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

- mbeddr is a language and ide for embedded programming that facilitates programming by providing higher level language mechanisms
- parallel programming is becoming ever more important
- yet, mbeddr is still lacking higher-level support for parallel programming
- compared to library extensions language based support for pp features multiple benefits: individual problem-specific syntax, reduction of erroneous input by type system enhancements, program-wide optimization of generated code
- short examples for benefits
- ParallelMbeddr introduces task-based explicit parallel programming with shared memory that can be synchronized
- Objectives: apropriate syntax, type system support for safer code, optimized output
- outline of the work
- andeutung der ergebnisse (bezug nehmen in conclusion)

Kommentar von Bastian
Question: Introduce example that is later used for the evaluation already here for motivational reasons?

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* [33, p. 61]. The law gives an upper bound on the possible speed up of an algorithm based on the amount of parallelizeable code.[1] 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 [17, 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" [26, 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" [11, 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 [9, 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

[1] 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" [9, 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" [12, p. 327].[2] 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 [19, p. 138].[3] Data races can be avoided by employing the low-level synchronization primitive called *mutex*.[4] 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 [27]. 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 [16, p. 42] and synchronized classes in the programming language *D* [2] were developed. These measures are usually built on top of mutexes [24, 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" [35, p. 119] thread. As a result, no thread will ever finish executing [13, p. 2-3].

[2] 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 [47]. Since this work is about parallelization of single processes, other kinds of race conditions are not considered further.

[3] 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.

[4] 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) [23]. In CSP, messages are sent from one entity to another. "The sender waits until the receiver has accepted the message (*synchronous* message passing)" [38, p. 138] before it continues its execution. Message passing with asynchronous message sends were deployed by the actor model [22] and pi calculus [30]. Although message passing avoids shared data and realizes communication generally via copies of data,[5] it still suffers from potential race conditions [32] when the atomicity of operations is not properly regarded by the implementation.

Communication Model: Transactional Memory

Transactional memory provides a non-blocking[6] memory model, which enables communication via "lightweight, *in-memory* transactions" [18, 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 [21].[7] 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" [5, 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 [18, 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

[5] 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.

[6] TODO: explain

[7] 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." [49, 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 [49]. 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 [8, 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.[8]

MPS and Mbeddr

TODO: Visualisieren

"JetBrains MPS[9] is an open source [...] language workbench developed over the last ten years by JetBrains. " [48] As such it provides a projectional editor which lets the user work directly on the abstract syntax tree (AST) of the program [14]. 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.[10] 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*

[8] 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) [8, 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.

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

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

"defines the denotational semantics for the concepts in the language" [46]. 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.[11] 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.[12] 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 [6, p. 48].

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

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

[12] 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.[13] 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 safe 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 [36], the syntax of new concepts is described by notations like the following:

```
1 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.

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

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* 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:

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

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.[14] Every task in ParallelMbeddr is represented by a thread as provided by the POSIX threads standard, a so-called *pthread*.[15] 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:

```
1 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.[16] 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:

```
1 struct Args_X {
2   t_1 v_1;
3   ...
4   t_n v_n;
5 }
```

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:

```
1 void* parFun_X(void* voidArgs) {
2   t* result = malloc(sizeof(t));
3   Args_X* args = (Args_X*) voidArgs;
4   *result = e';
5   return result;
6 }
```

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:

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

[14] http://openmp.org/

[15] 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.

[16] 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.[17] 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_i** to **v_n**, which need to be bound, becomes an mbeddr block expression:[18]

```
{
  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.[19]
Without any references to bind, a task is reduced to the compound literal:

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

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

[19] 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*.[20] 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:

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

```
#constant RANGESIZE = 300000000;
#constant RANGECOUNT = 4;
#constant THRESHOLD = RANGESIZE * RANGECOUNT;
```

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

```
int32 main(int32 argc, string[] argv) {
...
  Task<long double*>[RANGECOUNT] calculators;
  for (i ++ in [0..RANGECOUNT[) {
    uint32 start = i * RANGESIZE;
    uint32 end = start + RANGESIZE;
    calculators[i] = |calcPiRange(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[21] to the building blocks that were introduced in 3.1.2:

```

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

```

[21] 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 other minor adjustments.

```
25 void* parFun_0(void* voidArgs) {
26   long double* result = malloc(sizeof(long double));
27   Args_0* args = ((Args_0*) voidArgs);
28   *result = calcPiRange((args)->start, (args)->end);
29   free(voidArgs);
30   return result;
31 }
```

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 **|calcPiRange(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 **calcPiRange(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 [20]. 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:[22]

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:

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

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

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[23] with the initialized thread handle:[24]

```
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

[23] 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.

[24] 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 37.

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 can not be guaranteed in such a case, a dangling pointer [40] 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 [37, 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[25] 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:

```
1  int32 main(int32 argc, string[] argv) {
2    Task[RANGECOUNT] calculators;
3    Future[RANGECOUNT] partialResults;
4
5    ... // task declarations
6
7    for (int8 __i = 0; __i < RANGECOUNT; __i++) {
8      partialResults[__i] = runTaskAndGetFuture(calculators[__i]);
9      free (calculators[__i].args);
10   }
11
12   for (int8 __i = 0; __i < RANGECOUNT; __i++) {
13     result += *(((long double*) getFutureResult(&partialResults[__i])));
14     free((long double*) getFutureResult(&partialResults[__i]));
15   }
```

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

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

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.[26]

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 way.

3.3.1 Design

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

$t ::= ... | \mathit{shared}\langle u \rangle \mid \mathit{shared}\langle \mathit{shared}\langle u \rangle * \rangle$

$u ::= t \qquad \mathit{void} \neq u \neq t*$

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 the SAFETY CHAPTER, the base type may not denote a pointer to a value that is not shared.[27] 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`**:

$e ::= e.\mathit{get} \mid e.\mathit{set}(e)$

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

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`**.

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

[27] Differently said: 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.[28] 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.:

```
1 shared<shared<int32>> v;
2 // vContent is declared before it is used in the list => valid
3 sync(v, &(v.get) as vContent, &vContent->get as vContentContent) {
4   vContentContent->set(0);
5 }
6 // vContentContent is not in scope, here => invalid
7 vContentContent->set(1);
```

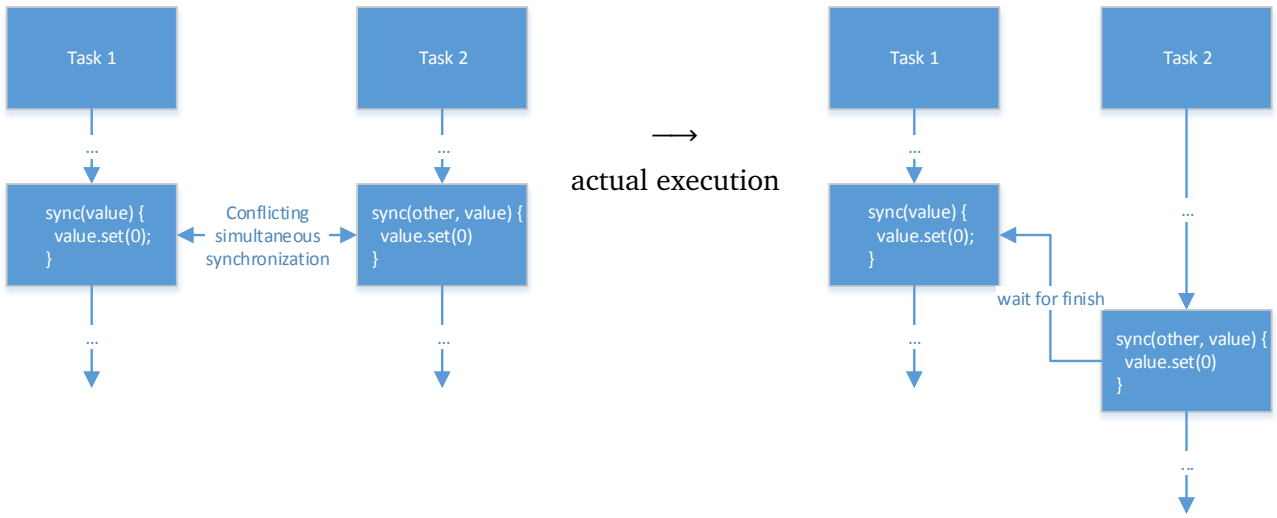
This restrictions results from the lexical scoping of synchronization resources: In order to determine whether the target of **.get** or **.set** is synchronized ParallelMbeddr checks whether the target is a reference to a visible and synchronized variable or to a named resource. This implies that shared 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 [45, 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.

[28] Actually, due to eager evaluation 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**:

```
1 shared<int32> value;
2 shared<int32>* valuePointer = &value;
3 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:

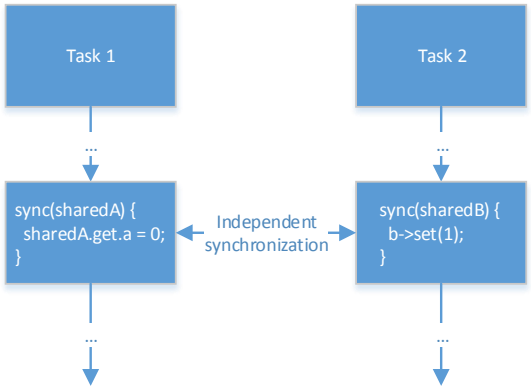


The possibility to refer to multiple shared resources in the synchronization list of a synchronization statement is not mere syntactic sugar for nested synchronization statements. Instead, the semantics of a synchronization list are that all referenced shared resources are synchronized at once, but with a possible time delay. Due to the design of the underlying implementation, deadlocks, as a result of competing synchronization statements, are thus avoided.[29]

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**:

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

```
1 struct A {
2   int32 a;
3   shared<int32> b;
4 }
5 shared<A> sharedA;
6 shared<int32>* sharedB;
7 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*, [30] provides comparatively fast access to its members but imposes additional memory requirements in comparison to slower hash map

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

implementations[31]. A disadvantage of hash maps is the non-deterministic performance[32] and increased access time that may be induced by hash collisions [25].

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 plus additional padding [28, 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:

```
1 struct SharedOf_t {
2   pthread_mutex_t mutex;
3   t' value;
4 }
```

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.[33] 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,[34] e.g:

[31] 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.

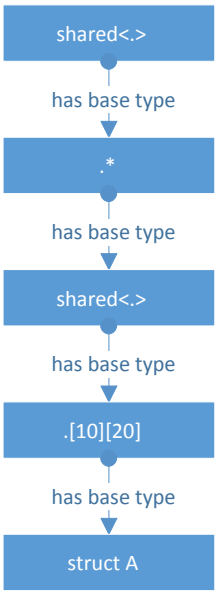
[32] Although real-time applications with their time-wise constraints are not a primary concern of this work, it should nevertheless be kept in mind that they are part of the embedded domain. Therefore, a solution that considers the characteristics of this domain is at least regarded advantageous.

[33] 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.

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

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

⟶ 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:

```
1 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:

```
1 e'.value
```
respectively
```
1 e'.value = f'
```

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

```
1 e'->value
```
respectively
```
1 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:

```
1  // for a proper shared type shared<t> and the type t' that t is reduced to
2  void mutexInit_X(SharedOf_t'* var) {
3    pthread_mutex_init(&var->mutex, &mutexAttribute);
4
5    // either if t is a shared type or a struct type:
6    mutexInit_X'(&var->value);
7
8    // or if t is an array type t[i_1]...[i_n] of 1 to n dimensions:
9    mutexInit_X'((SharedOf_t'*...*)var->value, i_1, ..., i_n);
10 }
```

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.[35] 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).

```
1  // for a proper array type t[]...[] of 1 to n dimensions where '...' denotes the occurrence of an accor
2  // and t' denotes the reduced-to type
3  void mutexInit_X(t'*...* var, int32 size_0, ..., int32 size_n) {
4    for (int32 __i_0 = 0; __i_0 < size_0; __i_0++) {
5      ...
6        for (int32 __i_n = 0; __i_n < size_n; __i_n++) {
7          // in case t is a struct type
8          mutexInit_X'(&var[__i_0]...[__i_n]);
9
```

[35] 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.[36] 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.[37] 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[38] 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,[39]

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.

[36] 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.

[37] 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 [1]. 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.

[38] In other words, only those control-flow-breaking statements *c*s that are either one of the statements *stmt*s 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.

[39] 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.:

```
1 // simple shared resource
2 shared<int32> v1;
3
4 // complex shared resource
5 shared<int32>[2][3] v2;
```

$\Longrightarrow$

```
1 SharedOf_int32_0 v1;
2 pthread_mutex_init(&v1.mutex, &mutexAttribute);
3
4 SharedOf_int32_0[2][3] v2;
5 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:

```
1 shared<int32> sharedValue;
2 sync(sharedValue) {
3   sync(sharedValue) {
4     sharedValue.set(42);
5   }
6 }
```

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:[40] 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:

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

[40] 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

```
1 sync(e) stmt_list
```

$\Longrightarrow$

```
1 startSyncFor1Mutex(&e.mutex);
2 stmt_list'
3 stopSyncFor1Mutex(&e.mutex);
```

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

```
1 sync(e_1, ..., e_n) stmt_list
```

$\Longrightarrow$

```
1 startSyncForNMutexes(&e_1.mutex, ..., &e_n.mutex);
2 stmt_list'
3 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*.[41]

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 [1]. 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:

[41] 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[42] 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.[43]

```
// 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) |];
```

[42] Busy-waiting means that the thread will repeatedly test a condition until it is met, without doing actual useful work [34, 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 [7, 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 [4]. 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/.

[43] 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 [44] 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.[44] Thus, maximum delays of about 100 ms (specifically 131 ms) are allowed.[45]

```
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 [29]:

```
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)

[44] 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**.

[45] 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 [10]). 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:

```
1 struct Queue {...}
2 void queueInit(shared<Queue>* queue);
3 void queueSafeAdd(shared<Queue>* queue, long double item);
4 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:

```
1 #constant RANGESIZE = 300000000;
2 #constant RANGECOUNT = 4;
3 #constant THRESHOLD = RANGESIZE * RANGECOUNT;
4 #constant MAPPERCOUNT = 2;
```

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

```
1 exported int32 main(int32 argc, string[] argv) {
2   shared<Queue> queue;
3   queueInit(&queue);
4   shared<Queue>* queuePointer = &queue;
5
6   shared<uint32> counter;
7   shared<uint32>* counterPointer = &counter;
8   sync(counter) { counter.set(0); }
9
```

```
  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[46] 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;

```

[46] **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 + RANGESIZE;
      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.[47] 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

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

TODO: maybe fix this

:

```
1 mutexDestroy_2(&queue);
2 pthread_mutex_destroy(&counter.mutex);
3 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.[48] 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:

```
1 VoidFuture futureInit_0(SharedOf_long_double_0* resultPointer, SharedOf_Queue_0* queuePointer) {
2   Args_1* args_1 = malloc(sizeof(Args_1));
3   args_1->resultPointer = resultPointer;
4   args_1->queuePointer = queuePointer;
5   pthread_t pth;
6   pthread_create (&pth, null, :parFun_1, args_1) |];
7   return (VoidFuture){ .pth = pth };
8 }
```

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>**.

[48] 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 + RANGESIZE;
    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" [31, 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.

```
1  // global variables:
2  shared<int32*> v1;
3  shared<int32*> v2;
4
5  int32 main(int32 argc, string[] argv) {
6    int32 sharedValue;
7    sync(v1, v2) {
8      v1.set(&sharedValue);
9      v2.set(&sharedValue);
10   }
11   |foo()|.run;
12   |bar()|.run;
13 }
14
15 foo() {
16   sync(v1) { *v1.get = 0; }
17 }
18 bar() {
19   sync(v2) { *v2.get = 1; }
20 }
```

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:

```
1 shared<shared<int32>> v;
2 sync(v) {
3   sync(v.get) {
4     // error: e.get seems to be unsynchronized
5     e.get.set(0);
6   }
7 }
```

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 introduced 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 it may induce unnecessary serialization during program execution if potential data hazards will actually never manifest at runtime[39]. 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 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 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 in case shared resources are never recursively synchronized. If such cases can be 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 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 [15, p. 27]. The amount of padding that is inserted depends on the difference of the byte sizes of the individual fields and the order of these fields [15]. Since the “art of C structure packing”[49] is no trivial task and space optimization is no primary concern of this paper, according work is left open for future research.

4.2 Time Optimizations

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 main dimensions. First, the amount of synchronization management (TODO), i.e. the time spent for acquiring and releasing locks, can be reduced by diminishing the number of performed

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

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 [39][41]. Due to the complexity and overhead of such techniques this thesis focuses on compile-time optimizations.

4.2.1 Basics

The applicable measures for optimizations distinguish 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*.

Pointer Analysis

Pointer analysis, also known as *points-to analysis*, tries to find the set of memory locations to which a pointer may point[43]. This information can be used. In the following example 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. The precision depends on two properties: whether the analysis is *flow-sensitive* or *-insensitive* and whether it is *inter-procedural* or *intra-procedural* [3]. The first property distinguishes, whether the particular flow of data and with it the order of statements is taken 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 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 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**.

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

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

Currently, mbeddr does not provide any form of pointer analysis.

Data-flow Graph

- pointer analysis, kinds of: inter/intra, may/must

4.2.2 Opportunities

Static analysis offers a variety of optimization opportunities. These distinguish 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 ideas were (mainly) borrowed from the according literature. The opportunities are ordered in the way they are applied in ParallelMbeddr. The first three 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, where 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 out 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) {
```

```
15    sync(var as varToGet) {
16      int32 val = varToGet->get;
17    }
18  }
```

Whereas **varToSet** in **init()** need not be synchronized since it is not yet shared with another task, the according **varToGet** obviously needs to be synchronized inside **task()**. Furthermore, shared resources may only be accessed by one task, at all. 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 still has to annotate the synchronization in line 11. This way the code need not be changed in case **QueueContainer** might eventually be equipped with non-shared data.[50] Nevertheless, the compiler should take care of eliminating locks for such data. In case

[50] Of course, this

functions are called with both read-only shared resources and written shared resources, the necessity of their according synchronization resources 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:

```
1  void sort(shared<int32[1000]>* array, int32 start, int32 end)
2  int32 middle = ...
3  sort(array, start, middle);
4  sort(array, middle + 1, end);
5  sync(array) {
6    // merge the sorted sub arrays
7  }
```

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:

```
1  void calculate(shared<double>* result) {
2    sync(result as myResult) {
3      // do something that is expensive and unrelated to the argument
4      double pi = calculatePi();
5      // now use myArg
6      myResult->set(pi);
7      // again, something unrelated
8      log(pi);
9    }
10 }
11
```

```
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 the synchronization context:

```
1 void calculate(shared<double>* result) {
2   double pi = calculatePi();
3   sync(result as myResult) { myResult->set(pi); }
4   log(pi);
5 }
```

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:

```
1 void increment(shared<int32>* c) {
2   int32 current, next;
3   sync(c as myC) {
4     current = myC->get;
5     next = current + 1;
6     myC->set(next);
7   }
8 }
```

$\longrightarrow$ split

```
1 void increment(shared<int32>* c) {
2   int32 current, next;
3   sync(c as myC) { current = myC->get; }
4
5   next = current + 1;
6
7   sync(c as myC) { myC->set(next); }
8 }
```

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 leverage points. Due to the current lack of a pointer-analysis analysis in mbeddr, at first a simplified pointer analysis was conceived, in order to be able to apply the optimization algorithms.

Pointer analysis

The analysis that was implemented in the course of this thesis makes several simplifications in order to fit into the scope of this work. These will be depicted later. As a starting point, a simplified directed data-flow graph is constructed. The nodes of this graph consist of local variable declarations, arguments[51] and references to either kind. For each reference r to a local variable or an argument x an arc (x, r)

[51] 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.

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.[52] 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(g) **do**
    ADD(a, (v, v)) ▷ a is the new alias graph
  end for
  repeat
    find some $(n, m) \leftarrow$ ARCS(g) where not $a[n]$ contains 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 argument and there are $i, j \in g[n]$ where $a[i] \neq a[j]$ **then**
      skip (n, m)
    else
      $a[n]$.ADD ALL($a[m]$)
    end if
  until no changes made
end function

aliases propagates alias information through the alias graph as long as any changes are possible. It repeatedly tries to find a node n who 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 with the missing aliases. If must-aliases need to be calculated (which is 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 argument value-to-argument arcs, since no other branchings are considered.

TODO: Check if sideward propagation can be removed in the code of alias()

TODO: Change dataflowgraph maker to include cross task boundaries for readonly-analysis

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

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 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**
      $\overline{c}$.ADD(v) ▷ $\overline{c}$ = 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 if 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($\overline{c}$, a, d) ▷ $\overline{c}$ = clean variables, a = aliases, d = additional data
  for all $s \leftarrow$ SYNC RESOURCES(data) **do**
    if $\overline{c}$ contains all $a[s.expr]$ **then** ▷ $s.expr$ evaluates to a shared resources or pointer thereof
      s.REMOVE
    else
      $f\_to\_\bar{s}[function\ ofs]$.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 INLINE(f, $\bar{s}$, $\overline{c}$, a, d)
  end for
end function

Every synchronization resource s is either directly removed or deferred. In case 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 in the following example:

```
1 void shareXButNotY() {
2   shared<int32> x;
3   shared<int32>* xP = &x;
4   shared<int32> y;
5   |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.[53] In this case the function *Try To Inline* 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 it is done by *Try To Inline*, the function can be copied and accordingly optimized:

```
  //...
  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
}
```

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 copying) 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 are considered whose expressions directly refer to one of the arguments of the surrounding function (like in the previous example). The *Try To Inline* function works as follows:

▷ f = function, $\overline{ds}$ = partially clean sync resources, $\overline{c}$ = clean variables, a = aliases, d = add. data
function TRY TO INLINE(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

[53] This merge is caused by the inter-procedural property of the current alias analysis. In case of an intra-procedural pointer analysis the following work would be facilitated.

```
    end for
    for all l, cs ← l_to_cs do                    ▷ pack the calls by equal sets of clean sync resources
        cs_to_l[cs].ADD(l)
    end for
    for all cs, l ← cs_to_l do                    ▷ copy the function for calls of equal sets of clean sync resources
        COPY FUNCTION(f, cs, l)
    end for
end function
```

Try To Inline 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* mine 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 in order to cluster calls which have equal sets of clean synchronization resources. These clusters are then used by *Copy Function* to generate optimized versions of the current functions. For the lack of valuable insight, the definitions of *Clean Calls For* and *Copy 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 ← d.variables do
        if ∃ e.set(_) in d.sharedSets where a[e] contains v then
            skip v
        else if ∃ e.get in d.sharedGets where a[e] contains v and ∃ f = _ where f contains e then
            skip v
        end if
        c.ADD(v)                                   ▷ c = readonly (clean) task variables
    end for
    REMOVE CLEAN LOCKS(c, a, d)
end function
```

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 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 due to 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 **recursive()** could otherwise detect that **value1 value2** are already synchronized at the beginning of the function.

```
1 void recursive(shared<int32>* value1, shared<int32>* value2) {
2   sync(value2 as myValue2) {
3     myValue2.set(2);
4   }
5   sync(value1 as myValue1) {
6     myValue1.set(5);
7     recursive(myValue1, myValue1);
8   }
9 }
```

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 context should be removed (or not added in the first place). This way, false recursive locks across tasks are omitted:

```
1 void task1(shared<int32>* value) {
2   sync(value as myValue) { value.set(1);
3     |task2(value)|.run                    // here, value is not synced anymore
4   }
5 }
6
7 void task2(shared<int32>* value) {
8   sync(value as myValue) { value.set(2); }
9 }
```

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 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 this shared resource would not have to synchronize it any more.

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

With a data-flow graph that was created by following these conditions, the alias graph can be calculated. In the following course of the work two algorithms for the recursive-lock removal are presented.[54] 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), i = inverse 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 $a[s.expr] = a[r]$ **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)
      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.

[54] The former algorithm was implemented for its simplicity in the 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>**).

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 path 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. Otherwise on every path from f to *main* there must each 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.

▷ 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 $s \leftarrow d.syncResources$ **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 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

5 Evaluation

- show if and how objectives mentioned in the introduction are met by the implementation
- use ilustrating examples

6 Evaluation

- show if and how objectives mentioned in the introduction are met by the implementation
- use ilustrating examples

Bibliography

[1] 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.

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

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

[4] 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.

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

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

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

[8] 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.

[9] 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.

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

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

[12] 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.

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

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

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

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

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

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

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

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

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

[22] 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.

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

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

[25] 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.

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

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

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

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

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

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

[32] 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.

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

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

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

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

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

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

[39] 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.

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

[41] 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.

[42] Erik Ruf. Context-insensitive alias analysis reconsidered. *ACM SIGPLAN Notices*, 30(6):13–22, 1995.

[43] 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.

[44] 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.

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

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

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

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

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