee$ ALIAS COVERED IN PATHS($x, s, a, f\_to\_\bar{\bar{c}}$))
            end for
            if $r$ then
                remove $s$
            end if
        end for
    end for
end function
                      ▷ $x$ = alias, $n$ = current node, $a$ = all aliases, $f\_to\_\bar{\bar{c}}$ = function to main paths' calls
function ALIAS COVERED IN SYNC($x, s, a, f\_to\_\bar{\bar{c}}$)
    return ∃ sync resource $s$ in SURROUNDINGSYNCS($n$) . TASK($n$) = TASK($s$) and $a[s.expr]$.CONTAINS($x$)
end function
                      ▷ $x$ = alias, $s$ = sync resource, $a$ = all aliases, $f\_to\_\bar{\bar{c}}$ = function to main paths' calls
function ALIAS COVERED IN PATHS($x, s, a, f\_to\_\bar{\bar{c}}$)
    for all $\bar{c} \leftarrow f\_to\_\bar{\bar{c}}$ [FUNCTION($s$)] do      ▷ consider all paths to the main function
        if there is no $c$ in $\bar{c}$ with ALIAS COVERED IN SYNC($x, c, a, f\_to\_\bar{\bar{c}}$) then
```

```
            return false
        end if
    end for
    return true
end function
```

Narrowing Synchronization Statements

Synchronization statements are narrowed by iteratively moving statements from the beginning of the statement lists to outside the synchronization statements. This process stops as soon as a statement is encountered for which the movement could introduce new data-races. Likewise statements at the end of each list are moved outside. In order to not interfere with the scopes of local variables, currently all moved statements of a synchronization statement s and s itself are nested inside a new statement block:

```
void foo(shared<int32>* var) {
  boolean b = true;
  sync(var as myVar) {
    boolean b = false;
    myVar->set(0);
  }
}
```

$\underset{\text{narrow}}{\longrightarrow}$

```
void foo(shared<int32>* var) {
  boolean b = true;
  {
    boolean b = false;
    sync(var as myVar) { myVar->set(0); }
  }
}
```

The crucial aspect of the narrowing algorithm is how to decide whether a statement may be moved. If no other optimization has taken place before, a trivial approach would be to check whether a first or last statement of the statement list of a synchronization statement s contains any references to a variable that is also referenced by a synchronization resource of s.[61] Due to the locality of synchronization contexts from a user's perspective, one could argue that this attempt would suffice. However, with the advent of the previous optimization techniques, this simple attempt could fail. If recursive locks are removed, the **.set** and **.get** expressions, which previously referred to according variables, then receive their synchronization contexts from other synchronization resources. This entails for example that synchronization resources need no longer be synchronized locally (in the same function), but can be synchronized across functions. The above mentioned simple approach would not regard such cases. Therefore, not the directly referred variables are compared, but the according aliases. The algorithm is depicted by the following pseudo-code:

```
                        ▷ $a$ = aliases, $d$ = add. data, $\overline{ro}$ = read-only aliases, $\overline{st}$ = single-task aliases
function NARROW SYNCS($a, d, \overline{ro}, \overline{st}$)
    for all $s \leftarrow d.syncStatements$ do
        while there is $t$ = FIRST STATEMENT IN($s$) where CAN BE SHIFTED($t$) do
            $t$.MOVE TO END OF($l_1$)                    ▷ $l_1$ = list of first shifted statements
        end while
        while there is $t$ = LAST STATEMENT IN($s$) where CAN BE SHIFTED($t$) do
```

[61] References to named resources need not be considered because they are resolved before any optimization takes place.

```
            $t$.Move To Start Of($l_2$)                    ▷ $l_2$ = list of first shifted statements
        end while
        if either $l_1$ or $l_2$ is not empty then
            replace old $s$ with new block { $l_1.members, s, l_2.members$ }
        end if
    end for
end function
```

The helper function *Can Be Shifted* gathers all expressions **`e.get`** and **`e.set(_)`** that may be evaluated for a specific statement t. For this purpose, it searches for such expressions: first in the AST of t itself and then in the ASTs of all functions that might be called for the evaluation of t. To this end, the complete function call branching of t[62] is investigated. A shift is declared as safe, if it is safe for every e in the found expressions **`e.get`** and **`e.set(_)`**. In turn, for e, the shift is unsafe, if its alias set is contained in one of the alias sets of the synchronization resources of the synchronization statement s. For this check, only those aliases need to be considered that are not read-only or single-task shared resources. The pseudo-code for *Can Be Shifted* is:

```
        ▷ $t$ = statement, $s$ = sync statement, $a$ = aliases, $\overline{ro}$ = read-only aliases, $\overline{st}$ = single-task aliases
function Can Be Shifted($t, s, a, \overline{ro}, \overline{st}$)
    for all $e.set(\_)$ and $e.get$ in the AST of $t$ or in the AST of every function of Call Branching($t$) do
        if there is a sync resource $r$ in $s$ with $a[r.expr]$.Contains All($a[e] \setminus (\overline{ro} \cup \overline{st})$) then
            return false
        end if
    end for
    return true
end function
```

Again, an intra-procedural analysis should be preferred in order to be able to optimize more aggressively. In the current implementation, however, the optimization does not use such precise information and instead optimizes quite conservatively. Therefore, the optimizer must instead check whether any alias of the variable reference e is contained in the alias set of a synchronization resource r.

[62] This branching comprises all directed subtrees (= aborescences) – of called functions – whose roots lie in the AST of t.

5 Evaluation

5.1 Code and Measurements

In order to show the advantages and disadvantages of the language design, the generator and the optimizer of ParallelMbeddr, two scenarios are investigated: The calculation of π and the sorting algorithm Quicksort. For both scenarios different solutions are presented. This is done to demonstrate the different aspects of the implemented optimization and depict further optimization potential for future enhancements of ParallelMbeddr.

5.1.1 Quicksort

The Quicksort scenario concerns itself with the sorting of an array of items. In order to get noticeable execution times for the program runs, the length of arrays usually has to be quite large. Since general support for the heap is currently missing in mbeddr and ParallelMbeddr, the items need to saved on the stack. Due to the limited usable memory on the stack, the arrays were chosen rather small. To create noticeable execution times nevertheless, the comparison of two items was artificially complicated by a function **doHeavyWork()** that simulates complex comparisons. At the beginning, the arrays are initiated with random values in each example. Afterwards, the usual divide-and-conquer approach of Quicksort is recursively applied.

Serial variant

For comparison reasons, first the serial variant is given. As was mentioned, initially a certain number of randomized items, in this case 200 is added to an array. The items are instances of the **struct Item**, which has an Integer-typed value field. Again, **Item** should be regarded as an element of an arbitrary type.

```
#constant numberOfItems = 200;
struct Item { int32 value; };

int32 main(int32 argc, string[] argv) {
  Item[numberOfItems] items;
  // initialize the array with randomized items
  initItems(items);
  quickSort(items, 0, numberOfItems - 1);
  /* maybe print the result here */
  return 0;
}
```

Not surprisingly, the **quickSort()** function divides the provided interval. As usual with Quicksort, the division is accomplished by sorting the provided interval of the array into two sub-intervals that contain items which are smaller, respectively bigger, than a chosen pivot element:

```
void quickSort(Item[numberOfItems] items, int32 left, int32 right) {
  if (left < right) {
    int32 middle = partition(items, left, right);
    quickSort(items, left, middle - 1);
    quickSort(items, middle + 1, right);
  }
}
int32 partition(Item[numberOfItems] items, int32 left, int32 right) {
  Item pivot = items[left];
  int32 i = left;
  int32 j = right + 1;

  while (true) {
    do { ++i; } while ( !(biggerThan(items[i], pivot)) && i < right );
    do { --j; } while ( biggerThan(items[j], pivot) );
    if (i >= j) { break; }
    swap(items, i, j);
  }

  if (left != j) { swap(items, left, j); }
  return j;
}
```

The comparison function **biggerThan()** compares the two items and simulates computationally complex work by calling **doHeavyWork()**:

```
boolean biggerThan(Item item1, Item item2) {
  doHeavyWork();
  return item1.value > item2.value;
}
void doHeavyWork() {
  for (i ++ in [0..heavyWorkSize[) {
    for (j ++ in [0..heavyWorkSize[) {
      j * j * j;
    } for
  } for
}
```

Unsafe parallelism

A trivial approach to parallelize the Quicksort algorithm would make use of the parallelization concepts, i.e. tasks and futures, while omitting safe communication via shared resources:

```
 1  void quickSort(Item[numberOfItems] items, int32 left, int32 right) {
 2    if (left < right) {
 3      int32 middle = partition(items, left, right);
 4  
 5      if (middle - left > threshold && right - middle > threshold) {
 6        // This is unsafe and should generally be avoided. In the current example, however,
 7        // the sync leak is harmless.
 8        Future<void> sorter1 = |quickSort(items, left, middle - 1)|.run;
 9        Future<void> sorter2 = |quickSort(items, middle + 1, right)|.run;
10        sorter1.join;
11        sorter2.join;
12      } else {
13        quickSort(items, left, middle - 1);
14        quickSort(items, middle + 1, right);
15      }
16    }
17  }
```

The function now decides whether two sub-intervals (slices) of the given interval of the array shall be recursively sorted like in the original example. If the intervals are long enough – have at least 20 items – the calls of **quickSort()** are handed to new tasks, which are immediately run. Future handles are then used to wait for the ending of the sub-tasks. This fork-join pattern is used so that in the end the first call to **quickSort()** does not return before the array is actually sorted. The futures cannot be directly joined, because otherwise line 8 would not be executed before the finish of **sorter1**, which ultimately would serialize the program. In the IDE, the code is marked with an error message at lines 7 and 8. Although every slice of the array is only processed by exactly one task, the type checker of the IDE recognizes a potential shared data leak: the pointer **items** is given to another task which makes every access to the pointed-to data potentially thread-unsafe. ParallelMbeddr is thus currently not able to determine if the elements of shared arrays are actually accessed by multiple tasks. For this reason, arrays must be wrapped in shared resources.

Coarse-Grained Protection

The first solution to the aforementioned problem is to wrap the whole array of items inside a shared resource **shared<Item[numberOfItems]> items**. By doing so, the distribution of pointers thereof becomes safe. The example code is changed to show the optimization potential for synchronization narrowing:

```
boolean biggerThan(shared<Item[numberOfItems]>* items, int32 index1, int32 index2) {
  // There is no noticeable synchronization management overhead in this example since the sync
  // overhead is dominated by the not synchronized heavy work. With doHeavyWork() inside
  // sync(...) the program is basically serialized. Therefore, apply lock narrowing to reduce
  // the amount of lock contention.
  sync(items as myItems) {
    doHeavyWork();
    return myItems->get[index1].value > myItems->get[index2].value;
  }
}
```

As can be seen, **doHeavyWork()** does not make use of the **items** pointers so that it should actually not reside inside the synchronization context. In the given code, this structure would cause unnecessary lock contention.[63] The items pointer is used for the synchronization of the array. The IDE demands that items must be named, to prevent the user from writing code which causes unsafe changes of the pointer that is used for accessing the array, as was explained in 3.4.6. For many synchronizations via pointers this naming necessity can become quite tedious, yet it is currently the only way to guarantee thread-safe access via pointers. Synchronization is now also necessary inside the initialization function and the swap funtion:

```
void initItems(shared<Item[numberOfItems]>* items) {
  for (i ++ in [0..numberOfItems[) {
    // Such fine-grained synchronization is a bad choice, ParallelMbeddr is, however,
    // currently not able to optimize this code.
    sync(items as myItems) { myItems->get[i].value = rand(); }
  }
}
int32 partition(shared<Item[numberOfItems]>* items, int32 left, int32 right) {
  Item pivot;
  // Items may not be taken by address since they are not shared resources themselves.
  // Since the elements of the considered array interval are not changed by another task,
  // in terms of data dependencies, however, it is safe to just copy the element to pivot.
  sync(items as myItems) { pivot = myItems->get[left]; }
  //...
}
void swap(shared<Item[numberOfItems]>* items, int32 i, int32 j) {
  sync(items as myItems) {
    Item temp = myItems->get[i];
    myItems->get[i] = myItems->get[j];
    myItems->get[j] = temp;
  }
}
```

As the code for of the **quickSort()** function in the user code, which was shown in listing (TODO: unsafe par.) is only changed in terms of the type of the **items** argument and therefore skipped in this discus-

[63] Again, in spite of this artificial structure, like every following code listing, the example is only meant to demonstrate the optimization potential for ParallelMbeddr.

sion. The generator translates the type **shared<Item[numberOfItems]>** to the type of the generated struct **SharedOf_ArrayOf_Item_0_t** that stores the array and a mutex which is used for synchronization purposes. The future type **Future<void>** is reduced to the void future type **VoidFuture_t**. The task and future initialization expressions in lines 7 and 8 of XXX are reduced to calls of **futureInit_X()** functions. The following listing shows the generated code for the line 7:

```
  // line 7...
  VoidFuture_t sorter1 = futureInit_0(middle, items, left);


VoidFuture_t futureInit_0(int32 middle, SharedOf_ArrayOf_Item_0_t* items, int32 right) {
  // the type and variable names are simplified for better legibility
  Args_0_t* args_0 = malloc(sizeof(Args_0_t));
  args_0->middle = middle;
  args_0->items = items;
  args_0->right = right;
  pthread_t pth;
  pthread_create(&pth, null, :parFun_0, args_0);
  return ( VoidFuture_t ){ .pth = node: pth };
}
```

The generated code for line 8 is equivalent to the one for line 7 except for names of the arguments and struct fields of **Args_1**. This shows that, if many tasks are defined, the generated code may grow substantially by the definition of **futureInit_X()** functions and argument structure declarations. In the current example, actually a single structure and initialization function would suffice as the types of arguments are equal. However, ParallelMbeddr does not optimize the code in this regard yet. In the future, equivalency checks of task expressions should therefore be implemented to reduce the amount of generated code.

Every synchronization statement is translated to a statement block like the one in the function **biggerThan()**:

```
boolean biggerThan(SharedOf_ArrayOf_Item_0_t* items, int32 index1, int32 index2) {
  {
    SharedOf_ArrayOf_Item_0_t* myItems = items;
    {
      doHeavyWork();
      startSyncFor1Mutex(&(myItems)->mutex);
      {
        stopSyncFor1Mutex(&(myItems)->mutex);
        return myItems->value[index1].value > myItems->value[index2].value;
      }
      stopSyncFor1Mutex(&(myItems)->mutex);
    }
  }
}
```

As expected, the synchronization statement is translated to mutex synchronization functions. An additional call for **stopSyncFor1Mutex()** is added before the return statement in order to prevent inconsistent synchronization states. The next according call is therefore useless and could generally be avoided. ParallelMbeddr does not yet apply a data-flow analysis to prevent the insertion of unreachable 'stop' calls. An according analysis should therefore be added in the future, when the built-in data-flow analysis of mbeddr reaches a reliable state. The statement block indicated by the braces in lines 7 and 10 are added to keep a clear distinction between the scopes of local variables inside the synchronization statement and the scopes of named resources. An alternative to this approach would be to prohibit name equalities of inner local variables and named resources. Line 5 shows that the call of **doHeavyWork()** was moved out of the synchronization context, which was accomplished by synchronization narrowing. For this reason, another statement block (lines 4 and 12) was added, again for the separation of local variable scopes. The outer-most block does the same for outer local variables and local variables resulting from the reduction of named resources. Since in this example code, no conflicting variables exist, these scopes are completely unnecessary and should only be generated on demand in the future.

Fine-Grained Protection

In the former approach, for every access to the array, the whole array had to be synchronized. Alternatively, every element can be separately protected by a shared resource. The type of the **items** variable is then changed to **shared<shared<Item>[numberOfItems]>**. Although the whole array still needs to be protected and synchronized in the user code, the compiler will find that all synchronizations of the complete array can be removed, since none of its members are ever overwritten:[64] Neither the array itself nor any of its items. The user's code must be changed to take into account the element-wise synchronizations:

```
void initItems(shared<shared<Item>[numberOfItems]>* items) {
  for (i ++ in [0..numberOfItems[) {
    sync(items as myItems) {
      sync(&myItems->get[i] as itemI) { itemI->get.value = rand(); }
    }
  }
}
int32 partition(shared<shared<Item>[numberOfItems]>* items, int32 left, int32 right) {
  shared<Item>* pivot;
  sync(items as myItems, &myItems->get[left] as itemLeft) { pivot = itemLeft; }
  // ...
}
boolean biggerThan(shared<shared<Item>[numberOfItems]>* items, int32 index1, int32 index2) {
  sync(items as myItems) {
    doHeavyWork();
    sync(&myItems->get[index1] as item1, &myItems->get[index2] as item2) {
```

[64] Actually, non of the items can ever be overwritten, since the IDE would otherwise detect an unsafe, thus not allowed, overwrite of a shared resource and trigger an error message.

```
      return item1->get.value > item2->get.value;
    }
  }
  return false;
}
void swap(shared<shared<Item>[numberOfItems]>* items, int32 i, int32 j) {
  sync(items as myItems) {
    sync(&myItems->get[i] as itemI, &myItems->get[j] as itemJ) {
      Item temp = itemI->get;
      temp = itemI->get;
      itemI->set(itemJ->get);
      itemJ->set(itemI->get);
    }
  }
}
```

Again, the reduction of **biggerThan()** shows the performed optimizations:

```
boolean biggerThan(SharedOf_ArrayOf_SharedOf_Item_0_0_t* items, int32 index1, int32 index2) {
  {
    SharedOf_ArrayOf_SharedOf_Item_0_0_t* myItems = items;
    {
      doHeavyWork();
      {
        SharedOf_Item_0_t* item1 = &myItems->value[index1];
        SharedOf_Item_0_t* item2 = &myItems->value[index2];
        startSyncFor2Mutexes(&(item1)->mutex, &(item2)->mutex);
        {
          stopSyncFor2Mutexes(&(item1)->mutex, &(item2)->mutex);
          return item1->value.value > item2->value.value;
        }
        stopSyncFor2Mutexes(&(item1)->mutex, &(item2)->mutex);
      }
    }
  }
}
```

The call **doHeavyWork()** is again shifted outside the synchronization context. Furthermore, the lock of **myItems** is dropped. In consequence, only the single array elements need to be locked. Thus, the lock contention of the previous approach was replaced by a synchronization management overhead for the individual elements.

Measurements

The presented approaches – except for the unsafe variant of the Quicksort algorithm – were timed for their execution times. The synchronization based programs were both tested with and without applied optimizations. The constants **numberOfItems** and **heavyWorkSize** (n and h in the following text) were adjusted to provide different constellations:

- for all approaches: $h = 1000$ with $n = 50$, $n = 100$ and $n = 200$
- for the fine-grained approach: $n = 20000$ with $h = 100$, $h = 50$ and $h = 0$

Every timing-test was executed 10 times in order to account for measuring errors.

n	50	100	200
average mean	0.721	1.693	3.920
standard deviation	0.006	0.005	0.015

Table 1: Run-times for serial Quicksort

	non-optimized			optimized		
n	50	100	200	50	100	200
average mean	0.718	1.707	3.944	0.723	1.117	2.135
standard deviation	0.001	0.006	0.009	0.004	0.005	0.011

Table 2: Results for coarse-grained Quicksort

The timings in tables 1 and 2 show that the non-optimized version of the coarse-grained implementation of Quicksort shows basically the same performance as the serialized one (which is obviously not optimized). This is no surprise as the code serializes most of the work that is performed by the algorithm. The optimized version shows the potential of lock narrowing: The speed-up of the optimized version converges to 2, which is also the maximum speed-up that can be achieved by the two processor cores on the test-system. This indicates, that the synchronization overhead in the optimized variant is dominated by the work of **doHeavyWork()**. For $n = 50$ no speed-up can be determined (the additional 5 ms can be attributed to measuring errors) which indicates that the number of items is too small too allow for parallel execution. Thus, no further tasks are initiated.

	non-optimized			optimized		
n	50	100	200	50	100	200
average mean	0.756	1.832	4.364	0.743	1.365	2.488
standard deviation	0.007	0.014	0.012	0.005	0.007	0.012

Table 3: Results for fine-grained Quicksort with $h = 1000$

For the according configuration of the fine-grained implementation of Quicksort, table 3 indicates only a small performance overhead when compared to the serial variant. The times result from the fact that the main work is again serialized, while additional synchronization for the single items is performed.

The overhead increases with the number of items that are sorted. This overhead is also reflected in the optimized version, which is thus slower than the coarse-grained implementation. Due to the dominance of **doHeavyWork()** the synchronization of the whole array seems to be negligible (when optimized) while the synchronization of the single elements noticeably decelerates the algorithm. Thus, the numbers suggest that such fine-grained synchronization can generally cause a significant performance loss, even if no actual lock clashes happen (since every element is only accessed by one task at a time). However, the timings in table 4 reveal that due to reduction contention also coarse-grained synchronization can cause severe performance losses. In this setting, the impact of **doHeavyWork()** is mitigated by adjusting h to 0, 50 and 100. In exchange, the number of items is set to 20000, in order to register a noticeable synchronization overhead. The numbers of the optimized version suggest that the main part of the execution time results from the amount of additional comparison work of **doHeavyWork()**. For $h = 0$, the execution time converges to 0. On the other hand, the execution times of the non-optimized version are significantly worse than those of the former runs. Even with no additional comparison overhead ($h = 0$), the program is slowed down to twice the worst execution time of the optimized variant.
These numbers show that the removal of read-only locks can at times lead to a significant performance boost, which is true also for the narrowing of synchronization statements.

	non-optimized			optimized		
h	0	50	100	0	50	100
average mean	6.587	8.609	13.354	0.132	0.809	3.129
standard deviation	0.116	0.085	0.101	0.001	0.006	0.025

Table 4: Results for fine-grained Quicksort with $n = 20000$

5.1.2 π

The π scenario is a continuation of the running example from chapter 3. To recap the idea: The number *pi* is calculated by iteratively adding numbers which are given by a function on the natural numbers. The precision of the resulting *pi* approximation correlates to the numbers that are added.

Serial Variant

The serial variant of this algorithm just adds the mapped values of 0 up to a certain threshold n to the result. The following code listing already indicates the block approach which is used by the parallel variants of this algorithm:

```
#constant BLOCKSIZE = 30000000;
#constant BLOCKCOUNT = 4;
#constant THRESHOLD = BLOCKSIZE * BLOCKCOUNT;

exported int32 main(int32 argc, string[] argv) {
  double result;
  for (int32 i = 0; i < THRESHOLD; i += BLOCKSIZE) {
    calcPiBlock(&result, i, i + BLOCKSIZE);
  }
  return 0;
}

long double calcPiBlock(uint32 start, uint32 end) {
  long double result = 0;
  for (uint32 i = start; i < end; ++i) {
    result += calcPiItem(i);
  }
  return result;
}

long double calcPiItem(uint32 index) {
  return 4.0 * (pow(-1.0, index) / (2.0 * index + 1.0));
}
```

Parallel Variant without Synchronization

The first parallel variant that is evaluated is the future-based algorithm whose translation was shown in 3.1.3:

```
exported int32 main(int32 argc, string[] argv) {
  long double result = 0;
  Task<long double*>[BLOCKCOUNT] calculators;
  Future<long double*>[BLOCKCOUNT] partialResults;

  for (i ++ in [0..BLOCKCOUNT[) {
    uint32 start = ((uint32) i) * BLOCKSIZE;
    uint32 end = start + BLOCKSIZE;
    calculators[i] = |calcPiBlock(start, end)|;
  }

  for (i ++ in [0..BLOCKCOUNT[) {
    partialResults[i] = calculators[i].run;
    calculators[i].clear;
  }

  for (i ++ in [0..BLOCKCOUNT[) {
    result += *(partialResults[i].result);
    free(partialResults[i].result);
  }
21
22    return 0;
23  }
```

The algorithm divides the interval of numbers, which shall be mapped to the terms that ultimately add up to the result, into blocks. Each block is assigned to a task that calculates partial sum according to the numbers of its block. The partial sums are then accessed in line 19 via the future handles of the running tasks. The memory of the results, which reside in the heap, are then freed, although at the end of the program this is usually not necessary. No synchronization is used in the given code as all communication happens implicitly and goes uni-directional from the calculator tasks to the main task via the result values. The translation of the code is skipped, as it was already shown in 3.1.3 and, for the lack of applied optimization, would not enrich the discussion of this chapter.

Simple Map-Reduce Approach

The second variant of the π algorithm equals the one presented in 3.3.4: The blocks are now not assigned at the definition of the tasks but via communication among the tasks. The *mapper* tasks use a **counter** to communicate the current progress of processed blocks. Furthermore, their partial sums are no longer received by the result values of their future handles. Instead, a shared queue is used to communicate the partial sum of each block to a reducer task:

```
exported int32 main(int32 argc, string[] argv) {
  shared<Queue> queue;
  queueInit(&queue);  // set all slots to 0
  shared<Queue>* queuePointer = &queue;

  shared<uint32> counter;
  shared<uint32>* counterPointer = &counter;
  sync(counter) { counter.set(0); }

  Task<void> mapperTask = |map(THRESHOLD, counterPointer, queuePointer)|;
  Future<void>[MAPPERCOUNT] mappers;
  for (i ++ in [0..MAPPERCOUNT[) { mappers[i] = mapperTask.run; }
  mapperTask.clear;

  shared<long double> result;
  shared<long double>* resultPointer = &result;
  |reduce(BLOCKCOUNT, resultPointer, queuePointer)|.run.join;

  return 0;
}

void map(uint32 threshold, shared<uint32>* counter, shared<Queue>* resultQueue) {
  while (true) {
```

```
24      uint32 start, end;
25
26      sync(counter as myCounter) {
27        start = myCounter->get;
28        if (start == threshold) { break; }
29        uint32 possibleEnd = start + BLOCKSIZE;
30        end = (possibleEnd <= threshold)?(possibleEnd):(threshold);
31        myCounter->set(end);
32      }
33
34      queueSafeAdd(resultQueue, calcPiBlock(start, end));
35    }
36  }
37
38  void reduce(uint32 numberOfItems, shared<long double>* result, shared<Queue>* resultQueue) {
39    sync(result as myResult) {
40      myResult->set(0);
41      for (uint32 i = 0; i < numberOfItems; ++i) {
42        long double item;
43        queueSafeGet(resultQueue, &item);
44        myResult->set(item + myResult->get);
45      }
46    }
47  }
```

The reducer task knows in advance how many blocks it is going to receive. It can therefore count the received elements to in line 41 to determine when it is finished. For simplicity reasons, the reducer synchronizes the shared resource of result variable during its whole lifetime. While such patterns can generally lead to deadlocks, it is safe in the current example as no other task acquires the result before the finish of the reducer. The mapper tasks do not know in advance the number of blocks that they will process which is why they use an unconditional loop in line 23. They can, however identify the last block via **threshold** and stop as soon as the **counter** variable has reached this value. It is crucial for the mappers to synchronize the counter variable from line 26 to line 32 and not apply fine-grained synchronization in order to avoid race conditions. It is the responsibility of the programmer to identify the according data dependencies and synchronize appropriately. For the current atomicity pattern (read a shared resource, process its value and overwrite its value) it might be possible to give first-class support in mbeddr in the future. However, the programmer would still have to identify shared resources whose usage match this pattern.

The queue whose functions are called in lines 3, 34 and 43, manages two fields **insertAt** and **deleteAt** which keep the indexes of the next free, respectively occupied slot of the queue. Every item of the queue is separately shared:

```
exported struct Queue {
  int32 insertAt;
  int32 deleteAt;
  shared<long double>[QUEUESIZE] data;
};
```

The queue works like a ring buffer: Items are always added at the front and removed at the back while the according indices **insertAt** and **deleteAt** are managed as if the last and the first slot of the array were connected. The implementation of the queue functions are not optimized for best performance in the user code. Instead, they are written in a way that shows how the optimizations of the compiler may take place. The called functions are defined as follows:

```
void queueInit(shared<Queue>* queue) {
  sync(queue as myQueue) { myQueue->get.insertAt = myQueue->get.deleteAt = 0; }
}

void queueSafeAdd(shared<Queue>* queue, long double item) {
  while (true) {
    sync(queue as myQueue) {
      int32 newInsertAt = (queueGetInsertAt(queue) + 1) % QUEUESIZE;
      int32 deleteAt = queueGetDeleteAt(queue);
      if (deleteAt == newInsertAt) { continue; }
      queueSetItemAt(queue, queueGetInsertAt(queue), item);
      queueSetInsertAt(queue, newInsertAt);
      break;
    }
  }
}

void queueSafeGet(shared<Queue>* queue, long double* result) {
  while (true) {
    sync(queue as myQueue) {
      // see above at queueSafeAdd()
      if (queueGetDeleteAt(queue) == queueGetInsertAt(queue)) { continue; }
      *result = queueGetItemAt(queue, queueGetDeleteAt(queue));
      int32 newDeleteAt = (queueGetDeleteAt(queue) + 1) % QUEUESIZE;
      queueSetDeleteAt(queue, newDeleteAt);
      return;
    }
  }
}
```

queueSafeAdd() and **queueSafeGet()** repeatedly check the indices of the queue for free, respectively occupied slots. If none is available, the check is repeated in a busy-wait manner. As the repetitions do not perform a wait after an unsuccessful check, the code will ultimately cause unnecessary workload for the CPUs. The user should therefore force the tasks to sleep for a certain amount of time in between. Yet, it may be difficult to balance the waiting time for acceptable responding times and CPU occupancy. For this reason it might be helpful in the future to offer conditional variables which, in the current example, would allow to wait for a free or occupied slot without having to worry about the implementation of a performant busy-wait approach. Such condition variables could either suspend a thread until the declared conditions are fulfilled or realize a busy-wait protocal similar to the one for the acquisition of locks. The two functions synchronize all statements in the busy-wait loops. This is done in order to prevent interfering changes of the index variables of the queue. The accesses to the queue thus become serialized (for this reason, in this example, the separate protection of each queue slot is actually superfluous). Therefore, the repeated acquisition and release of the lock for the queue, which causes unnecessary synchronization management overhead, could be optimized by instead nesting the loops inside the synchronization statements. This optimization technique, called lock coarsening, however must be used with care in order to not impair the responsiveness due to lock contention (see also section 5.3.5).

Of the remaining functions of the queue, the function **queueSetItemAt()** is given. The other helper functions have the same structure.

```
void queueSetItemAt(shared<Queue>* queue, int32 index, long double newItem) {
  sync(queue as myQueue) {
    sync(&myQueue->get.data[index] as wrappedItem) { wrappedItem->set(newItem); }
  }
}
```

queueSetItemAt() first has to synchronize the access to the shared queue. It then does the same for the slot of the queue whose value is to be overwritten. At this point, two optimization are possible. First, the queue need not be locked, since for the only call of this function the queue is already synchronized. Recursive-lock optimization can be performed. Furthermore, since every synchronization of the queue slots happen in a context where also the whole queue is synchronized, the latter synchronization dominates the former one. Therefore, the slots actually need not be synchronized. This technique is similar to the removal of enclosed monitor locks in Java (see 5.3.5). If the necessary properties for these optimizations only hold for some calls of **queueSetItemAt()** function inlining or duplication could be performed (see 4.2.3). A look at the generated code for the current example reveals that of the presented optimizations the removal of the recursive lock is performed:

```
void queueSetItemAt(SharedOf_Queue_0_t* queue, int32 index, long double newItem) {
  {
    SharedOf_Queue_0_t* myQueue = queue;
    {
      {
        SharedOf_long_double_0_t* wrappedItem = &[| node: myQueue -> value |].data[index];
        startSyncFor1Mutex(&(wrappedItem)->mutex);
        { wrappedItem->value = newItem; }
        stopSyncFor1Mutex(&(wrappedItem)->mutex);
      }
    }
  }
} queueSetItemAt (function)
```

The synchronization for **myQueue** is removed, whereas the one for **wrappedItem** remains, which is the expected result of the implemented optimizer. On the other hand, the synchronizations of the queue in the aforementioned functions **queueSafeGet()**, **queueSafeAdd()** and **queueInit()** remain, as expected.

Extended Map-Reduce Approach

While in the last version of the π algorithm the mappers shared a single queue, the extended variant assigns one queue to each mapper. The reducer gets access to each of these queues. Furthermore another container is introduced for the queues:

```
struct FlaggedQueue {
  shared<int32> itemCount;
  shared<Queue> queue;
  shared<boolean> isFull;
  shared<boolean> isFinished;
};
```

This container is used in the following way. Whereas in the former approach partial sums were communicated, in the extended variant, a mapper adds every single item to the queue (this is, of course an approach which, for performance reasons, would not be pursued in real life). The mapper repeatedly fills the queue with as many items as possible, which depends on the number of free slots left in the queue and on the items left to be calculated for the current block of numbers. After each such round in signals the reducer via **isFull** that it is done for the moment and via **itemCount** how many items were actually added to the queue (since the queue itself does offer an according functionality). When the mapper is finished, it signals the reducer this informs the reducer via **isFinished**, so that reducer finally knows when it is finished itself. The result queues are declared in the main function:

```
exported int32 main(int32 argc, string[] argv) {
  // ...
  shared<shared<FlaggedQueue>[MAPPERCOUNT]> resultQueues;
  shared<shared<FlaggedQueue>[MAPPERCOUNT]>* resultQueuesPointer = &resultQueues;
  Future<void>[MAPPERCOUNT] mappers;
  for (i ++ in [0..MAPPERCOUNT[) {
    sync(resultQueues, &resultQueues.get[i] as resultQueue) {
      sync(&resultQueue->get.itemCount as itemCount) { itemCount->set(0); }
      sync(&resultQueue->get.isFull as isFull) { isFull->set(false); }
      sync(&resultQueue->get.isFinished as isFinished) { isFinished->set(false); }
      queueInit(&resultQueue->get.queue);
      mappers[i] = |map(THRESHOLD, counterPointer, resultQueue)|.run;
    }
  }

  shared<long double> result;
17   shared<long double>* resultPointer = &result;
18   |reduce(resultPointer, resultQueuesPointer)|.run.join;
19 }
```

The result queues are wrapped inside a shared resource in order to be safely shareable with the reducer. in line 18. The code reveals a circuitousness which results from the copy-semantics of tasks: In order to actually share **resultQueues** and **result** with the reducer, according pointers have to be defined in advance. If the programmer would instead reference these data via address like in

```
|reduce(&result, &resultQueues)|.run.join;
```

then actually not addresses of the original shared resources would be copied. Instead, the shared resources would be copied themselves and pointers to these copies would in turn be copied into the reducer task. Ultimately, this would lead to erroneous code. This is a result from the fact that the value of every referenced variable inside a task expression is copied into the arguments structure, which the thread for this task is fed at runtime. In the future the semantics should probably be distinguished in this regard, such that every address-referenced variable of a shared resource is actually copied by its address not by its value. The aforementioned mapper and **calcPiBlock** are changed to the following code:

```
void map(uint32 threshold, shared<uint32>* counter, shared<FlaggedQueue>* partialResultQueue) {
  while (true) {
    uint32 start, end;
    // ...
    calcPiBlock(start, end, partialResultQueue);
  }
  sync(partialResultQueue as myQueue, &myQueue->get.isFinished as isFinished) { isFinished->set(true);
}

void calcPiBlock(uint32 start, uint32 end, shared<FlaggedQueue>* resultQueue) {
  int32 mapCounter = 0;
  for (uint32 i = start; i < end; ++i) {
    sync(resultQueue as queue) { queueSafeAdd(&queue->get.queue, calcPiItem(i)); }
    ++mapCounter;

    if (mapCounter == QUEUESIZE - 1 || i == end - 1) {
      sync(resultQueue as queue, &queue->get.itemCount as itemCount) { itemCount->set(mapCounter); }
      sync(resultQueue as queue, &queue->get.isFull as isFull) { isFull->set(true); }
      mapCounter = 0;
      while (true) {
        sync(resultQueue as queue, &queue->get.isFull as isFull) { if (!isFull->get) { break; } }
        timespec sleepingTime = (struct timespec){ .tv_nsec = };
        nanosleep(&sleepingTime, null);
      }
    }
  }
}
```

As already explained, inside **calcPiBlock()** the mapper adds many items to the queue as possible. It then changes the flags **itemCount** and **isFull** so that the reducer will empty the queue. The mapper thus subsequently waits in a busy-wait manner for the **isFull** flag to become invalid.

```
 1  void reduce(shared<long double>* result, shared<shared<FlaggedQueue>[MAPPERCOUNT]>* resultQueues) {
 2    sync(result as myResult) { myResult->set(0); }
 3
 4    boolean[MAPPERCOUNT] areFinished;
 5    for (i ++ in [0..MAPPERCOUNT[) { areFinished[i] = false; }
 6    int32 isFinishedCount = 0;
 7
 8    while (true) {
 9      // try to read from one queue if any one is available
10      for (i ++ in [0..MAPPERCOUNT[) {
11        if (areFinished[i]) { continue; }
12        sync(resultQueues as myQueues, &myQueues->get[i] as resultQueue) {
13          sync(&resultQueue->get.isFull as isFull, &resultQueue->get.isFinished as isFinished) {
14            if (isFull->get) {
15              addPartialResults(result, &resultQueue->get.itemCount, &resultQueue->get.queue,
16                                &resultQueue->get.isFull);
17            } else if (isFinished->get && setFinished(i, areFinished, &isFinishedCount)) {
18              return;
19            }
20          }
21        }
22      }
23      timespec sleepingTime = (struct timespec){ .tv_nsec = DELAY };
24      nanosleep(&sleepingTime, null);
25    }
26  }
```

The synchronizer busy-waits for one of tasks to be either finished (line 17) or for the queue of the respective task to become full. In the latter case **addPartialResults** adds the items of the filled slots to the overall result, and invalidates the **isFull** flag:

```
void addPartialResults(shared<long double>* result, shared<int32>* itemCount, shared<Queue>* queue, sha
  sync(queue as myQueue, itemCount as myItemCount) {
    for (i ++ in [0..myItemCount->get[) {
      sync(result as myResult) {
        long double tempResult;
        queueSafeGet(myQueue, &tempResult);
        myResult->set(myResult->get + tempResult);
      }
    }
  }
  sync(isFullFlag as flag) { flag->set(false); }
}
```

The previous code listings show some optimization potential. First, **resultQueues** need not be synchronized in line 12, since array elements are not allowed to be overwritten. The same holds for the container **resultQueue** in line 12 (TODO: of listing reduce). On the other hand, since **resultQueue** and its counterparts in line lines 7, 13, 17, 18 and 21 (TODO: of listing map), always dominates the its fields synchronization-wise, an optimization might find that the synchronization of the whole queue and a removal of the synchronizations of its fields may lead to better performing code. Such optimization assessments would require some sort of heuristics and should be considered for future extensions of ParallelMbeddr. Nevertheless, the synchronization in line 2 (TODO: of addPR) of **myQueue** causes the queue to be always synchronized in **queueSafeGet()** since line 6 contains the only call for this function. Thus, the synchronization inside this function, which was shown in the previous parallelization approach, can be removed; along with all recursive locks in the helper functions for the queue. Another subtle optimization could be applied for every synchronization of the queue: Since the use of **isFull** forbids the mappers to add new items to the queue as long as this flag evaluates to **false** and vice versa for the reducer, there can actually never be any conflicting accesses to the queues fields. The utilization of the queue thus induces two states that are determined by the value of **isFull**. It is left to future research, how such state-dependent synchronization can be exploited for the elision of locks. The actual performed optimizations concern the synchronizations of the overall **resultQueues** container and those of the individual actual queues. The optimizer detects the read-only usage of **resultQueues** and infers that locks thereof can be removed. It should be noted, that the optimizer is not able to detect this usage pattern for each queue container in the array of **resultQueues** since the according optimization for nested shared resources is not supported, yet. However, due to the use of pseudo-aliases, in an unsafe mode optimization mode, the optimizer detects the recursive locks of the individual queues in **queueSafeGet()** and the other helper functions and deletes them. For the lack of relevance, the optimized code is omitted in this discussion.

Measurements

5.2 Optimization

Standard deviation Average mean

5.3 Comparison

While there are multiple approaches to parallel programming in C, to the knowledge of the writer none offer the combination of thread-safety by design and approaches for optimization. Most progress seems to come from other languages. Therefore, the following paragraphs will present relevant ideas of a few languages that either already influenced the design and optimization of ParallelMbeddr or might do so in the future. The comparison starts with a short depiction of one of the most well-known parallelization enhancements[65] for C.

5.3.1 OpenMP

OpenMP[66] is an API to parallel programming for C/C++ and Fortran[3, p. 1]. It supports both task parallelism and data parallelism. OpenMP is used by extending valid C code with the **`#pragma`** directive,[67] which instructs the compiler how to parallelize a program. It can therefore be seen as a declarative way of enriching single-threaded programs with parallelism. However, OpenMP does not check the code for correctness regarding thread-safety. This means that it is the programmer's responsibility to ensure that data dependencies do not entail data races. Nevertheless, it offers extensive support for parallelization. For instance, OpenMP offers multiple strategies to distribute the processing of arrays onto multiple threads. This, of course, is much easier to realize with the absence of strong thread-safety guarantees. The following language, on the other hand, fills this gap by combining both features. In this regard it might serve as one possible Paragon for the future development of ParallelMbeddr.

5.3.2 ParaSail

The programming language *ParaSail* offers implicit parallelism. It does so by guaranteeing at compile time that, theoretically, every valid expression can be evaluated in parallel. In contrast to 'pure' languages like Haskell, it does not sacrifice side-effects for this purpose [66]. However, in order to automatically prove the safeness of parallel executions, it avoids concepts that are crucial to languages like C, like pointers and global variables [65]. In these regards, it differs from the focus of this work. Nevertheless, its approach to data collections may also provide ideas for future research: ParaSail makes heavy use of indexable containers like lists or trees. When equipped with proper pre- and postconditions, containers can be sliced in a thread-safe manner [64]. This means that they must be designed in a way which lets the compiler prove that slices (i.e. sub-intervals) of their elements do not overlap. In such a case, segmented views of the original container can be manipulated in a thread-safe manner in parallel. While this feature seems to be promising for future enhancements of ParallelMbeddr, it should be noted that the presence of pointers in C could impede an according realization of containers.

5.3.3 Æminium

Like ParaSail, the programming language *Æminium* realizes implicit parallelism. As opposed to ParaSail, it uses explicit annotations on methods by the user to guarantee thread-safety. These annotations declare *access permissions* ([14]) on the parameters of function arguments and return values. The compiler uses these access permissions (*shared*, *immutable*, *unique*) to prove the correctness of programs in terms of thread-safety. If possible, executions are parallelized automatically. This may for instance happen if only

[65] Enhancement is meant in the sense of easier utilization.
[66] http://openmp.org/wp/, accessed: 2014-08-23
[67] See https://gcc.gnu.org/onlinedocs/cpp/Pragmas.html for details.

immutable data is involved. The presence of mutable shared data, however, requires the programmer to wrap according statements inside atomic blocks. While atomic blocks could be inferred by the compiler, the language pursues this path “for granularity reasons” [63]. This approach equals ParallelMbeddr’s requirement to annotate synchronization blocks explicitly. Both languages thus enable the user to “have fine-grain control over the size of critical sections” [63]. Thus, the user controls the trade-off between the responsiveness (the synchronization contention, i.e. the time that threads are blocked by others) of the system and the synchronization management overhead. Furthermore, it can directly influence the presence of higher-level data dependencies, that “cannot [be] directly inferred via data dependencies” [63]. To this end, Æminium offers data groups that allow the user to specify which data must be synchronized together. An according language feature would enrich ParallelMbeddr with better support for higher-level dependencies. Furthermore, data groups would allow the compiler to make use of less mutexes.

5.3.4 D

A similar feature to data groups in Æminium is offered by the multi-paradigm language *D*.[68] D provides a thread-safety-first approach to parallelization. The programmer has to manually protect data that is shared between threads. One way of achieving this goal is by synchronizing classes. If a class is synchronized, its data must be private, and every call of one of its methods is locked for unique access. If fields themselves have non-primitive types, i.e. class-types, their classes must also be synchronized. In order to tie the protection of such a field and an object more closely, D allows objects to be owned by others in terms of their mutexes. An owned object uses the mutex of its owner. D thus enables semantics similar to data groups in Æminium. While this feature is still lacking in ParallelMbeddr, the language offers D’s support for explicit synchronization for multiple objects in one synchronization statement for coarse-grained, deadlock-free synchronization. While ParallelMbeddr uses a busy-waiting approach to achieve this goal, D uses mutex ordering, which acquires mutexes in the same order in all threads, regardless of the syntactic order in the programmer’s code [9].[69]

5.3.5 Java

The programming language Java has a form of synchronization similar to D’s. However, its guarantees concerning thread-safety are considerably weaker. In Java, the focus of synchronizations is rather computation-oriented than data-oriented. The programmer is himself responsible for identifying the blocks of statements and methods that need to be synchronized, and needs to annotate them. Unlike in D, a class can thus have synchronized and not synchronized methods. Furthermore, although the access through a method *m* of such a *monitor* (i.e. an object which applies the explained form of synchronization) is synchronized, simultaneous access via unsynchronized other methods to the same data,

[68] http://dlang.org/, accessed: 2014-08-23

[69] For general information on this technique, see https://www.securecoding.cert.org/confluence/display/seccode/POS51-C.+Avoid+deadlock+with+POSIX+threads+by+locking+in+predefined+order, accessed: 2014-08-23.

which m accesses, is still possible. In spite of these limitations, the research on lock-related optimizations in Java was used as a starting point for similar optimizations in ParallelMbeddr. The optimizations accomplished in Java include lock-elision for re-entrant monitors, i.e. monitors whose synchronized methods are called recursively. Furthermore, enclosed and thread-local monitors are optimized. The latter relates to single-task lock elision in ParallelMbeddr. The former can be seen as an optimization technique that was not applied in ParallelMbeddr. If the field f of a monitor m is always accessed by a synchronized method of m, and f is itself a monitor, then the synchronizations for f's methods can be omitted. Similar, in ParallelMbeddr the synchronizations for a shared resource n, which is nested inside another one s, could be omitted, if n is always synchronized inside a synchronization context of s [7]. Another optimization applied in Java is lock-coarsening, which tries to reduce the synchronization management overhead by broadening synchronization contexts [29]. Such optimization can increase the amount of lock contention. For this reason, good heuristics are mandatory. Another optimization is adaptive locking, which switches between spinning (for short time intervals) and suspension (for longer ones) when a thread waits for the release of a lock. Another technique, which was proposed for Java, is lock reservation [40]. Lock reservation assumes that locks are repeatedly requested by one thread at a time. It exploits this property by storing at runtime for each lock an owner-thread which might change anytime. While accesses via a lock l of the o owner thread need not be locked, other threads have to lock l. Hence, if for a certain amount of time o is the only thread to enter the synchronized methods of a monitor protected by l, then meanwhile no locking for l has to be applied. It remains to be investigated whether such optimization is suitable for the resource-constraint embedded domain.

5.3.6 Rust

While Java tries to reduce the amount of necessary locks at compilation time, the programming language *Rust*[70] cedes this effort to the programmer by providing appropriate data types. As a systems programming language, Rust targets performance, but also memory-safety (i.e. null pointers and dangling pointers are prevented) and thread-safety. Among other things, the latter is accomplished by the use of owned and borrowed pointers. If data is owned by a thread, the thread can safely access it without the need of synchronization. The data can be used inside functions by borrowing a pointer to the function. This pointer has a limited lifetime and can never leave its creator thread. However, ownership can be transfered to other threads. For data that need to be accessed from multiple threads, Rust offers the wrapper type **`Mutex<T>`** which "provides synchronized access to the underlying data" [4]. A mutex in Rust is the equivalent to a shared resource in ParallelMbeddr. Any access to the underlying data must be accompanied by a lock of the mutex container. Rust further offers a signal-wait approach via condition variables for the type **`Mutex<T>`**. Concluding, with the depicted language features and various others, Rust offers manifold opportunities to avoid locks in the first place. For this reason, Rust can serve as a prototype for future language-based enhancements of ParallelMbeddr.

[70] http://www.rust-lang.org/, accessed: 2014-23-08

6 Conclusion

This thesis showed an approach to leverageing mbeddr's language extensibility support for providing advanced support for parallel programming in the embedded domain with C. By developing the new language abstraction ParallelMbeddr on top of mbeddr, the work provided an explicit parallelization approach for task based parallelism. For this purpose the concepts 'task' and 'future' were developed. It was found that these concepts enable a convenient approach to parallel execution of code: The definition of code that shall be executed in parallel is significantly facilitated when compared to working with low level libraries like POSIX threads (as can be seen at the generated code which makes use of pthreads). The thread-safe communication between tasks was realized by shared resources which wrap data to be shared between tasks. The communication via shared resources was restricted to explicit synchronization contexts that the user had to define by making use of the synchronization statement concept. By enhancing the type system with a shared data type for shared resources and restricting the ways shared resources may be used, ParallelMbeddr thus simplifies reasoning of side-effects of communication between tasks. Thus, the creation of thread-safe code should be facilitated. By applying explicit synchronization, the system forces the programmer to think about the right scoping of synchronization contexts. The guaranteed thread-safety, however, is currently limited to low-level data-dependencies. Languages like Æminium show how to approach this problem and should be used as paragons for future extensions of ParallelMbeddr: Dependencies between data should be expressed at the definition of the data, not only when it is used. Otherwise, the language suffers from the danger of data-races resulting from higher-level data-dependencies just as Java does with low-level data-dependencies, if methods or statement blocks are not properly synchronized.

As in mbeddr the safety checks happen in the background in real-time, the user directly receives feedback concerning safety issue of the written code. However, this approach demands the checks to be not too complex computational-wise. Hence, the code currently burdens the user with lexical scopes of synchronization statements. One approach to mitigate this problem in the future is the introduction of further data types or type annotations (for instance **`synced<t>`**) by the user, which diminish the use of synchronization – hence, locks in the generated code – in the first place. Another type-based approach would be the introduction of owned pointers and borrowed pointers, as they are implemented in Rust, which eliminate the necessity for synchronization altogether. Such type advances could both make the code more explicit (and certainly more complicated) and accelerate the generated code. While these approaches cede the reduction of locks to the user, inspired by research on Java, this work showed how to perform performance optimizations in the compilation process. It presented algorithms for the reduction of single-task and read-only locks, which share the property of removing locks for whole shared resources, whenever this is possible. Approaches for the removal of recursive locks were given next. The test results in the evaluation showed evidence, that specifically for the current lexical scopes of synchronizations (and the necessarily resulting recursively locks), such optimization should certainly be applied, in order to mitigate the resulting synchronization management overhead. On the other hand, single-task locks seem to be either a code smell for an unnecessary uses of shared data by the user. Or they result from re-use of library code. The programming and re-use of library code which makes use of ParallelMbeddr's concepts is currently not supported and should be a concern of future research. As the optimization algorithms generally benefit from an inter-procedural alias-analysis, which can not be performed when libraries are written, optimizations of such code, too, is left to future research. While for these reasons and the similarity to read-only locks, single-tasks where not evaluated (although extensively tested), the removal of read-only locks was. It was shown that the optimization of shared containers of shared data can in some cases improve the performance of the generated code. The last optimization, the data-dependence safe narrowing of synchronization statements can also improve the performance of the executed code significantly. The presented optimizations should, however, be seen as prototypes, since in the implementation they support only simple data structures. Additionally, the lack of a precise alias analysis in mbeddr limits the scope of optimizations that can safely be executed with the prototypical analysis that was implemented as a surrogate. For the evaluation, the recursive-lock reduction therefore was configured to perform optimizations which were safe for the considered scenarios, but can be dangerous – in terms of thread-safety – for others. Two important fields of future research should therefore be the extension of mbeddr with a precise alias-analysis and general optimization support for more complex data structures. The presented algorithms should greatly profit from this.

Another task for the future is the development of algorithms which do not, unlike the presented ones, assume a simultaneous execution of threads but, instead, take the succeeding and interleaved execution of tasks into account. This way, those locks could for instance be removed which synchronize data that up until a certain point are only used by a single task. Additonally, apart from static analysis for the detection of shared data of interleaved tasks which are guaranteed to be used by only one task at a time, model checkers like CBMC could be employed for the same purpose. CBMC can be used to perform test-runs of code which makes use of assertions. Additionally, for programs that are simple enough, CBMC can verify the absence of false assertions by unwinding all possible paths through the program [1].[71] An optimization analysis could exploit this feature by assigning access counters to shared resources, removing synchronization statements and asserting that despite this removal, in every possible run of the program, a certain shared resource is always executed by at most one task at a time. For such resources the according locks could be removed. At the time this thesis was written, however, CBMC was not ready to be used yet for multi-threading. In this regard the applicability of CBMC for optimizations has to be left to future research.

Apart from optimization concerns, the scenarios presented in this work showed that for instance the implementation of shared resources as C structures may entail the generation of many according declarations. Without concrete scenarios from the industry it remains uncertain whether the amount of generated code has a negative impact on the applicability of ParallelMbeddr. Nevertheless, future research should investigate the impact of padding that may be inserted into the generated structs and whether according optimization should be pursued. The scenarios further showed that general paral-

[71] For instance, if loops and recursive functions need to be evaluated more often than the user allows, they will limit the assertions that CBMC can verify.

lelization problems, like the dining philosophers problem and map-reduce via communication queues, can be implemented with ParallelMbeddr. The performance and scalability of the resulting code does not only depend on the quality of performed optimizations, but also on the general utilization of synchronization. For instance, the pi-example of section (TODO) showed that the implementation of a queue and communication via the queue require a busy-wait approach on the application level. Although concepts like message passing can therefore in a sense also be implemented by the user, it is questionable if the performance and usability of such an implementation have already reached their limits. Therefore, native support for concepts like message-passing or condition variables could both improve the usability and may, in certain cases, enhance the performance of the written code.
Overall, the thesis showed, how first-class language support on top an existing language can both ease the writing of parallel data-race free code and use the compiler for optimizing the resulting code in terms of synchronization overhead. To the knowledge of the writer, it also gave the first implementation of such an approach for C in a real-world setting, the development environment mbeddr. While this work laid the foundation for parallelization support for mbeddr, it showed how much potential reside in the chosen approach and how further enhancements can raise the accomplishments. From an industrial point of view, one main task for the future will be the complete integration of ParallelMbeddr with all of mbeddr's various language concepts and the stabilization for real-world applications.

Bibliography

[1] Cbmc: Bounded model checking for c/c++ and java. `http://www.cprover.org/cprover-manual/cbmc.shtml`. Accessed: 2014-08-24.

[2] mbeddr, engineering the future of embedded software. `http://mbeddr.com/`. Accessed: 2014-08-24.

[3] *OpenMP Application Program Interface,*, version 4.0 edition.

[4] Rust documentation: Struct sync::mutex. `http://doc.rust-lang.org/sync/struct.Mutex.html`. Accessed: 2014-08-24.

[5] The Open Group pthread_mutex_destroy, pthread_mutex_init - destroy and initialize a mutex. `http://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_mutex_destroy.html`. Accessed: 2014-07-15.

[6] Leveraging the benefits of symmetric multiprocessing (smp) in mobile devices. Texas Instruments, 2009.

[7] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static analyses for eliminating unnecessary synchronization from java programs. In *Static Analysis*, pages 19–38. Springer, 1999.

[8] A. Alexandrescu. *The D Programming Language*. Pearson Education, 2010.

[9] Andrei Alexandrescu. Concurrency in the d programming language. `http://www.informit.com/articles/article.aspx?p=1609144&seqNum=15`. Published: 2010-07-06.

[10] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[11] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[12] David Basin, Samuel J Burri, and Günter Karjoth. Obstruction-free authorization enforcement: Aligning security with business objectives. In *Computer Security Foundations Symposium (CSF), 2011 IEEE 24th*, pages 99–113. IEEE, 2011.

[13] L.W. Baugh and University of Illinois at Urbana-Champaign. *Transactional Programmability and Performance*. University of Illinois at Urbana-Champaign, 2008.

[14] Kevin Bierhoff and Jonathan Aldrich. *Modular typestate checking of aliased objects*, volume 42. ACM, 2007.

[15] Jeroen Borgers. *Do Java 6 threading optimizations actually work?* Published: 2008-06-18.

[16] T.D. Brown. *C for BASIC Programmers*. Silicon Press, 1987.

[17] D.R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley professional computing series. Addison-Wesley, 1997.

[18] G.C. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency: Predictability vs. Efficiency*. Series in Computer Science. Springer, 2006.

[19] D.E. Culler, J.P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/software Approach*. The Morgan Kaufmann Series in Computer Architecture and Design Series. Morgan Kaufmann Publishers, 1999.

[20] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[21] D.M. Dhamdhere. *Operating Systems: A Concept-based Approach,2E*. McGraw-Hill Higher Education, 2006.

[22] E.H. D'Hollander, G.R. Joubert, F. Peters, and U. Trottenberg. *Parallel Computing: Fundamentals, Applications and New Directions: Fundamentals, Applications and New Directions*. Advances in Parallel Computing. Elsevier Science, 1998.

[23] I.A. Dhotre. *Operating Systems*. Technical Publications, 2007.

[24] Tomas Evensen. Multicore challenges and choices: Deciding which solution is right for you. `http://leadwise.mediadroit.com/files/8537WP_Deciding_which_multicore_solution_is_right_June2009.pdf`. Published: 2009.

[25] Martin Fowler. Language workbenches: The killer-app for domain specific languages? `http://www.martinfowler.com/articles/languageWorkbench.html`. Published: 2005-06-12.

[26] F. Franek and F. Franěk. *Memory as a Programming Concept in C and C++*. Cambridge University Press, 2004.

[27] V.K. Garg. *Concurrent and Distributed Computing in Java*. Wiley, 2005.

[28] J.M. Garrido, R. Schlesinger, and K. Hoganson. *Principles of Modern Operating Systems*. Jones & Bartlett Learning, 2011.

[29] Brian Goetz. *Java theory and practice: Synchronization optimizations in Mustang*. Published: 2005-10-18.

[30] R. Guerraoui, M. Kapałka, and N. Lynch. *Principles of Transactional Memory*. Synthesis lectures on distributed computing theory. Morgan & Claypool Publishers, 2010.

[31] S. Haldar and A.A. Aravind. *Operating Systems*. Pearson Education, 2009.

[32] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985.

[33] T. Harris, J.R. Larus, and R. Rajwar. *Transactional Memory*. Synthesis lectures in computer architecture. Morgan & Claypool, 2010.

[34] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[35] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[36] Lorin Hochstein, Jeffrey Carver, Forrest Shull, Sima Asgari, Victor Basili, Jeffrey K Hollingsworth, and Marvin V Zelkowitz. Parallel programmer productivity: A case study of novice parallel programmers. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 35–35. IEEE, 2005.

[37] A. Holub. *Taming Java Threads*. Apresspod Series. Apress, 2000.

[38] Joel Hruska. The death of cpu scaling: From one core to many - and why we're still stuck. `http://www.extremetech.com/computing/116561-the-death-of-cpu-scaling-from-one-core-to-many-and-why-were-still-stuck`. Published: 2012-02-01.

[39] Kun Huang, Gaogang Xie, Rui Li, and Shuai Xiong. Fast and deterministic hash table lookup using discriminative bloom filters. *Journal of Network and Computer Applications*, 36(2):657–666, 2013.

[40] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *ACM SIGPLAN Notices*, volume 37, pages 130–141. ACM, 2002.

[41] B. Lewis and D.J. Berg. *Multithreaded Programming with Java Technology*. Sun Microsystems Press Java series. Prentice Hall, 2000.

[42] C.S. LLC, M.L. Mitchell, A. Samuel, and J. Oldham. *Advanced Linux Programming*. Pearson Education, 2001.

[43] Jason Loew, Jesse Elwell, Dmitry Ponomarev, and Patrick H Madden. Mathematical limits of parallel computation for embedded systems. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, pages 653–660. IEEE Press, 2011.

[44] R. Love. *Linux System Programming: Talking Directly to the Kernel and C Library*. O'Reilly Media, 2013.

[45] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003.

[46] Robin Milner. The polyadic pi-calculus: a tutorial. Technical report, Logic and Algebra of Specification, 1991.

[47] A. Mittal. *Programming In C: A Practical Approach*. Pearson Education, 2010.

[48] Robert H. B. Netzer, Timothy W. Brennan, and Suresh K. Damodaran-Kamal. Debugging race conditions in message-passing programs. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '96, pages 31–40, New York, NY, USA, 1996. ACM.

[49] P. Pacheco. *An Introduction to Parallel Programming*. An Introduction to Parallel Programming. Elsevier Science, 2011.

[50] Bhatt P.C.P. *Introduction To Operating Systems: Concepts And Practice An 2Nd Ed.*

[51] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[52] S. Prata. *C Primer Plus*. Pearson Education, 2013.

[53] J. Protic, M. Tomasevic, and V. Milutinović. *Distributed Shared Memory: Concepts and Systems*. Systems Series. Wiley, 1998.

[54] Ravi Rajwar and James R Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305. IEEE Computer Society, 2001.

[55] R. Reese. *Understanding and Using C Pointers*. O'Reilly Media, 2013.

[56] Amitabha Roy, Steven Hand, and Tim Harris. A runtime system for software lock elision. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 261–274. ACM, 2009.

[57] Radu Rugina and Martin C Rinard. Pointer analysis for structured parallel programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(1):70–116, 2003.

[58] William N Scherer III and Michael L Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248. ACM, 2005.

[59] J. Shirazi. *Java Performance Tuning*. Java Series. O'Reilly Media, Incorporated, 2003.

[60] Adam R Smith and Prasad A Kulkarni. Localizing globals and statics to make c programs threadsafe. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 205–214. ACM, 2011.

[61] Konstantin Solomatov and Vaclav Pech. JetBRAINS generator user guide. `http://confluence.jetbrains.com/display/MPSD30/Generator`. Accessed: 2014-07-18.

[62] C. Steven Hernandez. *Official (ISC)2 Guide to the CISSP CBK, Second Edition*. (ISC)2 Press. Taylor & Francis, 2009.

[63] Sven Stork, Paulo Marques, and Jonathan Aldrich. Concurrency by default: using permissions to express dataflow in stateful programs. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 933–940. ACM, 2009.

[64] S. Tucker Taft. *ParaSail Reference Manual - Draft*.

[65] Tucker Taft. Parasail: Less is more with multicore. `http://www.embedded.com/design/programming-languages-and-tools/4375616/1/ParaSail--Less-is-more-with-multicore`. Published: 2012-06-19.

[66] Tucker Taft. A pointer-free path to object-oriented parallel programming. `http://parasail-programming-language.blogspot.de/2012/08/a-pointer-free-path-to-object-oriented.html`. Published: 2012-08-01.

[67] Markus Völter. Generic tools, specific languages.

[68] John Whaley and Monica S Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Notices*, volume 39, pages 131–144. ACM, 2004.

[69] E. White. *Making Embedded Systems: Design Patterns for Great Software*. O'Reilly Media, 2011.