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:

Inhaltsverzeichnis

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

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

2 Background

2.1 Processes and Threads

Every program in (delayed or interrupted) execution 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[**?**, 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.

2.2 Parallelism and concurrency

If multiple threads are "in the middle of executing code at the same time"[18, p. 124(?)] they are processed concurrently. They can actually executed at the same time on different processors or interleaved on a single processor. The former is further called parallelism. Parallelism generalized to "the quality of occuring at the same time"[9, p. 91] can manifest in different ways. TODO: consider mentioning data races that are involved with both execution models

2.3 Types of parallelism

At least four different kinds of parallelism have been conceived. From an application programmer's view seen at a very low level of the computer exist bit-level parallelism and instruction-level parallelism. 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[8, p. 15]. Instruction-level parallelism, also called pipelining, is the simultaneous utilization of multiple stages of the execution pipeline of the processor. Both bit-level parallelism and instruction-level parallelism primarily reside on the hardware level and the operating system level and are, thus, no subject of this work. Data parallelism is present if the same calculation is performed on multiple sets of data and can be regarded as a specialization of task parallelism which denotes the simultaneous execution of different calculations "on either the same or different data"[8, p. 125]. The latter being the more general approach to software-level parallelism is the subject of this work.

2.4 Data races

With multiple threads running in parallel and having access to the shared data of their process a second class of errors that is unique to parallel (and distruted) programming arises. These so-called synchronisation 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 unsynchronised memory references by two processes on one memory location, of which at least one reference is a write access"[10, p. 327][1]. 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.

[1] As the definition implies data races are not limited to threads and the shared process data. E.g. file-based race conditions can even occur between two different processes[29]. Since this work is about parallelization of single processes other kinds of race conditions are not further considered.

2.5 Communication model: shared memory

The memory model that underlied the former treatment of processes that 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[13, p. 138][2]. Data races can be avoided with help of the low-level the synchronization primitive *mutex*[3]. A mutex can be locked by exactly one thread. Any other thread that tries to lock the same mutex is blocked until the locking thread releases the mutex[19]. 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(TODO: forward reference) and synchronized classes in D(TODO: forward reference) were developed. These measures are usually built on top of mutexes[17, p. 25]. Another disadvantage of mutexes is their vulnerability for deadlocks which means that multiple processes are in a state where "each is waiting for release of a resource which currently held by some other process"[2, p. 119] such that no progress will ever finish executing[11, p. 2-3].

2.6 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)(TODO: reference!). In CSP messages are sent from one entity to another. "The sender waits until the receiver has accepted the message (*synchronous* message passing)"[26, p. 138]. Message passing with asynchronous message sents were deployed by the actor model[16] and pi calculus[22]. Although message passing avoids shared data and realizes communication generally via copies of data[4] it still suffers from potential race conditions[23].

2.7 Communication model: transactional memory

Transactional memory provides a non-blocking[5] memory model which enables communication via "lightweight, *in-memory* transactions"[12, p. 3] which 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[6][15]. 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"[4, p. 13].

2.8 Coarse- and fine-grained synchronization

In order to keep structures and computations synchronized the simplest approach to avoid data races is to use the available measures 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[12, pp. 1-2].

[2] As for race conditions the model is not limited to intra-process communication via threads or similar approaches to parallelization. Communication between two process can also be realized via shared memory but is not in the scope of this work.

[3] Semaphores as a second synchronization primitive are closely related to mutexes. Since they do not provide further insights for the discussion they are not further investigated in this work.

[4] 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 like in terms of monitors that were developed by Hoare to reduce deadlocks in CSP. The main notion of the message passing concept nevertheless goes without shared data.

[5] TODO: explain

[6] To this end the corresponding transactions may need to be reexecuted as a whole.

2.9 Embedded programming

"An embedded system is a computerized system that is purpose-built for its application."[31, 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 just accomplishs the job[31]. Thus, 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 [7, pp. 1-2]. Therefore the predictability of the program's execution time becomes an issue for real-time systems[7].

2.10 MPS and mbeddr

"JetBrains MPS[8] is an open source [...] language workbench developed over the last ten years by JetBrains. "[30] As such it provides an projectional editor which lets the user directly work on the abstract syntax tree (AST) of the program[1]. It supports the development and composition of potentially syntactically ambiguous modular language extensions in combination with the development of integrated development environments 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 be further processed by the gcc tool chain[9]. The implementation of C in mbeddr does not only provide extensions to the core of C but also has a few differences to the basis of C99. They will be introduced as needed.

2.11 C

The semantics of C differ from modern object-oriented languages like Java in a variety of ways. In order to clarify some of the choices that were made for the design and implementation of ParallelMbeddr the most relevant differences shall be outlined. Like Java C leverages pass-by-value semantics for function parameters. Differently though it generally copies the referenced values into the memory that is allocated for function calls whereas Java copies the references themselves. Thus a change to a field of a struct instance that was copied in such a way does not affect the original struct instance[10]. On the other hand arrays are treated like pointers 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. In order to have an array be copied into a function 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 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 on the area of the stack that was allocated for this function[11]. A peculiarity of C is that global variables may only be initialized with constant expressions such that it is not possible to initialize a global variable with an arbitrary function call of a proper type[5, p. 48].

TODO: Add explanations for MPS aspects, type-system checking rules... as needed

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

[7] Mbeddr(TODO: forward reference) does not have first-class support for the quantification of related parameters like the worst-case execution time (WCET)[7, p. 8], yet. For this reason 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.

[8] http://jetbrains.com/mps

[9] http://gcc.gnu.org/

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

[11] Otherwise the 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.

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

3 Design and Implementation

In this chapter 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 the design and the translation to plain mbeddr C code[12]. 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. Whenever necessary relevant details of the implementation in mbeddr are briefly depicted.

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 this return value.

$t ::= ... | Task<t>$

Due to implementation reasons(TODO: ref) 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:

$e ::= ... | e.clear$

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

If a task is copied by the pass-by-value semantics of C the copied task will share the heap-managed data, 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. Keep in mind that a running instance of a task will not be affected by clearing the task by that it was created.

3.1.2 Translation

The pthreads library 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 as opposed to frameworks like OpenMP (TODO: reference). Every task in ParalllelMbeddr is represented by a thread as provided by the POSIX threads standard. 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:

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

```
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[13].

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

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

where every **v_i** represents an equally named reference in the task expression to a variable of type **t_i**.

The generated function **parFun_X** is then given an instance of **Args_X** which it uses to bind the references of the task expression to. The full function definition of a task **e** of type **Task<t*>** is, thus:

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

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

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

If the embraced expression of a task does not contain any reference of this kind (e.g. only references to global variables) the **args** definition line is omitted as is clearly the—otherwise empty—declaration of **struct Args_X**. In this case **e'** equals **e**.

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

```
void* parFun_X(void* voidArgs) {
  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 that captures both the initilization state and the computation of the embraced expression[14]. As the void pointer of the arguments does not keep their type information and therefore their size an additional field argsSize is needed in order to be able to create copies of the arguments later on(TODO forwared ref).

```
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 occurence 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 seperately 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** becomes an mbeddr block expression[15]:

---

[13] In the following explanations **X** will always denote some arbitrary suffix. Keep in mind that these suffixes do not necessarily coincide for different kinds of components.

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

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

```
{
  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

```
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[16].

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

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

By above definition of **`parFun_X`** it becomes clear that the output of a task is routed via the heap instead of the stack. Additionally 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. In exchange both the result and the arguments of a task have to be deleted by the programmer by hand. Since the result is returned via a pointer onto the heap it becomes obvious that the return type of a task must either be a void type or a pointer type. 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.

The clearance of a task **`e.clear`** in the generated code is 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 is chosen such as to be easy to understand in exchange for a realistic scenario. In this program for a range of numbers from 0 up to some threshold the cardinality of the set of factors is calculated in parallel:

```
#constant threshold = 40;
exported int32 main(int32 argc, string[] argv) {
  // every task calculates the factor count for one number
  Task<int32*>[threshold] tasks;
  for (int8 i = 0; i < threshold; ++i) {
    tasks[i] = |calcFactors(i + 2)|;
  } for
  // ...
  return 0;
} main (function)
```

The generation to C99 code yields the following intermediary helper code[17]:

```
// generic declarations module
exported struct Task {
  void* args;
  (void*)=>(void*) fun;
};
// user-defined module
struct Args_a0a0b0b { // note the arbitrary suffix
```

[16] Whereas in C for every struct type **`T`** a typedef has to be defined in order to reference this type directly with **`T`** instead of **`struct T`** in mbeddr this definition is done implicitly.

[17] Mbeddr specific details have been reduced in order to enhance the legibility of the code

```
  int8 i;
};
void* parFun_a0a0b0b(void* voidArgs) {
  int32* result = malloc(sizeof(int32));
  Args_a0a0b0b* args = ((Args_a0a0b0b*) voidArgs);
  *result = calcFactors((args)->i + 2);
  return result;
} parFun_a0a0b0b (function)
inline Task taskInit_a0a0b0b(int8 i) {
  Args_a0a0b0b* args_a0a0b0b = malloc(sizeof(Args_a0a0b0b));
  args_a0a0b0b->i = i;
  return (Task){ args_a0a0b0b , :parFun_a0a0b0b };
} taskInit_a0a0b0b (function)
```

The generated struct contains exactly one element for the referenced variable in the task expression. A new reference to this element replaces the corresponding reference in the expression embraced by the task when it is copied into the parallel function. The initialization function allocates memory on the heap and uses it to safe the value of the variable. The content of the main function simply becomes:

```
Task[threshold] tasks;
for (int8 i = 0; i < threshold; ++i) {
  tasks[i] = taskInit_a0a0b0b(i);
} for
```

TODO: Figure out argument deletion and write about it here

3.2 Futures

Whenever a task **t** is run, as will be shown, a *future* is generated. Futures in ParallelMbeddr are based on Halstead's definition of a future[14]. 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[18]:

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

As with tasks the type of a future parameterized by its return type:

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

e.run denotes the launch of task **e** whereas **e.result** joins a future **e** 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:

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

As will be shown in the next section a **result** returns a pointer to a heap-managed value. Hence the programmer has to take of freeing the value eventually.

3.2.2 Translation

A future type **Future<t*>** is translated to a generic **struct** that contains a handle to 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:

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

```
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 with the initialized thread handle[19]:

```
exported 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 };
} runTaskAndGetFuture (function)
```

The code shows that the arguments to be provided to the thread are copied onto a new location on the heap although they already reside on the heap as was shown in the previous section (TODO: backref). It is necessary to do so in order to avoid a dangling pointer. These could arise when a task is cleared before running out of scope so that the arguments get deleted while the running task instance, the declared thread is not finished, yet. Furthermore generally every thread needs its own copy of the data in case it modifies it. A corresponding 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. It 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 existance of the value after thread termination could not be guaranteed a dangling pointer[27] could emerge. The only safe alternative that fits the task-future structure well is to allocate memory on the heap and return the address of this memory(TODO: back-ref to parFun). The translation of the result retrieval is:

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

The **else** block reflects the usual future semantic. First the future is used to join the thread which blocks the execution until the thread is finished. Additionally the result is copied into the designated slot of the future. The result is at last 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 decide 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:

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

[19] 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 will become clear in the synchronization section(TODO: ref).

Both aforementioned generated functions take their future parameters by address. This is necessary to make the setting of the future data work. Otherwise only copies of the provided future arguments would be filled with data which would clearly not be intended by the programmer. This necessity 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 before without being stored and accessed via address. Hence the code contradicts the previously given definition of the translation of **`result`**. A first caveat would be to change the line to:

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

This on the other hand is not allowed because **`task.run`** is no lvalue[25, pp. 147-148] which disallows the utilization of the address operator. In order to allow for such chainings the unmodified code two wrapper functions, one for each **`join`** and **`result`**, are provided. These functions each take a future by argument, thus binding it to an adressable location, and call above corresponding functions in turn:

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

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

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

```
runTaskAndGetFuture(e')
```

```
runTaskAndGetVoidFuture(e')
```

```
getFutureResult(&e')
```

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

3.2.3 Example code

The running factors example can now be extended with future-related expressions:

```
// futures is an array of futures that on
// each return a pointer to a heap—managed value of type int32
Future<int32*>[threshold] futures;
for (int8 i = 0; i < threshold; ++i) {
  futures[i] = tasks[i].run;
} for

int32*[threshold] results;
for (int8 i = 0; i < threshold; ++i) {
  results[i] = futures[i].result;
  printf ("fibo[%d] = %d\n", i, *(results[i]));
} for
```

The sample translation contains the aforementioned generic definitions for future handling. The expressions become simple function calls:

```
Future[threshold] futures;
for (int8 i = 0; i < threshold; ++i) {
  futures[i] = runTaskAndGetFuture(tasks[i]);
} for

int32*[threshold] results;
for (int8 i = 0; i < threshold; ++i) {
  results[i] = ((int32*) getFutureResult(&futures[i]));
  printf ("fibo[%d] = %d\n", i, *(results[i]));
} for
```

3.3 Shared memory

The previous chapters introduced the means to enable parallel execution of code in terms of tasks and futures. Still missing is the communication between tasks. The communication model of choice for ParallelMbeddr is shared memory. The reason for this choice follows from the objectives of the model: it should offer a reasonable performance, considering that it is supposed to be used in embedded systems; it should be safe by design in order to avoid the trip hazards that are involved with low-level synchronization approaches like mutexes. Transactional memory seems to be not ready for the embedded domain for performance reasons. By following argumentation message passing does not offer profound advantages in comparison to shared memory if the access to the shared memory is controlled in a sane way. Usually message passing forbids shared memory between two parallel XX of execution. Communication is then realized via sent messages. Strict seperation of memory does not fit the usual C workflow concerning pointer arithmetic. Therefore some form of memory sharing would have to be introduced into a message passing model in order to reduce performance loss. As this would lead to the same problems that the general shared memory model already suffers 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 ever be attached in order to simplify the communication between tasks.

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 ressource*. Thus, a shared ressource can be regarded as a wrapper of data that is to be shared. In order to make use of a shared ressource it has to be synchronized first. Specific language elements are used to access and change the value of a shared ressource.

3.3.1 Design

The ressources to be shared are typed with the shared ressource type:

$t ::= ... | shared<u> | shared<shared<u>*>$

$u ::= t \qquad u! = t*$

The type parameterization denotes the base type of a shared type, i.e. the type of the data that is wrapped by a shared ressource. Due to reasons that will be explained in the SAFETY CHAPTER a shared ressource may have an arbitrary base type that does not denote a pointer to a value that is not shared[20]. Further restrictions apply to shared types which are also investigated in the aforementioned section. The same applies to the **`.set`** expression by which the value of a shared ressource can be modified; the value can be retrieved via **`.get`**:

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

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

The syntax `stmts` of statements in mbeddr is extended by the synchronization statement:

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

where `res` denotes the syntax of possible synchronization references. Each synchronization reference **`res`** wraps a reference to a shared ressource. It can either be of type **`shared<t>`** or of type **`shared<t>*`**. A shared ressource can be synchronized as is or be named[21]:

$res ::= e \mid e\ as\ [resName]$

The latter allows the programmer to refer to the result of an arbitrary complex expression which evaluates to a shared ressource inside the **`sync`** statement. Hence, a named ressource can be seen as syntactic sugar for a local variable declaration of a specific type. The scope of a named ressource is restricted to the synchronization statement; it can be referenced from anywhere inside the abstract syntax tree (AST) of the synchronization sa

The type of such a reference is given by the corresponding named synchronization reference under the condition that **`[resName]`** is a descendant of the statement list in the abstract syntax tree (AST) of the code. Thus, the scope of a named synchronization ressource is restricted to the synchronization statement list.

In contrast to Java's synchronization blocks (TODO: google reference) 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

[20] Differently said: A pointer wrapped in a shared ressource must point to a shared ressource itself.

[21] text

can obviously not be safely excluded with this scheme ParallelMbeddr ties the protection to the data that shall be shared. Every shared ressource is therefore protected seperately but application-wide. Concluding, consider two synchronization blocks which are about to be executed in parallel. If they contain synchronization references which overlap in terms of their referenced shared ressources their executions will be serialized:

Kommentar von Bastian
Maybe make diagram

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

// simultaneous executions
sync(value) { value.set(0) } sync(other, value) { value.set(0) }
```

==>

```
// serialized executions, not necessarily in this order
sync(value) { value.set(0) }
sync(other, value) { value.set(0) }
```

The possibility to refer to multiple shared ressources in the synchronization list of a synchronization statement is not mere syntactic sugar for nested synchronization statements. Instead the semantics of a synchronization list is that all referenced shared ressources are synchronized over at once, but with a possible time delay. Hence, deadlocks by competing synchronization statements are avoided.

As a result of the fact that generally access to shared ressources is ressource-centric, a value—wrapped in a shared ressource—which contains nested shared ressources is indepedently protected from the latter. Therefore a shared ressource of a struct with a shared member **b** is indepently synchronized from **b**:

Kommentar von Bastian
Maybe make diagram

```
struct A {
  int32 a;
  shared<int32> b;
}
shared<A> sharedA;
shared<int32>* b;
sync(sharedA) { b = &(sharedA.get.b); }
// simultaneous executions which will not affect one another
sync(sharedA) { sharedA.get.a = 0; } sync(b) { b->set(1); }
```

3.3.2 Implementation

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. Nevertheless any 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, the mutex was chosen: each protected data item is assigned exactly one.

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 ressource **`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 made (pseudo-) synchronization safe: E.g. 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 return 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 ressource which in this scenario is not feasible. Hence, the design was chosen to not allow for such protection and consequently a global map would not be beneficial in this regard. A map solution would entail a space-time tradeoff. For illustrative purposes consider Google's C++ *dense_hash_map* [22] provides comparatively fast access to its members but imposes additional memory requirements to slower hashmap implementations[23].

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. On the contrary the access of 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 access to the map thread-safe). The computational overhead imposed by a member lookup is deemed negligible in the overall application performance. The space required for the struct equals that of the individual fields plus additional padding[20, pp. 303 ff.]. The latter depends on the size of the data to be stored. In order to keep the the padding small small smart member ordering should be applied.

For this work the struct solution was chosen in order to keep the computational complexity of mutex lookups small (and deterministic -> TODO: ref) while not imposing too much space overhead and datum lookup overhead. For each kind of shared type **`shared<t>`** with the translated basetype **`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[24]. 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: Consider 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[25]. **TODO:** Maybe add visualization
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 shared types related code 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 shared types related code 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 mutex in the equally named struct field 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 in order 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 who 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

[22] http://goog-sparsehash.sourceforge.net/doc/dense_hash_map.html
[23] http://incise.org/hash-table-benchmarks.html
[24] Although the resolution of typedefs and constants impede corresponding changes 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.
[25] Clearly, this tree will have only one branch.

- 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 do that which makes sense as any variable **v** of this type would only 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 more complex type **v"**. The declaration of **v'**, respectively **v"** would then trigger the initialization of the mutex of the shared ressource that **v** points to. The resulting mutex initialization functions for types of the aforementioned kind are declared as follows[26]: **TODO:** Reparatur von shared<shared<int32>[5] five; siehe Code

TODO: shared+struct: Faelle fuer arrays einfuegen

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

// for a proper array type t[]...[] of 1 to n dimensions where ... denotes the occurrence of accordingly many symbols
// and t' denotes the reduced-to type
void mutexInit_X(t'*...* var, int32 size_0, ..., int32 size_n) {
  for (int32 __i_0 = 0; __i_0 < size_0; __i_0++) {
    ...
      for (int32 __i_n = 0; __i_n < size_n; __i_n++) {
        // in case t is a struct type
        mutexInit_X'(&var[__i_0]...[__i_n]);
        // or, in case t is a shared type with generic C base type:
        pthread_mutex_init(&var[__i_0]...[__i_n].mutex, &mutexAttribute);
      }
    ...
  }
}

// for a proper struct type t of a struct t { u_1 f_1; ...; u_n f_n } and according reduced field types u_1' to u_n'
void mutexInit_X(SharedOf_t* var) {
  //in case u_i demands further initialization
  mutexInit_X'(&var->f_i);
  // or, in case u_i is a shared type with generic C base type:
  pthread_mutex_init(&var->f_i.mutex, &mutexAttribute);
  ...
  //in case u_j demands further initialization
  mutexInit_X'(&var->f_j);
  // or, in case u_j is a shared type with generic C base type:
  pthread_mutex_init(&var->f_j.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 equally type the according function parameters.

The deletion of mutexes is defined quite similar with the main difference that the utilized according pthread function only takes one mutex. Therefore only the deletion for mutexes nested in ressources of shared types is shown:

```
void mutexDestroy_1(SharedOf_t'* var) {
```

[26] The meaning of the mutex attribute will be declared later on.

```
    pthread_mutex_destroy (&var->mutex);
    mutexInit_0(&var->value);
}
```

These presented functions are used to initialize mutexes at the begin 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[27]. As forced by mbeddr the programmer thus has to specify a main function in one of the implementation modules. 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[28]. The deletion of mutexes for shared ressources must be accomplished before they get out of scope which, again, depends on the kind of variables they are refered to.

TODO: Markus fragen, ob immer per main-Fkt. eingestiegen wird und ob Bibliotheksverwendung auch moeglich ist

Local variables' mutexes are destroyed before the end of their surrounding scopes. Additionally special care has to be taken of any control flow breaking statement *c* that occurs in the AST of the same function of some local variable declaration *l* that refers to some value with a (nested) shared ressource. **TODO:** forward ref to syncs

If

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

c must be preceded with a destruction call of the mutex of the shared ressource of *l*. **TODO:** Implement!

The proper destruction of mutexes of function arguments on the other hand just requires according function calls at the end of their functions and before any return statement that refers to their functions. The actual function to call for a variable or argument is either one of the **`mutexDestroy_X()`** functions previously explained or a direct call of **`pthread_mutex_destroy()`** for "simple" shared ressources of generic C types, e.g.:

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

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

==>

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

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

To recap: The mutex of a shared ressource is either directly initiliazed and destroy via appropriate pthread functions or it is indirectly handled via functions that are based on the types of the values that shared ressources are nested inside. This approach was chosen in order to reduce the amount of code duplication that would otherwise occur if mutexes of variables with shared ressources would be handled inline. As a result the generated code becomes more easily readable. 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*.

TODO: PrÃ¼fen, ob mutex-init-Reihenfolge richtig ist

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:

[27] There is no way to combine the definition and the initialization of mutexes as they are used in ParallelMbeddr.

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

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

Since each synchronization statement locks the mutexes of the refered shared ressources (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[29]: A mutex attribute that specifies the recursiveness must be defined and initialized first. It can then be used by arbitrarily many mutexes. For this purpose an application-wide attribute is defined in a generic module that is imported by all user-defined modules. It is initialized at the beginning of the main function:

```
// inside the generic module:
extern pthread_mutexattr_t mutexAttribute
// at the beginning of main:
pthread_mutexattr_settype(&mutexAttribute, PTHREAD_MUTEX_RECURSIVE);
pthread_mutexattr_init(&mutexAttribute_0);
```

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 becomes:

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

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

The statements are kept inside their statement list block in order to keep the scope of local variables inside synchronization statements. The block is reduced to another block where breaking statements that break the program flow structure may be preceded by a similar call of the **`stopSyncForNMutexes()`** function under certain conditions. 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 hold:

- *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*.

In this manner inconsistent synchronization states of shared ressources due to the aforementioned statements are omitted.

For each arity of synchronization ressources seperate 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:

```
// the corresponding function 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);
}
```

[29] 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

Abstracted over 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 taken care of. The illusion of atomic synchronization is realized by an implementation of the obstruction-free[30] busy-wait protocol *Polite*. In order to resolve conflicts *Polite* uses exponential backoff. The according backoff 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 backoff function to delay its execution for a randomized amount of time and repeats afterwards. This scheme enables competing threads to proceed.

```
1  // again, the equivalent function declaration for 1 mutex is skipped
2  exported void startSyncForNMutexes(pthread_mutex_t* mutex_0, ..., pthread_mutex_t* mutex_n) {
3    uint8 waitingCounter = 0;
4    uint16 mask = 16;
5    uint32 seed = (uint32)(uintptr_t) &waitingCounter;
6    while (true) {
7      if ([| pthread_mutex_trylock (mutex_0) |] != 0) {
8        backoffExponentially(&waitingCounter, &mask, &seed);
9      } else if ([| pthread_mutex_trylock (mutex_1) |] != 0) {
10       [| pthread_mutex_unlock (mutex_0) |];
11       backoffExponentially(&waitingCounter, &mask, &seed);
12     } ...
13     } else if ([| pthread_mutex_trylock (mutex_n) |] != 0) {
14       [| pthread_mutex_unlock (mutex_n—1) |];
15       ...
16       [| pthread_mutex_unlock (mutex_0) |];
17       backoffExponentially(&waitingCounter, &mask, &seed);
18     } else {
19       break;
20     }
21   }
22 }
```

The backoff realized by Polite randomizedly delays the execution by less than $limit = 2^{n+k}$ ns[28]. n denotes the retry counter and k denotes some constant offset which can be machine-tuned. When n reaches a specific threshold m it is reset to 0. The randomized wait time of the exponential backoff is used to avoid livelocks which could happen if two threads would repeatedly compete for the same ressources and delay their execution for equal amounts of time. In the current implementation **`backoffExponentially()`** the offset k is set to 4 and the threshold m is set to 17[31]. Thus, maximum delays of about 100 ms (specifically 131 ms) are allowed[32]. Note that **`backoffExponentially()`** keeps its main state inside the **`startSyncForNMutexes()`** function which will therefore be re-initialized before the execution of every synchronization block:

```
1  inline void backoffExponentially(uint8* waitingCounter, uint16* mask, uint32* seed) {
2    *mask |= 1 << *waitingCounter;
3    randomWithXorShift(seed);
4    struct timespec sleepingTime = (struct timespec){ .tv_nsec = *seed & *mask };
5    nanosleep(&sleepingTime, null);
6    *waitingCounter = (*waitingCounter + 1) % 13;
7  }
```

The generation of the pseudo-randomized delay is realized via utilization of the Marsaglia's Xorshift random number generator[21]:

[30] Busy-waiting means that the thread will repeatedly test a condition until it is met, without doing actual useful work[24, 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[6, p. 77]). Obstruction-free means that the execution of any thread which is at some time run in isolation such that the execution of obstructing other threads is interrupted meanwhile will progress. The existance of obstruction-freedom guarantees that no deadlocks will occur[3]. 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/.

[31] 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`**.

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

```
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()`.

4 Evaluation

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

Literaturverzeichnis

[1] Language workbenches: The killer-app for domain specific languages?

[2] *Introduction To Operating Systems: Concepts And Practice An 2Nd Ed.* Prentice-Hall Of India Pvt. Limited, 2007.

[3] 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.

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

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

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

[7] 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.

[8] 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.

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

[10] 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.

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

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

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

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

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

[16] 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.

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

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

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

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

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

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

[23] 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.

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

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

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

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

[28] 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.

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

[30] Markus VÃ¶lter. Generic tools, specific languages.

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