FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Nicolas Pfeiffer

A Wait-Free Cactus Stack Implementation for a Microparallelism Runtime

Masterarbeit im Fach Informatik

2. März 2020 — MA-I4-2020-02

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstr. 1 · 91058 Erlangen · Germany

A Wait-Free Cactus Stack Implementation for a Microparallelism Runtime

Masterarbeit im Fach Informatik

vorgelegt von

Nicolas Pfeiffer

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Florian Schmaus, M.Sc.**
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat**
Beginn der Arbeit:	**2. September 2019**
Abgabe der Arbeit:	**2. März 2020**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Nicolas Pfeiffer)
Erlangen, 2. März 2020

ABSTRACT

Microparallelism runtimes are an important tool for developing parallel software that enables the usage of feather-weight threads. Randomized work-stealing is the prominent way of scheduling and load balancing in these platforms, and the fork-join model is often used to express the task-parallelism. That entails a parallel calling stack, a so-called *cactus stack*. Work-stealing guarantees bounds on time and space consumption and promises nearly linear speedup for computations with sufficient parallelism. However, in practice, performance depends on many factors like the method used to implement tasks and the cactus stack, as well as the synchronization used internally. Fibril implements continuation-stealing tasks and an efficient cactus stack that performs similar or better than existing runtimes like Intel Cilk Plus and Threading Building Blocks. However, Fibril uses blocking synchronization internally. Locks serialize critical sections and cause bottlenecks, where threads have to wait until they can acquire the lock. Lock-based synchronization of critical sections with high contention results in bad scaling over many cores.

This thesis presents a fully lock- and wait-free implementation of continuation-stealing tasks and cactus stack, that supports the fork-join model. It is based on the implementation of Fibril, but replace the locks by a wait-free algorithm, that allows better utilization of CPU cores. Benchmark results show that the wait-free approach scales better over many cores. However, simultaneous multithreading can partially compensate for the worse scalability of locks. Furthermore, the cactus stack implementation, adapted from Fibril, resulted in reduced performance, below that of Cilk Plus, in some benchmarks. That indicates that there is still room for improvement in the cactus stack implementation, but also in terms of memory management and performance of the parallel calling convention.

KURZFASSUNG

Mikroparallelismus Laufzeitumgebungen sind ein wichtiges Werkzeug bei der Entwicklung von paralleler Software, dass die Verwendung von federgewichtigen Fäden ermöglicht. Randomisiertes *Work-Stealing* ist eine beliebte Methode der Ablaufplanung und der Lastenverteilung in solchen Programmierplattformen und das *fork-join* Programmiermodel wird oft genutzt, um die Taskparallelität auszudrücken. Dies hat einen parallelen Aufrufkeller zur Folge, der auch als *Cactus-Stack* bezeichnet wird. *Work-Stealing* hat obere Schranken für die Laufzeit und den Platzverbrauch und verspricht annähernd linearen *Speedup* für Berechnungen mit ausreichender Parallelität. In der Praxis hängt die Leistung allerdings von mehreren Faktoren ab, wie dem Verfahren, wie Tasks und der *Cactus-Stack* implementiert sind, und welche Art der Synchronisierung in der Laufzeitumgebung verwendet wurde. Fibril ist eine Implementierung, die Tasks mit *Continuation-Stealing* umsetzt und einen effizienten *Cactus-Stack* verwende, und damit vergleichbare oder bessere Leistung erzielt als existierende Laufzeitumgebungen wie Intel Cilk Plus und Threading Building Blocks. Allerdings verwendet Fibril blockierende Synchronisierung. Blockierende Synchronisierung serialisiert einen kritischen Abschnitt und kann zu einem Flaschenhals werden, wobei Fäden warten müssen, bis sie den kritischen Abschnitt betreten können. Blockierende Synchronisierung von kritischen Abschnitten, um die es einen starken Wettstreit gibt, skaliert schlecht mit vielen Prozessorkernen.

Dieser Arbeit stellt eine komplett nicht-blockierend und warte-frei synchronisierte Implementierung von *Continuation-Stealing* basierten Tasks und *Cactus-Stack* vor, die das *fork-join* Programmiermodel unterstützt. Die Implementierung basiert auf der Fibrils, ersetzt aber die blockierende Synchronisierung durch einen warte-freien Algorithmus, der es ermöglicht, Prozessorkerne besser auszulasten. Die Ergebnisse der Benchmarks zeigen, dass der warte-freie Ansatz besser mit der Anzahl verwendeter Prozessorkernen skaliert. Allerdings kann die Verwendung von *Simultanem Multithreading* das schlechtere Skalieren von blockierender Synchronisierung zum Teil ausgleichen. Außerdem verursacht der auf Fibril basierte *Cactus-Stack* Leistungseinbrüche, unter die Leistung von Cilk Plus, in manchen Benchmarks. Das zeigt, dass es in der Implementierung des *Cactus-Stacks*, aber auch was die Speicherverwaltung und Leistung der parallelen Aufrufkonvention angeht, noch Verbesserungspotenzial gibt.

CONTENTS

1 INTRODUCTION

Multi-core processors for Personal Computers (PCs) and PC-like systems exist for around one and a half decades. In recent years the average core count of multi-core processors more than doubled, increasing potential computing power. However, the software must be written explicitly parallel to take advantage of additional processor cores. Furthermore, a problem needs to have sufficient parallelism to scale with the number of cores, when computed in parallel. Some problems cannot be parallelized.

Writing efficient and correct parallel software is difficult. Some tools and strategies help develop parallel software. When it comes to task-parallel software, using kernel-level threads is often not the right approach since the creation of these threads is computationally expensive and does not yield good performance when the problem is split into many small tasks. For such applications, feather-weight threads, or fibers, are a better solution. These are user-space scheduled threads with low costs for creation and destruction.

To use fibers, a microparallelism runtime is required, that implements an interface to create and schedule the fibers. Such a runtime consists of a pool of kernel-level threads, the workers, that execute the fibers. Work-stealing is the prominent way of scheduling and load balancing in microparallelism runtimes.

The fork-join model is a flexible and easy-to-use way of expressing parallelism. It is often used as an interface for creating and synchronizing work packages in the form of fibers. Task creation can be implemented as child-stealing, where the created task is scheduled for execution, or as continuation-stealing, where the created task is executed directly, and the continuation of the forking function is scheduled for execution, instead. The implementation can have a fundamental impact on the performance of a runtime.

Furthermore, fork-join parallelism entails a tree-like calling stack, a so-called cactus stack. The cactus stack forms as a result of the diverging structure of dependencies between tasks due to the parent-child relationships between them. The way a microparallelism runtime implements its cactus stack has a direct influence on its performance and space consumption. Ideally, the cactus stack implementation should yield good speedup and have a low memory footprint. In practice, however, implementations cannot fully reach these goals or have to restrict interoperability between serial and parallel code. This is also known as the cactus stack problem. Yang and Mellor-Crummey [YMC16] have presented a “practical” solution to the cactus stack problem that promises good performance and low physical memory consumption, while being fully interoperable.

The research project Extensible Micro-Parallelism Experimentation Runtime (EMPER) will be used as the basis for the implementation work of this thesis. It is a microparallelism runtime that uses child-stealing to implement asynchronous tasks. It uses fully wait-free work-stealing dequeues for scheduling and wait-free private semaphores for synchronization of tasks.

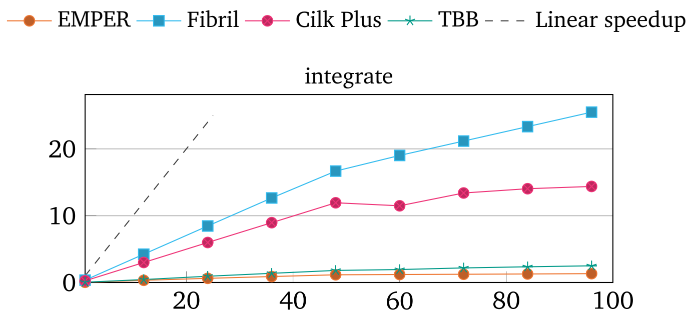

Figure 1.1 – Comparsion of micro-parallelism runtimes.

The X axis are the number of worker threads and the Y axis are the speedups T_{serial}/T_P, where T_P is the execution time using P worker threads.

Figure 1.1 shows the performance of different microparallelism runtimes on 1–96 threads. Intel Threading Building Blocks (TBB) [CM08; Suk09] and EMPER both use child-stealing, while Intel Cilk Plus [Rob13] and Fibril [YMC16] use continuation-stealing. While both methods can perform nearly identical in some cases, there are many problems where child-stealing performs very poorly, while continuation-stealing can achieve relatively good speedups. In this example, Fibril out-performs both TBB and EMPER by up to 10.2×. Cilk Plus does not perform as well in this application but still manages to perform up to 5.8× better than the child-stealing runtimes.

However, Fibril uses locked based synchronization internally, as highlighted in red in Listing 1.1. Locks serialize a critical section and can cause threads to wait at the beginning of the section. This is considered to hinder scalability by preventing parallelism and, thus, speedup. This thesis assumes that replacing the lock-based synchronization by a lock- and wait-free algorithm can further improve performance. Therefore, in this thesis, the possibilities of a fully lock- and wait-free implementation of a fork-join parallelism runtime with lazy task creation will be explored and analyzed. For this purpose, continuation-stealing tasks and a cactus stack based on the work of Yang and Mellor-Crummey [YMC16] will be implemented in EMPER, as the basis for the wait-free approach.

```
/* ... */
sync_lock(frptr->lock);
if (frptr->count-- == 0) {
    if (frptr->stack.ptr != fibrili_deq.stack) {
        stack_reinstall(frptr);
    }
    sync_unlock(frptr->lock);
    longjmp(frptr, frptr->stack.top);
} else {
    if (frptr->stack.ptr == fibrili_deq.stack) {
        STATS_COUNT(N_SUSPENSIONS, 1);
        stack_uninstall(frptr);
    }
    sync_unlock(frptr->lock);
}
/* ... */
```

Listing 1.1 – An excerpt of Fibril's code with lock-base synchronization.

2 FUNDAMENTALS

This chapter will first give an introduction into fork-join parallelism and the cactus stack, in Section 2.1. Then the so-called “cactus stack problem” will be explained in Section 2.2 and strategies to solve it in Section 2.3. Finally, EMPER will be explored briefly in Section 2.4.

2.1 Introduction to Fork-Join-Parallelism

The fork-join model is used to express logical task parallelism. It consists of the two keywords `fork` and `join`[1] that allow manipulation of the control flow of a program. In the following, the term *strand* will be used to describe a series of sequentially executed instructions that do not contain a `fork` or `join`. More precisely, `fork` and `join` form the starting and ending points of a strand [Hal12].

`fork` “splits” a strand into two, it “forks” the strand, whereby the initial strand ends and two new strands begin. The two new strands may be executed in parallel, but do not have to. `join` has the opposite effect of `fork`. Where `fork` splits strand, `join` merges them. At a `join` two or more strand reunite, and only a single strand leaves the `join`. More precisely, the incoming strands end and a new strand leaves the `join`. The execution of strands in an application forms a Directed Acyclic Graph (DAG), where the `fork` and `join` points are the vertices, and the strands are the edges. Time defines the direction of the edges [Hal12].

This model of fork-join parallelism can be further restricted in its combination with functions. The resulting model of *strict fork-join parallelism* [Hal12] requires that every function has exactly one incoming strand and one outgoing strand, analogous to a fully serial model. A function can fork off another function. The two functions have an asymmetric relationship with the forking function being the *parent* and the forked off function being the *child*. For this strictness property to hold, a function can fork off arbitrarily many child functions that may (but do not have to) run in parallel

[1] Sometimes different names are used, e.g. in Cilk [FLR98] they are called spawn and sync, but the semantics usually remain the same.

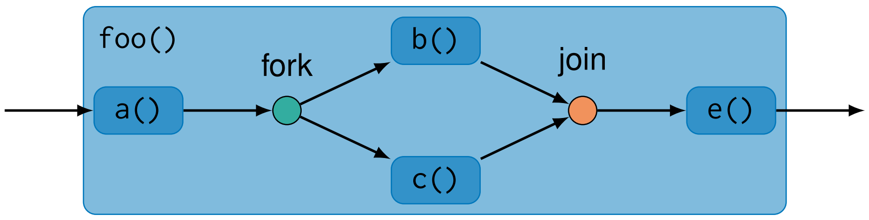

Figure 2.1 – A DAG of strict fork-join parallelism.

with the parent function and each other, but before the function can return, it has to join with all of its children. A `join` synchronizes a parent function with its children since the parent has to wait for all of its children to finish execution. In a similar fashion to function calls where the caller, the parent function, cannot return before the callee, the child function, does, the child function is strictly nested within the parent, as shown in Figure 2.1. That means that serial semantics are preserved. If a program is executed by one processor and strands are not executed in parallel, but rather in a serialized fashion, all forks behave like function calls. This property that the semantics of a program remain the same if all forks are substituted for function calls (and joins become no-ops) is referred to as *C elision* or *serial elision* of a program [FLR98].

```
foo() {
    a();
    fork b();
    c();
    join;
    e();
}
```

Listing 2.2 – A simple fork-join example.

When it comes to implementing a runtime that supports fork-join parallelism, many decisions regarding the architecture have to be made. Some of which can have a fundamental impact on performance and efficiency. In the following, major design choices will be analyzed and discussed.

Generally, such a runtime has a pool of worker threads, in the following referred to just as workers, that execute in parallel. These workers execute tasks, which are feather-weight threads. The usage of kernel-level threads as tasks is disadvised because the creation of kernel-level threads is too expensive. When a worker encounters a `fork`, it creates a new task, that is then scheduled for execution by the pool of workers. Listing 2.2 shows a code example for such a scenario and Figure 2.1 the corresponding DAG. Assume worker W_1 executes function a() before forking off function b(). The first decision to be made is which worker executes which function. W_1 can continue to execute c() and have another worker W_2 execute b(), or it can execute b() itself and have the W_2 continues with the execution of c(). While this might seem like a minor detail, it can very well have a tremendous influence on efficiency as will be discussed in more detail in Section 2.1.2 and Section 2.1.3. Similarly, when joining b() and c(), executed by W_1 and W_2, only one worker can continue after the `join` and execute f(). The choice is between W_1 that initially executed a() (or another specific worker, to be more general) or the worker executing the last task to join [Rob14]. This decision and its consequences will be discussed in Section 2.1.4. Scheduling of tasks, even though not the focus of this thesis is important for the understanding of advanced topics of this thesis and will be explored in Section 2.1.1. Lastly the main topic of this thesis, the fork-join call stack or cactus stack will be explained in Section 2.1.5 and the problems encountered when trying to implement it efficiently in Section 2.2.

2.1.1 Scheduling by Work-Stealing

Scheduling is a broad field of study. The impact scheduling has on performance is analyzed in much detail. Depending on the application and the requirements, different scheduling strategies are suitable. For scheduling and load balancing fork-join tasks in multicore systems, randomized work-stealing has become the established strategy. It is used by Cilk [FLR98], Cilk++ [Lei09;

Fri+09], Intel Cilk Plus [YMC16; Rob13], OpenMP [YMC16; Rob14], TBB [CM08; YMC16; Rob14] and Fibril [YMC16].

In contrast to *work sharing* scheduling strategies, where new tasks after creation get assigned to a worker in hopes of reaching an even distribution of work among workers, the idea behind *work-stealing* is that newly created tasks are not shared, but rather kept with the worker that created the task, but can be stolen by another work. The algorithm for *randomized work- stealing* as described by Blumofe and Leiserson [BL99] works like this: Every worker has its own ready-queue for new tasks, as can be seen in Figure 2.2. The queues are double-ended queues, dequeues, that have a top and a bottom end. A worker treats its dequeue like a linear stack, pushing new tasks to the bottom, and after finishing a task, the worker pops the next task to be executed from the bottom end. If a worker runs out of work, that is, its dequeue is empty, and an attempt to pop failed, then it becomes a thief and tries to steal tasks off the top ends of other workers' dequeues. It therefore randomly selects a worker, the victim, and attempts to steal the topmost task from the victim's dequeue. Should the thief fail to steal from the victim (in case the victim's dequeue is empty), then the procedure is repeated, starting with selecting a new victim. When the thief eventually succeeds, it then proceeds to execute the stolen task. By doing so, it potentially creates new tasks that are then pushed to the bottom of the thief's dequeue again.

This way of scheduling tasks minimizes communication between workers and the migration of data because workers are only required to interact with each other when they run out of work and have to steal. Conversely, in a work-sharing scheduler, tasks will be assigned to workers even if they still have work to do. And since the data a task works on is often related to the data of the parent task, migration of tasks also includes migration of data. More precisely, when a processor executes a task, and the task spawns a new task, the work the new task will do is, in many cases, directly linked to the data of the parent task. Therefore, the same processor executes it, the data is, with high probability, already present in caches, whereas, should the task be executed by a different processor, there is a higher chance that the new processor has to load the data from slower memory.

Furthermore, Blumofe and Leiserson showed that randomized work-stealing, when stealing is unrestricted achieves strong bounds on time and stack space for *strict* multithreaded computations.

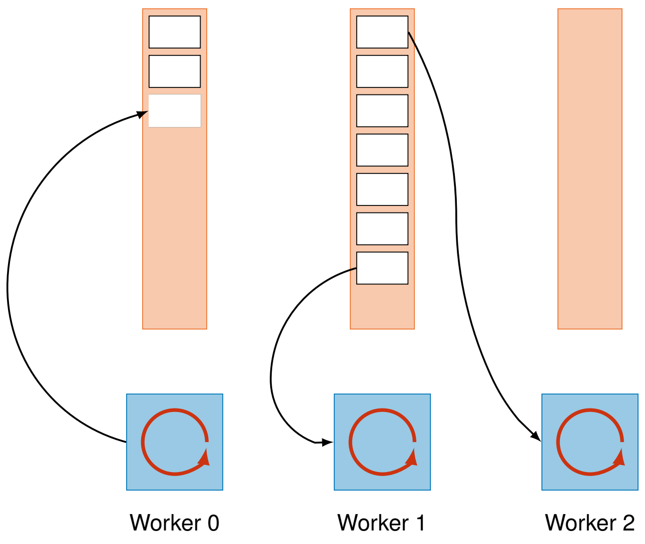

Figure 2.2 – Work-stealing dequeues example.

The given definition of "strict" computations requires that in a fork-join activation tree, the only edge into the subtree emanates from the parent that spawned the subtree and goes to the root of the subtree. The only edges leaving the subtree are joining edges that go from a given task to the ancestors of that task. Strict fork-join parallelism, as defined in Section 2.1, is a subclass of strict multithreaded computations. Strict fork-join parallelism is even more restrictive because it allows only one join edge from the root of the subtree to its parent.

Blumofe and Leiserson proved that a randomized work-stealing scheduler can achieve the following upper bounds for expected execution time and stack space usage, when stealing is unrestricted: Let T_1 be the *work* of a deterministic, strict multithreaded computation, its execution time of the computation on one processor, and let T_∞ be the *critical path length*, or *span*, of the computation, its ideal execution time on an infinite number of processors[2], with constant value c_∞, the *span overhead*. Then the computation can run on P processors in expected time

$$T_P \leq T_1/P + c_\infty T_\infty. \tag{2.1}$$

This inequation, also known as "Brent's Lemma", guarantees *linear speedup* in the number of processors P when the average parallelism of the computation is much larger than the number of processors P, that is $T_1/T_P \gg P$. Furthermore, if S_1 is the required stack space to execute the computation on one processor, then the computation on P processors uses

$$S_P \leq PS_1. \tag{2.2}$$

The bound given by the second inequation guarantees that the increase in stack space usage by parallel execution of the computation is linear in the number of processors P at worst [YMC16].

2.1.2 Child-Stealing

When it comes to implementing a fork-join runtime, there is a design choice of how to map tasks onto workers, as pointed out at the end of Section 2.1. In the following, one of these alternatives will be explored and analyzed. Child-stealing is probably the more intuitive way of implementing fork. The worker executing the task that encounters a fork creates the new task and leaves it to be stolen for execution by another worker. The initial worker continues normally with the execution of its task after the fork. Figure 2.3 shows such a scenario, where the task T_2, created at a fork, is executed by W_2, another worker than W_1, the one that created the task.

While this design might look appropriate for a fork-join runtime, it has a flaw concerning memory consumption that can reduce the performance and efficiency of a runtime using it. The following example will help to illustrate the downside of child stealing. The loop in Listing 2.3 iterates n times, spawning a new task each iteration. The loop is an example that also stands for more complex loops, where the number of iterations n cannot be computed in advance, because it can depend on results of the execution of iterations, like a `while(true) { ... }` loop with break condition inside the loop. In child-stealing the task executing the loop has to run the whole loop to completion, creating all n tasks, before it reaches the join and can start to execute any of the new tasks, as seen in Figure 2.4. If there are no free workers available that can start working on the n tasks right away, this requires space to hold all n tasks in memory at the same time. This space consumption is proportional to n and, therefore unbounded for some loops. Depending on the problem size and the loop, child-stealing can be impractical in such scenarios. Moreover, allocating the required memory to store many tasks can hurt performance, lowering speedup. To circumvent this space blowup, there are strategies to restrict the creation of tasks. Unfortunately, there is the possibility of restricting

[2]The critical path length of a computation can also be understood as the longest path of instructions, from start to end of the computation, that has to be executed sequentially.

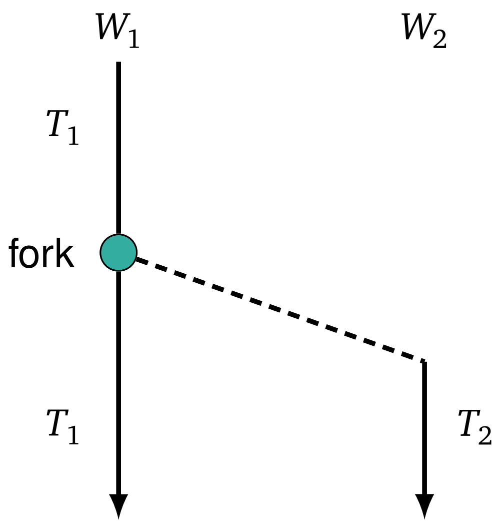

Figure 2.3 – Example timeline of child-stealing.

parallelism too much and, thus, decreasing the achievable speedup. Nevertheless, child-stealing has some advantages worth mentioning compared to its alternative (discussed in Section 2.1.3). It is easier to implement as “just a library”, without compiler support, and it can have a performance advantage in some situations because the costs of creating and dispatching a task is generally a bit lower [Rob14].

```
for (int i = 0; i < n; i++)
    fork f(i);
join;
```

Listing 2.3 – An example of a forking loop.

2.1.3 Continuation-Stealing

In child-stealing, `fork` is a statement that a new task *must* be created. There is no way of specifying that a task *should* be created and leaving the decision whether a task will be created to the runtime to decide based on the availability of free workers. Such *lazy task creation* [MKH91] is necessary to bound the space consumption of task creation without hindering speedup, as discussed previously (in Section 2.1.2). This is also referred to as *dynamic parallelism* since the decision is made dynamically at run-time.

Continuation-stealing is a way of implementing a dynamic `fork` that allows for concurrency but does not enforce it. When a worker W_1 is executing a task T_1 and encounters a `fork`, it creates the new task T_2 but starts to work on the new task T_2 right away and pushes the *continuation* of the initial task T_1 onto the bottom of its work-stealing dequeue. The continuation of T_1 can then be stolen by another worker, W_2, that continues the execution of the initial task, thus creating parallelism. The left timelines of Figure 2.5 illustrate this scenario. However, in the case that there

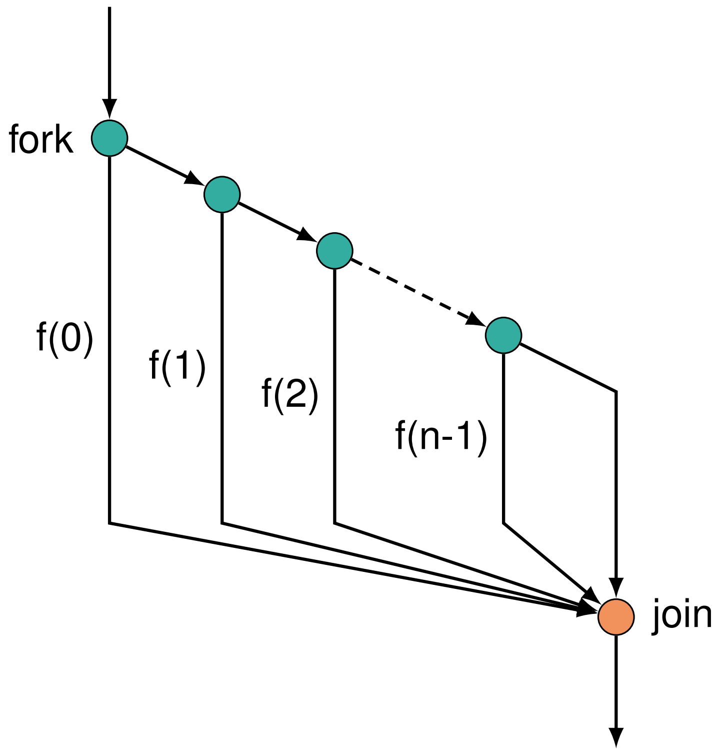

Figure 2.4 – DAG of a forking loop.

is no available worker that steals the continuation, W_1 will eventually complete the execution of T_2 and then try to pop T_1 from the bottom of its dequeue. Since T_1 was not stolen, W_1 will succeed and continuing the execution of T_1 itself, as shown by the right timelines of Figure 2.5. Semantically this is identical to a function call. Therefore, the parallelism is dynamical and depends on the availability of free, or work-stealing, workers.

The difference between child-stealing and continuation-stealing becomes apparent when going back to Figure 2.4, the loop example of Section 2.1.2. With continuation-stealing, the worker encountering the `fork` inside the loop will spawn the new task and start executing it right away, leaving the continuation to be stolen. If a free worker is available, it will steal the continuation, execute the next iteration of the loop, and by doing so, spawn one more task, which will, in turn, be executed right away, leaving the continuation for the next thief to steal. The amount of simultaneously existing tasks is linear in the number of worker P and therefore bounded, whereas with child-stealing all n tasks will be spawned before any of them are executed if there are no free workers.

However, while `fork` is semantically identical to a function call, if the continuation is not stolen, the computational costs associated with `fork` are not comparable. Since the continuation has to be saved and restored and has to be pushed to and popped from the work-staling queue, the costs become a multiple of that of a function call [Rob14].

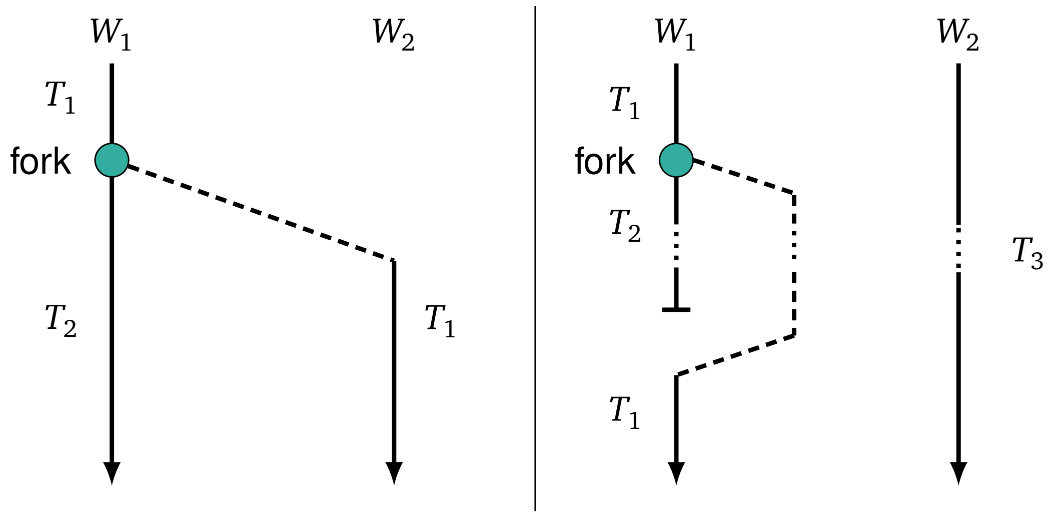

Figure 2.5 – Example timelines of continuation-stealing.

2.1.4 Joining Strategies

For implementing `join`, there are two options to chose from regarding which worker executes a task after joining, called "stalling" and "greedy" scheduling. With stalling scheduling, the worker that was executing the task before any `fork`, or for that matter any other specific worker[3], has to continue executing following a `join`. In Figure 2.1, that would be the worker that executed `a()`. The name "stalling" comes from the fact that the worker that has to execute the task might be busy, while free workers that could continue execution of the task are not allowed to do so. The free workers can then try to find any other work, but if there is not any, they have to wait and "stall" until new work is available.

With greedy scheduling, on the other hand, the last worker to join, whether it is the worker executing the task that encountered the `join` or one executing a child task, will proceed to execute after the `join`. That way, there are no stalling workers as long as there is work to do. Furthermore, greedy scheduling is necessary in order to achieve the time-bound of Equation (2.1). Since greedy scheduling is a form of stealing in a work-stealing scheduler because the worker executing a task can change at the `join`, also called *joining steal*, a stalling scheduler would restrict stealing and thus break the time-bound [Rob14; BL99].

Nevertheless, stalling schedulers have an advantage when it comes to task identity. Stalling schedulers preserve the identity of a task and, therefore, work better with some mutex implementations, where a change of the executing worker, as happens with greedy scheduling, can break break the mutex or cause a deadlock [Rob14].

2.1.5 Cactus-Stack

Fork-join parallelism entails a particular non-linear call stack, a so-called *cactus stack*. To understand how it differs from a linear call stack of a C-like language, it helps to understand first how a linear stack works and why it suffices. In such a C-like language, every function instance has an activation frame that contains its state. That includes the arguments for the function instance, the return

[3]A system might have additional constraints for scheduling, e.g. tasks might have affinities towards workers.

address, and the local variables the function uses. When a function is called, an activation frame for a new function instance is allocated, and when the function instance returns the activation frame is freed again. The order in which activation frames are allocated and freed, when a program is executed, matters, and dictates the layout of active activation frames. Figure 2.6 shows the *invocation tree* for a simple function `fib(n)` that computes the nth number in the Fibonacci-sequence, for which the code is given in Listing 2.4. The invocation tree shows the relationship between function instances. The node `fib(3)` represents a function instance of the `fib(n)` function invoked with argument `3`. It calls itself recursively with arguments `2` and `1` during its lifetime. Therefore the node of function instance `fib(3)` is the parent of nodes `fib(2)` and `fib(1)` in the invocation tree. The child nodes are arranged from left to right in the order they are invoked, `fib(2)` is called before `fib(1)` in function instance `fib(3)`.

A depth-first traversal of the invocation tree matches the order in which function instances are invoked during the execution of a program. Since in a serial program, a call suspends execution of the caller until the callee returns, a parent node's lifetime in the invocation tree cannot end before its child node's lifetime does. Therefore, all function instances on the path from an alive node to the root of the invocation tree, including the root, must be alive, too, at the same moment in time. Furthermore, no two function instances with the same depth in the invocation tree can be alive at the same time, because a parent function instance is suspended until its child has returned and can thus not invoke any other function instance. Hence, all activation frames of child function instances of a given parent function instance can reuse the same memory area[4]. This property allows a serial program to use a linear stack, where the activation frame for a callee is allocated directly beneath the activation frame of the caller. When the callee returns, the activation frame will be freed, and the space on the stack can be reused. Figure 2.7 illustrates this growth and shrinking of the linear stack. It shows how the arrangement of activation frames changes over the course of the computation of `fib(4)`.

```c
int fib(int n)
{
    if (n < 2)
        return n;
    int a, b;
    a = fib(n - 1);
    b = fib(n - 2);
    return a + b;
}
```

Listing 2.4 – Fibbonacci function.

Conversely, in a parallel program, using fork and join, a function can have multiple active children at the same time. The very idea of fork is to have a function fork off a child function and then continue execution in parallel with the child, allowing to call or fork more children. Therefore, while the invocation tree stays the same, the argument that nodes with the same depth in the invocation tree cannot be alive at the same time is not true anymore in a parallel program. Without the property that a function can have only one child at any moment in time, a linear stack does not suffice for allocating activation frames anymore. Any attempt to allocate an activation frame for a new child on a linear stack, while there is already an active child's activation frame on the linear stack, would lead to a collision, because a new activation frame is always allocated directly beneath the parent's

[4]In case the activation frames differ in size, the memory area will overlap only partly.

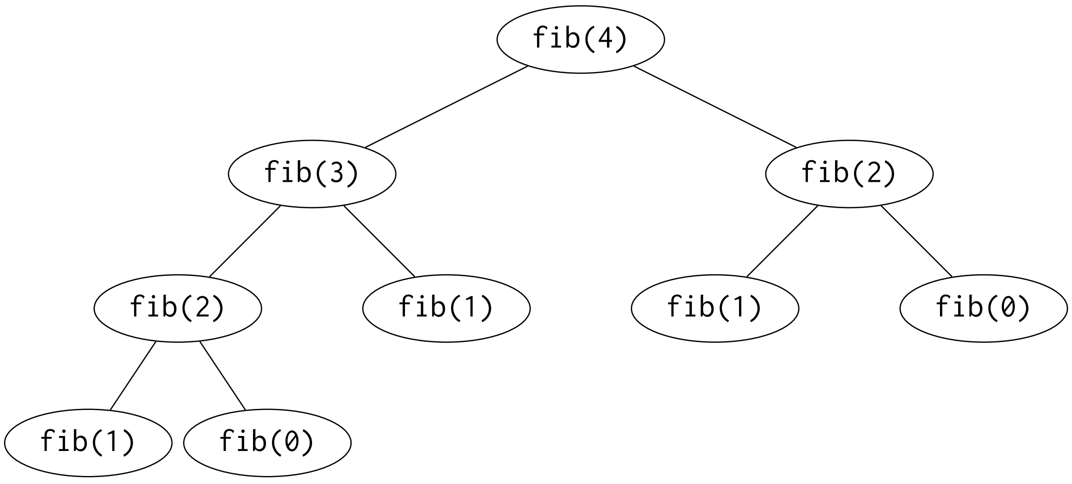

Figure 2.6 – The invocation tree of `fib(4)`.

activation frame. In Figure 2.7, when `fib(3)` would try to allocate an activation frame for `fib(1)` while the one of `fib(2)` is still alive, it would try do so at the same point on the linear stack, causing a collision of both function instances' activation frames.

```
int fib(int n)
{
    if (n < 2)
        return n;
    int a, b;
    fork a = fib(n - 1);
    b = fib(n - 2);
    join;
    return a + b;
}
```

Listing 2.5 – Parallel version of the Fibbonacci function.

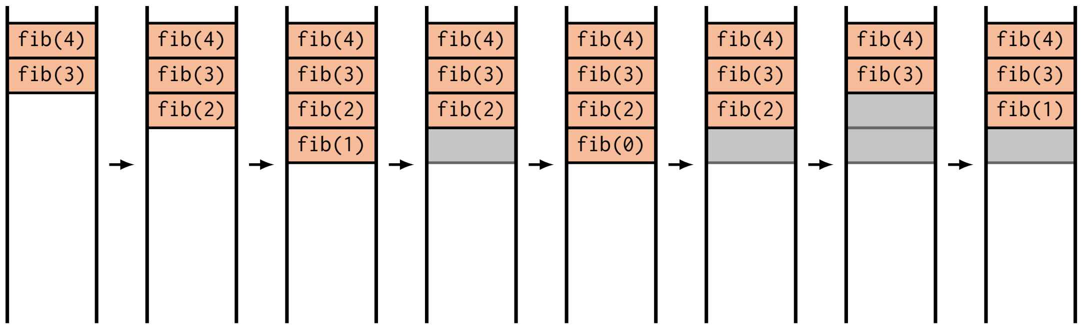

Figure 2.7 – Linear stack example for the invocation of `fib(4)`.

In order to solve this problem a different data structure has to be used to allocate activation frames. When a function instance already has one active child, whose activation frame is located on the same linear stack as its own and it wants to create a new child, it has to allocate a new linear stack for the new child to use. The child function instance can then allocate its activation frame on the top of the new stack and use it as a normal linear stack, except that in order to return, the child function instance has to switch back to the parent's linear stack. To do so, it has to store a pointer to the parent's activation frame located on the parent's stack. That way, a tree of stacks is formed, with the linear stacks being the nodes, pointing to the parent node. More precisely, an N-ary tree is formed, since a function instance can have multiple children on different stacks simultaneously and a stack can have arbitrarily many activation frames with multiple children. This tree-like stack data structure that can be traversed from the leaves to root (but usually not the other way) is also known as *cactus stack* [YMC16].

At any moment in time, during the execution of a computation, the invocation tree of *active* function instances consists of the subset of the invocation tree's nodes that are alive at that moment in time. If at any moment during computation, every extant leaf of the tree is executed by a worker, that is, no suspended frame is a leaf of the invocation tree of active function instances, as present at that instant, then all leaves are "busy". This is referred to as the *busy-leaves* property. From this property follows that there are no more than P leaves at any moment during execution, where P is the number of workers. Furthermore, S_1 is defined as the stack space consumption of the path, from a leaf to the root, that requires the most space. Therefore, the stack space is bounded by PS_1, the space-bound of Equation (2.2), if the busy-leaves property is maintained [BL99].

Listing 2.5 shows the parallel version of the `fib(n)` function. The cactus stack layout for the parallel computation of `fib(4)`, as formed by continuation-stealing, is shown in Figure 2.8. Analogous to the linear stack example of Figure 2.7, the activation frame for the first child function, the forked off child function, is allocated on the linear stack, below its parent's frame. When the continuation of the parent function is stolen and the second child function is invoked, its stack frame is allocated on a new stack. The cactus-stack layout for child-stealing is different, and it depends on the implementation and the work-stealing actually happening at run-time, in general.

The fact that for fork-join parallelism, a linear stack is not sufficient comes directly from the structure of fork-join parallelism itself. Therefore, every fork-join parallelism runtime *has* to implement a cactus stack, whether it uses child- or continuation-stealing, greedy, or stalling scheduling. An efficient implementation of the cactus stack is necessary to achieve good performance while maintaining practicality [YMC16; Lee+10]. In Section 2.2, the problems of implementing the cactus stack will be discussed, and in Section 2.3, some strategies to solve these problems will be shown.

2.2 The Cactus-Stack Problem

The way a work-stealing fork-join parallelism runtime implements the cactus stack defines fundamental properties of the runtime, like its performance, efficiency, or even usability. Ideally, it should have the following three properties: It should maintain the time-bound of Equation (2.1) and space-bound of Equation (2.2), induced by the work-stealing scheduler. The time-bound enables a program with sufficient parallelism to achieve nearly perfect linear speedup when using the runtime, while the space-bound ensures space consumption does not blow up and stays at a practical level to be usable for general-purpose systems. For software to be reusable and to be usable with serial binaries that are compiled to use a linear stack, it needs to be interoperable with serial code. That means, in particular, that functions that fork off other functions and functions that can be forked

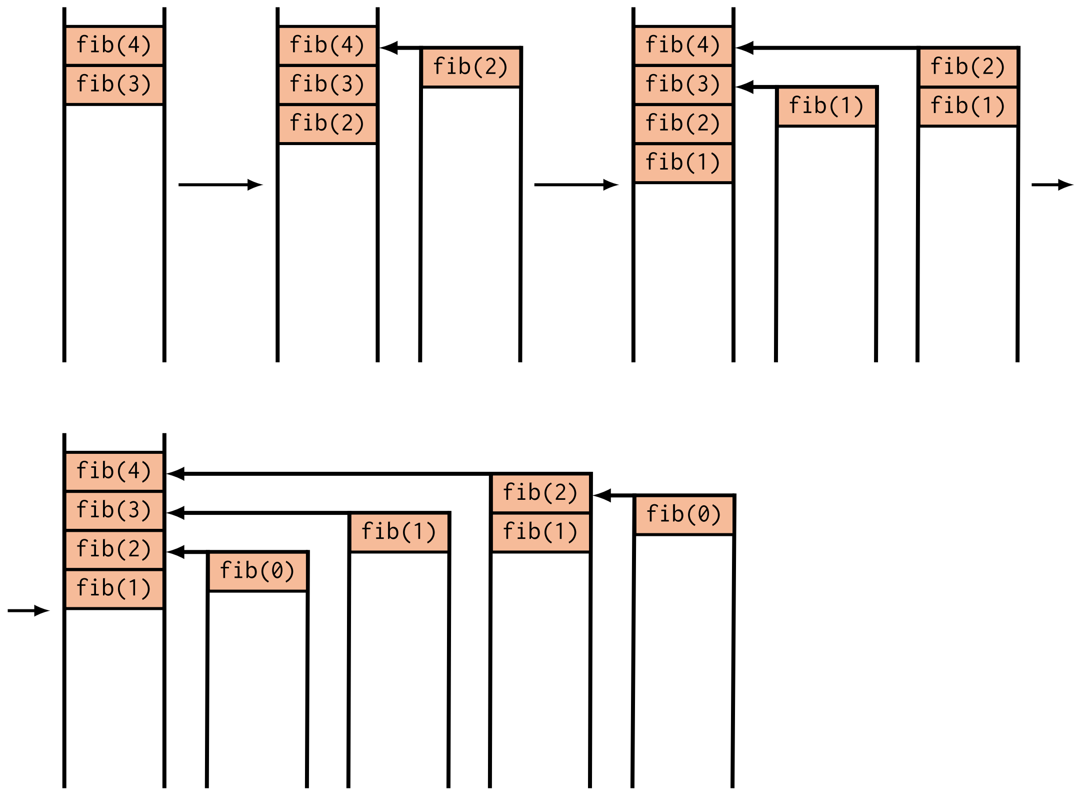

Figure 2.8 – Continuation-stealing cactus stack example for for the invocation of `fib(4)`.

off should use a standard, serial calling convention. This property is also known as *serial-parallel reciprocity* [YMC16; Lee+10].

In practice, however, it turned out that it is difficult to achieve all three properties simultaneously, and most implementations sacrifice one property to maintain the other two. This is also known as the *cactus stack problem*. Cilk [FLR98] uses a custom calling convention that prohibits serial C code from calling a cilk function directly, to maintain the strong time- and space-bounds. TBB [Suk09] allows calling a parallel function, but restricts stealing and thereby breaks the time-bound to avoid space blowup, leading to sub-linear speedup for some computations [YMC16]. In Section 2.3, various approaches and implementations will be discussed.

To better understand why it is difficult for a cactus stack implementation to achieve all three properties, it is necessary to understand why a naive implementation of the behavior described in Section 2.1.5 does break the space-bound. The following explanation assumes a runtime using a greedy work-stealing scheduler that implements continuation-stealing. As explained in Section 2.1.4 a stalling scheduler already breaks the time-bound and can, therefore, not be used to solve the cactus stack problem fully . Furthermore, child-stealing can result in unbounded space consumption for keeping track of created tasks, which can be impractical. And while this space consumption does not break the space-bound of Equation (2.2), which is concerned with stack space blowup, a case, similar to the following example, can be made to show that a naive implementation would break the space-bound as well.

As described in Section 2.1.1 and Section 2.1.3, when a task returns from a `fork` the executing worker tries to pop the bottommost task, the previously pushed continuation, from its work-stealing dequeue. If the continuation was stolen by another worker, the attempt will fail. In accordance with greedy scheduling, the worker then has to try to resume the stolen continuation. If the task that returned from the fork, is not the last to join, that is, there are other pending children, then the worker cannot resume the continuation and now has a suspended activation frame on its stack. The worker has to start work-stealing to find new work and allocate an activation frame for the new task, should it find one. However, the worker cannot do so on its current stack because it inhibits the suspended frame. If the worker would use the space below the suspended frame, it could cause a collision, because another worker could resume the suspended frame, before the stolen task is completed, as per greedy scheduling. If the resumed task then tries to use the same space on the stack to allocate another activation frame, it would cause the collision. The suspended frame can also not be moved in order to prevent this from happening. Moving a frame to a different address in memory would invalidate pointers to local variables inside the frame [5]. That is the core issue of the cactus stack problem. Solving the cactus stack problem means finding a solution that maintains all three of the postulated properties.

The naive approach of using a new stack, whenever the current one is blocked by a suspended frame, does not satisfy the space-bound, since each stack could hold as little as one frame, while still occupying S_1 pages of memory. The space-bound of Equation (2.2) requires that every worker P does not use more than S_1 pages of stack space. However, the activation frames of any worker can be scattered over up to D stacks, where D is the fork depth, the maximum amount of forks on any path from a leaf to the root of the invocation tree. Therefore, the space consumption becomes DPS_1, which breaks the space-bound for any $D > 1$. This can lead to impractical space consumption since D depends on the application and can be arbitrarily large. Solving the space blowup often requires sacrificing other properties. Cilk [FLR98] allocates continuations on the heap and thereby avoids having suspended activation frames blocking the stack. However, to do so, it is necessary to use a custom calling convention that prohibits serial code from directly calling a cilk function. Thus losing *serial-parallel reciprocity*. Other approaches restrict stealing to specific tasks and stall to maintain the space-bound, sacrificing the time-bound by doing so, which leads to sub-linear speedup for some computations [YMC16; Lee+10; Suk09].

2.3 Related work

There are different strategies to try to solve the cactus stack problem. Most solve it only partly and lose the interoperability with serial code or have only strong bound on time or space, and the other bound is weak at best. In the following, some of these approaches and existing implementations will be discussed. Table 2.1 gives an overview of strategies and implementations and their properties.

The idea behind “recompile everything” is to allocate activation frames on the heap rather than on the stack. That way, the situation that a stack is blocked because of a suspended frame can be avoided. However, since the serial calling convention allocates frames on the stack, parallel functions are not interoperable with serial code anymore. A custom compiler has to be used, that supports heap-allocated activation frames. Therefore, everything that is supposed to work with

[5] In a native environment pointers are not updated when when a frame is moved to different address in memory. Banning the use of pointers to local variables would impair the interoperability of a runtime, thus, resulting in a runtime that does not solve the cactus stack problem.

Strategy	Serial-Parallel-reciprocity	Time Bound	Space Bound
recompile everything	no	very strong	very strong
one stack per worker	yes	very strong	no
depth-restricted stealing	yes	no	very strong
limited-depth stacks	yes	no	very strong
new stack when needed	yes	very strong	weak
Fibril	yes	very strong	strong
recycle ancestor stacks	yes	very strong	weak
leapfrogging	yes	no	strong
TLMM cactus stacks	yes	strong	strong

Table 2.1 – Attributes of different strategies for implementing a cactus stack.

the parallel code has to be recompiled to work with the custom calling convention. However, in case functionality of a binary, already compiled without the parallel calling convention in mind, is used, it will not work. Lee et al. [Lee+10] speculate about the possibilities of using "binary-rewriting" techniques to patch such binaries to be compatible. For shared libraries, this would have to be done "on the fly", at startup of the application or at loading time of the shared library. Such techniques, however, would be difficult or impractical to implement due to optimizations in the binaries. Nevertheless, there are implementations using this approach. Cilk [FLR98] and Cilk++ [Lei09; Fri+09] implement heap-allocated continuations in combination with a greedy work-stealing scheduler. These implementations rely on a custom compiler. At a `spawn`[6], the compiler generates code that saves all local variables to the heap-allocated continuation activation frame. After stealing a continuation or after resuming a suspended continuation after a `sync`, the local variables will be loaded from the heap-allocated continuation again. That way, there are never suspended frames on the stack, and one stack per worker is sufficient. Thus, a strong time- and space-bound can be maintained in favor of *serial-parallel reciprocity* [Lee+10].

Another strategy, called "one stack per worker", limits every worker to only one linear stack. When a worker has a suspended frame on its stack, it starts work-stealing and executes stolen tasks on its own stack, using the next free space below the suspended frame. More precisely, the thief sets its base pointer to the top of the stolen continuation's frame residing in the victim's stack, to access local variables in the stolen frame. Then it sets its stack pointer to the next free space on its own stack. When the worker executes the stolen continuation, and a new function is called or forked off, the new frame will be allocated on the thief's stack at the position of the stack pointer, thus growing the thief's stack. When the thief was already deep in its stack, and the stolen continuation is shallow in the victim's stack, the thief's stack can potentially grow very deep, up to $2S_1$. With this, the busy-leaves property does not hold anymore, since the stack grows much larger than S_1. Since a worker has to steal work whenever the deepest frame on its stack cannot be resumed or popped, its stack grows further. This process can repeat arbitrarily many times in succession and, therefore, breaks the space-bound [Lee+10; YMC16].

A variation of "one stack per worker" is called "depth-restricted stealing". To prevent the stack from growing arbitrarily large, a worker is only allowed to steal frames that are deeper in the victim's stack than the bottommost frame on its own. A frame already deep in the stack will not grow the stack as much as a shallow one. Thus, a workers stack should not become larger than S_1, and a

[6]Cilk uses the keywords `spawn` and `sync`, instead of `fork` and `join`.

strong space-bound can be achieved. However, by restricting stealing, a worker can come into a situation where there is work to do, but the worker is not allowed to execute any of it, due to the restriction, and has to stall, thereby breaking the time-bound. Sukha [Suk09] has shown that for some computations, "depth-restricted stealing" is not faster than a sequential execution, yielding only constant speedup [Lee+10; YMC16]. TBB combines child-stealing with a stalling scheduler that uses a strategy similar to "depth-restricted stealing". TBB is entirely library-based and does not need dedicated compiler support. It is fully *serial-parallel reciprocal* and has a strong space-bound, but no time-bound.

Another strategy similar to "one stack per worker" is "limited-depth stacks". As the name suggests, this strategy bounds the size a stack can grow, by limiting the maximal depth of a worker's stack. When a certain depth is reached, the worker is not allowed to perform further stealing until frames on its stack are freed. That way, a strong space-bound can be achieved, but the time-bound is sacrificed because the time spent waiting cannot be compensated for. This prevents linear speedup for computations with copious parallelism [Lee+10].

The strategy "new stack when needed" is essentially like the example given in Section 2.2. Whenever a worker has a suspended frame on the top of its stack, and it has to go work-stealing, it allocates a new stack to execute the stolen tasks on. Analogous to "one stack per worker", the thief uses its base pointer to address local variables in the stolen frame by setting it to the stolen frame on the victim's stack, while the thief's stack is used to allocate new frames in case of a call or fork. Since there are no frames allocated below a suspended frame, it can be resumed immediately when it gets ready, and the busy-leaves property is maintained. Therefore, the *physical* space consumption of extant leaves is bounded by PS_1 at any moment in time, because there are only P extant frames simultaneously and a leaf's stack can only grow S_1 deep. However, as explained in Section 2.2, the *virtual* space consumption is up to DPS_1 with D being the fork depth, because the stack ancestry of any leaf can be scattered over D linear stacks and any of the linear stacks could individually grow up to S_1 large at some moment in execution. Thus, physical space consumption can grow up to DPS_1, too, and the stack space consumption breaks the bound of Equation (2.2). Furthermore, as long as a linear stack contains any frames, it cannot be recycled, and the memory freed. As a result of the high virtual memory usage, a lot of swapping can occur, because unused portions of linear stacks are backed up by swap space, even though they do not contain any data [Lee+10]. Intel Cilk Plus [Rob13], the successor of Cilk and Cilk++, implements a strategy similar to this. To avoid breaking the space-bound, the worker have to take linear stacks from a pool with a limited amount. When the pool is empty, a worker cannot acquire a new stack and has to stop work-stealing and stall until another worker puts an unused stack back into the pool. By limiting the total amount of stacks, a strong space-bound is achieved, while *serial-parallel reciprocity* is maintained. However, the time-bound is sacrificed due to stalling, when the limit of available stacks is reached [YMC16].

As a solution to the high virtual address-space consumption and swapping of "new stack when needed", Lee et al. [Lee+10] suggest the unmapping of unused stack frames, so that they are no longer backed by swap space. Although, they suspect it might incur too much overhead since the stack space has to be remapped before being usable again. Instead, the possibility of a lazy stack frame reclaim is suggested. Yang and Mellor-Crummey [YMC16] further pursued this strategy. They implemented a fully library-based runtime with a greedy continuation-stealing scheduler, that works similarly to Cilk's randomized work-stealing scheduler. However, they implement fork and join as C macros and do not rely on compiler support. When a worker has a suspended frame on top of its stack and has to allocate a new stack in order to execute stolen tasks, it unmaps the unused portion of the stack from the page boundary below the suspended frame to the bottom end of the stack. They use the `madvise` Linux system call with the `MADV_DONTNEED` flag which, instructs the Operating System (OS) that pages can be freed and do not need to be backed up by swap space any longer, but that the mapping in virtual address space should remain. When a memory address in a region that was unmapped this way is reaccessed, a page fault is triggered, and the OS will allocate a new page for the accessed address. Therefore reclamation of stack space becomes a no-op and happens automatically. That way they achieve a strong space-bound of $S_P \leq P(S_1 + D)$ for *physical* memory, while *virtual* address space consumption can still be up to DPS_1, which they deem "practical" for 64-bit address spaces. Furthermore, their implementation is fully *serial-parallel reciprocal*, since they do not rely on a custom calling convention. They also claim the implementation can maintain the strong time-bound but omit a prove. Nevertheless, the results presented by Yang and Mellor-Crummey support this. They also show that the unmapping of stacks and the thereby incurred increase in page faults do not have a strong impact on performance.

The strategy "recycle ancestor stacks" is another variation of "new stack when needed". The difference is that when a worker starts work-stealing, it does not allocate a new stack right away. Instead, it checks blocked stacks first. If the suspended frame on top of a blocked stack is suspended at a join and an ancestor of the stolen frame, then the worker uses that stack to execute the stolen task, rather than a new stack. Since resuming the suspended frame requires completion of all descendants, which includes the stolen frame, the suspended frame cannot be resumed before the stolen frame is completed, and a collision on the stack cannot happen. This strategy can reduce space consumption greatly, but still does not hold a strong space-bound. Furthermore, the search for a usable stack introduces further overhead, weakening the time-bound [Lee+10].

A technique very similar to "recycle ancestor stacks" and "depth-restricted stealing" is called "leapfrogging". Originally suggested for implementing futures by Wagner and Calder [WC93], it can be used as a strategy to implement cactus stacks. Instead of searching for a reusable stack like with "recycle ancestor stacks", stealing is restricted to tasks that are descendants of the suspended frame on a worker's current stack. If such a task is found, it is executed on the worker's stack. That way, a collision of frames can be avoided. Since no new stacks are allocated at all, a strong space-bound can be achieved. However, stealing is restricted even more than with "depth-restricted stealing", resulting in a weak time-bound [YMC16].

A solution that uses memory mappings to implement cactus stacks was suggested by Lee et al. [Lee+10]. They implemented a runtime based on Cilk-5 and called it Cilk-M. It uses Thread-Local Memory Mappings (TLMMs) to simulate linear stacks for workers. Every worker sees only its linear stack, and all worker stacks are at the same location in virtual memory but can have individual, thread-local mappings to physical pages. The rest of the virtual address space is shared, making the area the stacks are mapped a TLMM area. When a worker steals a task, it has to access local variables in the stolen frame. In order to do so, it maps the physical page the stolen frame resides in and its ancestry pages, the pages further up in the stack, to its own stack. The pages get mapped to the same virtual addresses they had in the victim's stack. That way, the addresses of local variables remain the same, and pointers into frames are still valid. Since all stacks are in the same region in virtual address space, the thief has to unmap any physical pages currently mapped at these addresses, including pages containing suspended frames. Victim and thief then share the same physical pages and see the same stack ancestry. The thief can then start to execute the stolen frame. However, since victim and thief share the physical page the stolen frame resides in, the thief has to be cautious when it has to call or fork a function while executing the stolen frame. If victim and thief try to use the same space below the stolen frame in the shared page to allocate new frames, it would come to a collision. Therefore, the rest of the shared page is unusable for the thief, and it has to set its stack pointer to the next lower page boundary and map new pages below the stolen ones to allocate frames there. These new pages are private to the thief until he becomes the victim of another worker, and the pages get stolen, in which case they become shared pages. This leads to fragmentation in the stack, and some pages can contain as little as one frame, but space can be reclaimed after a join. The way Cilk-M implements the cactus stack prevents stacks with only one frame, and virtual and physical memory is conserved. Therefore, a strong time- and space-bound can be achieved, while *serial-parallel reciprocity* is maintained. However, Cilk-M needs an OS that supports TLMM. But, currently, there are no such OSs, so an implementation would have to use a custom, modified kernel, making it impractical for real-world applications. Another downside is that workers cannot see or access each other's stacks. Thus, stack-allocated shared data structures, such as MCS locks, do not work with Cilk-M [YMC16; Lee+10].

2.4 EMPER

EMPER is the microparallelism runtime used as the basis for the implementation of this thesis. EMPER is the acronym of Extensible Micro-Parallelism Experimentation Runtime. It is a research project. It is implemented using mostly high-level C++ language features with the purpose of being easily extensible to research different aspects of microparallelism. It supports testing different scheduling strategies, e.g., work-stealing with affinity hints to exploit data locality. Furthermore, it uses lock-free data structures and analyzes its effects on scaling.

In the following, the initial state of EMPER, before the implementation work of this thesis, will be analyzed to understand what this thesis is building on. EMPER already uses a greedy randomized work-stealing scheduler, but as mentioned above, it also supports other strategies. Its scheduler is lock- and wait-free through the usage of lock-free work-stealing dequeues and private semaphores. It uses child-stealing to implement asynchronous tasks but does not offer a direct fork-join interface. Instead, mapping fork-join onto asynchronous tasks and private semaphores can be used to create similar behavior. This could either be done in the form of user-defined functions or with compiler support, similar to Intel Cilk Plus. However, it is not possible to create dynamic parallelism this way.

Furthermore, when a task is blocked at a semaphore and has to wait for child tasks, the worker has to allocate a new stack to execute stolen tasks. The result is a behavior similar to a cactus stack implementation using the "new stack when needed" strategy. While this should hold a strong time-bound and has full *serial-parallel reciprocity*, the space-bound is not maintained. Additionally, child-stealing can result in unbounded space consumption for bookkeeping of created tasks. Overall, an implementation of fork-join parallelism with room for improvements. Nevertheless, the parallelism offered by EMPER in the form of asynchronous tasks works well for other parallel programming models.

ARCHITECTURE 3

This chapter shall give an overview off what this thesis tries to accomplish, in Section 3.1, the abstract components of the implementation and their relationships, in Section 3.2, and basic design used in Section 3.3.

3.1 Approach of this Thesis

The object of this thesis is to implement a fast, efficient, interoperable fork-join microparallelism runtime in C++, that uses only lock- and wait-free synchronization internally, and analyze its effect on scaling. Therefore, the implementation uses continuation-stealing for the benefits of lazy task creation and to avoid the potentially unbounded space consumption of bookkeeping of tasks. For work-stealing, the preexisting lock-free queues of EMPER's scheduler are used. The cactus stack implementation is based on the work of Yang and Mellor-Crummey [YMC16] since it can be used for a purely library-based runtime that does not rely on compiler support. Furthermore, it promises to be a practical solution to the cactus stack problem that fully solves the problem and does not require a custom kernel like the approach of Lee et al. [Lee+10]. However, the implementation of Yang and Mellor-Crummey [YMC16] uses locks to synchronize internal data-structures. Thus, lock- and wait-free algorithms are used instead.

3.2 Components

In order to achieve the desired behavior and to build the cactus stack, the implementation needs some components to manage the state and coordinate interactions between workers. Each function instance, that forks off child functions, has an associated continuation to save and restore the state of that function instance. Continuations also keep track of child function instances, have a reference to the stack on which the frame of the function instance is located, and take the role of tasks, which are pushed onto or are popped and stolen from work-stealing dequeues. The scheduler has a work-stealing dequeue per worker. Workers can invoke the scheduler to push tasks to or pop them from their work-stealing dequeue. In case a worker runs out of work, the scheduler performs randomized work-stealing. When a task is stolen, the victim can call the scheduler and try to resume the stolen task. The scheduler then checks if the continuation can be resumed, if the victim is the last child of the continuation, and resumes the continuation or starts work-stealing to find a new task. Each worker has one stack or context as they are called in EMPER, to execute tasks on. When a worker needs a new stack, because its current one is blocked by a suspended task, the worker can invoke the context manager to allocate a new context.

3.3 Basic Design

The basic design of this implementation is very similar to that of Fibril [YMC16], using continuation-stealing to build a cactus stack following the “new stack when needed” strategy. The key building blocks are fork, join, work-stealing and managing of stacks. In the following, these four parts will be explained separately.

3.3.1 Forking

The purpose of fork is to create the necessary conditions for parallelism to happen. Therefore, it has to do two things: create a task that can run concurrently and scheduling it, so that it can be stolen. Since the continuation of a frame is a task and running the task means resuming execution of the continuation, the state of the continuation has to be updated at a fork. More precisely, the pointer to the instruction of the function, where execution has to continue, needs to be updated to the next instruction following the fork. Then the continuation is pushed to the bottom of the executing worker’s work-stealing dequeue, where it can be stolen by other workers. With this, all preparations are done, and the worker can now start to execute the child function that should be forked off. The worker can call the child function normally and use its linear stack to allocate a stack frame. If another worker steals the continuation, parent and child will execute in parallel.

When the worker that encountered the fork and executed the child function returns from the child function, it tries to pop the bottommost task from its work-stealing dequeue, in order to check if the continuation was stolen. If the popping was successful, the worker can be sure that the continuation was not stolen, because no worker can push tasks to any other work-stealing dequeue than its own. In this case, the worker can continue the execution of the parent function instance normally, making the fork semantically equivalent to a function call. However, if the continuation was stolen, the worker cannot continue normally, because another worker already continued execution of the parent. The worker can then try to resume the continuation. The worker has to check if the continuation is waiting at a join, and if the worker’s child task was the last one to finish. If both conditions are met, the worker can restore the state saved in the continuation and continue the execution of the function instance following the join. If the conditions are not met, that is one or more other child tasks are still in execution or the parent task has not reached a join yet, the continuation cannot be resumed. In this case, the worker has run out of work, since the popping of the bottommost task can only fail if the dequeue is empty. The worker then has to start work-stealing itself to find new work.

3.3.2 Joining

A join statement is a synchronization point between parent and concurrent child tasks. Therefore, when a worker encounters a join, it has to check if the continuation of the parent task was stolen at a fork. If the continuation was never stolen, then the worker that reached the join has already finished executing all child tasks itself and can continue execution after the join. However, if the continuation was stolen, the worker has to try to resume the continuation. Similar to the case of fork, when the continuation is stolen, the worker can only resume the continuation if all child tasks have finished. If that is the case, the worker can resume the continuation and continue execution after the join. Otherwise, the worker has to start work-stealing to find new work.

3.3.3 Running stolen Tasks

When a worker successfully steals a task from another worker, it has to run the task. That means the worker has to restore the saved state of the continuation and start executing the function at the saved position, after a fork. Since the function instance's stack frame is located on another worker's stack and the other worker is using its stack to execute the forked child function, the thief cannot use the stack of the continuation. However, the worker has to access the stack frame of the function instance to read and write local variables when executing the function instance. Therefore, the thief has to use its own stack to allocate stack frames, if the stolen function instance calls or forks more child functions, and use a pointer, saved in the continuation, that points to the stack frame, to access its local data.

3.3.4 Managing Stacks

In order to properly build the cactus stack and avoid collisions, workers have to switch to a new stack or switch back to an old stack at the right moment in execution. A new stack is necessary when a worker has a suspended frame on top of its stack and has to start work-stealing, as explained in Section 2.1.5. Conversely, when a suspended frame is resumed, the worker that continues execution of that frame needs to switch to the stack that contains the suspended frame. Since every worker can have only one stack at a time, the worker has to free its current stack before switching to the one with the blocked frame. This is safe because the current stack of the worker is always empty at that point.

The point at which workers might switch stacks is always when trying to resume a continuation when a worker returns from a fork, and finds its continuation to be stolen, or when a worker encounters a join. If a worker cannot resume the continuation, it has to check whether its stack is the stack the continuation points to. If the continuation has a reference to another stack than the worker's current stack, the worker does not have to switch stacks and can start work-stealing. Otherwise, if the continuation points to the worker's current stack, the worker has to allocate a new stack before it can steal work and execute new tasks. Additionally, the worker has to unmap unused pages of the continuation's stack.

However, if a worker can resume a continuation, the opposite is the case. If the continuation points to the worker's stack, the worker is already on the correct stack and can immediately resume execution. Otherwise, the worker has to free its current stack and switch to stack the continuation points to before it can resume the continuation.

4 IMPLEMENTATION

The fork-join implementation of this thesis is designed to work on hardware with x86-64 (AMD64) Instruction Set Architecture (ISA), using the standard System V Architecture Binary Interface (ABI) [Mat+14] for the AMD64 architecture that defines the calling convention and stack frame layout. However, a similar approach could also work on different architectures.

4.1 Fibril

This section introduces the continuation-stealing cactus stack implementation of this thesis. EMPER is extended with a class Fibril, named after the runtime Fibril [YMC16], which it is based on, that implements the fork-join Application Programing Interface (API) with continuation-stealing. Listing 4.6 shows the base structure of the Fibril class and its interface. An object of type Fibril functions as a task that can be scheduled and pushed to a worker's work-stealing dequeue and can be stolen and executed consequently. It has a member of the type Continuation that implements the actual continuation behavior. A continuation is always tied to a function instance and its stack frame, it is the continuation of that function instance. Every function instance, that forks off child functions, needs its own Fibril object. The constructor of Fibril invokes the constructor of its continuation. The `fork()` and `join()` methods belong to the API of EMPER, while `run()` and `tryResume()` are called internally by the scheduler. In the following sections, the code of Fibril will get extended stepwise until all major implementation details are explained.

```
class Fibril {
    Continuation continuation;

    inline Fibril() : continuation() { /* ... */ }

    void run() { /* ... */ }

    void fork(/* ... */) { /* ... */}

    void join() { /* ... */ }

    void tryResume() { /* ... */ }
}
```

Listing 4.6 – Basic members of the `Fibril` class.

4.2 Continuations

The Continuation class implements the hardware-specific logic that allows saving the state of a function instance and restoring it at a later point to resume execution of the function instance. The state of a function instance consists of the values of all arguments and local variables and the point in machine code where execution should continue. A function instance keeps its local variables on the stack, within its stack frame, or in registers. Since the location of a stack frame in memory should not be changed, as explained in Section 2.2, it is sufficient to save all values held in registers in the stack frame and then save a pointer to the stack frame, in order to preserve the state of local variables.

4.2.1 x86-64 - Calling Convention and Stack Layout

To build continuations that also support building a cactus stack, it is useful to understand first how the calling convention works and how the resulting layout of a stack frame is structured. The x86-64 architecture has 16 64-bit wide general-purpose registers. When a function calls another function, half of the general-purpose registers, the scratch registers, `rax`, `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, `r10` and `r11`, have to be saved by the caller, while the callee preserves the other half, `rbx`, `rsp`, `rbp`, `r12`, `r13`, `r14`, and `r15`. The `rsp` register is used as the stack pointer and `rbp` as the base pointer of a stack frame. Furthermore, the registers `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9` are used to pass arguments to functions. These registers pass the first six function arguments, in the order, they are listed, `rdi` passing the first and `r9` passing the sixth argument. Further arguments are passed on the stack in reverse order. If a function wants to return a value, it can pass it back to the caller in the `rax` register.

Figure 4.1 illustrates how the stack grows downwards on an x86-64 machine when a function `a()` calls a function `b()` that accepts eight arguments. Initially, there is only the stack frame of function `a()` on the stack. It contains a return address and the saved base pointer of the caller function, `rbp`. The base pointer register `rbp` points to the saved `rbp` value of the caller and marks the base of the stack frame, the top end of the stack frame's variable size area where local variables are stored. The stack pointer register `rsp` points to the bottom end of the stack frame. Before the actual call to `b()` happens, the caller has to save caller saved registers. It writes the values of live variables, held in registers, back to the variables inside its stack frame. Then, to pass the eight arguments to the callee, `a()` first moves the first six arguments the registers `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`, before pushing the remaining two arguments, `b() arg7` and `b() arg8`, in reverse order onto the stack. Next, the call instruction is executed. It pushes the return address, the address of the next instruction following the call instruction, `ret a()`, onto the stack, and sets the instruction pointer `rip` to the first instruction in the code of function `b()`. Since the callee has to build its stack frame and needs to use the `rbp` register to point to the new stack frame, it has to save the current value of `rbp` by pushing it onto the stack, because `rbp` is a callee saved register. It then moves the value of `rsp` to `rbp`, before subtracting an offset from `rsp`, so `rsp` points further down on the stack, in order to reserve the space for the local variables of `b()` on the stack. With this, the stack frame of `b()` is fully build up, and the function can start executing its code. To address local variables, it can use the base pointer with a negative offset, and to address the arguments on the stack, `rbp` with a positive offset of 16 bytes or more, has to be used.

When function `b()` wants to return, it has to dismantle its stack frame first. It sets the `rsp` to `rbp` and can then pop the saved `rbp` of the caller function back from the stack into the `rbp` register. The stack pointer `rsp` points the return address, and the `ret` instruction can be executed. The `ret` instruction pops the return address, pointed to by `rsp`, from the stack into the instruction pointer

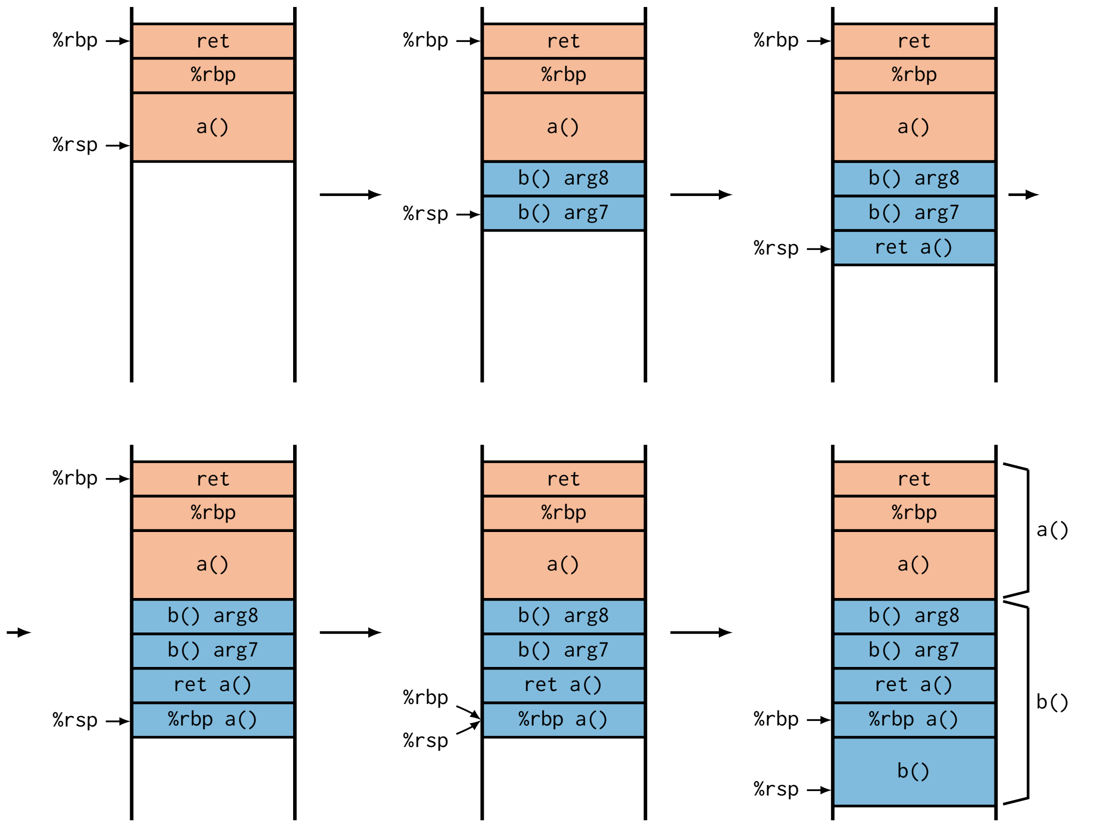

Figure 4.1 – Example of a linear stack on x86-64.

register `rip`. Execution continues in the caller function. Function `a()` has to remove the arguments pushed onto the stack by adding an offset to `rsp`. The callee's stack frame is now completely cleaned up, and the function `a()` can restore values from its stack frame to registers and continue execution [Mat+14].

These processes can be exploited to build continuations and a cactus stack without having dedicated compiler support. In order to save the state of a stack frame, a continuation has to back up the stack and the base pointer of the stack frame. If the continuation can ensure that the compiler generates code to write back all values kept in registers when saving the state of a function instance, saving `rsp` and `rbp` is sufficient to restore the state later on. Since the base pointer of a stack frame does not change, a continuation can save a copy of it one time, when it is created, and reuse it over its lifetime. The same[7] applies to the stack pointer. The only thing else that has to be saved to restore the state of a function instance is an instruction pointer to the instruction in the function's code, where execution should continue when the continuation is resumed. The saved instruction pointer has to be updated every time the continuation is saved since a function can be saved and restored at different points.

Since the compiler already generates code to save the caller saved registers before a function call, function calls can be used as the basis to implement the save operation of a continuation. The

[7]The stack pointer changes when arguments are passed to a function, but returns to its original position after the call returns. It can only change when the function uses variable length arrays and the size of one array changes after the continuation has been initialized. However, the use of variable size arrays is disadvised.

continuation only needs to instruct the compiler to also generate code that saves the callee saved registers before the function call, too. Furthermore, the call instruction pushes the return address onto the stack. Since it is a pointer to the next instruction following the call instruction, it can be used as the saved instruction pointer. Additionally, in order to build a cactus stack, a worker needs to be able to access the local data of an function instance, when resuming a continuation, but use a different stack to allocate new stack frames when calling or forking functions. Since local variables are addressed by the base pointer, while pushing arguments and calling functions uses the stack pointer, this behavior can be created by having a worker, that resumes a continuation, set its base pointer to the value saved in the continuation, to point to the stack frame, and set its stack pointer to the top of its stack, to allocate stack frames there.

4.2.2 The `fibril` keyword

If the size of a stack frame is fixed, the value of `rsp` does not change over the lifetime of the stack frame. In that case, it is possible to address local variables and arguments via the stack pointer, rather than the base pointer, because the relative position between stack pointer and variables does not change. Therefore, when the compiler knows the exact size of a stack frame and that its size is fixed, the compiler often decides to omit the base pointer and use the stack pointer to address local variables and arguments as an optimization. Doing so gives the compiler one more general-purpose register for holding the values of local variables or the overhead of saving and restoring the value of `rbp` can be saved if the function does not use the register.

However, since workers require that local variables of a function that use continuations to fork are addressed via the base pointer for building the cactus stack, this optimization would prevent this approach to build a cactus stack. Therefore, the compiler must be instructed not to use this optimization for functions that fork off other functions. The GNU Compiler Collection (GCC) [Gcc] offers a function attribute `optimize` that allows specifying optimizations a function should be compiled with. These optimizations can then differ from the optimizations used for the rest of the compilation unit. With this function attribute, the compiler can be instructed to use or not use the optimization of omitting the base pointer. EMPER uses a preprocessor macro to define the `fibril` keyword, as seen in Listing 4.7, that defines the `optimize` function attribute with the string value `no-omit-frame-pointer`. This attribute instructs the compiler to use the frame pointer to address variables and arguments for the function it is used on. Therefore, the `fibril` keyword has to be used for all functions that use continuations to fork off child functions.

```
#define fibril __attribute__((optimize("no-omit-frame-pointer")))
```

Listing 4.7 – The `fibril` macro.

Nevertheless, there are cases where the compiler generates code to address local variables via the stack pointer, despite the use of the `fibril` keyword. One such example is aligned objects that are stored as a local variable inside the stack frame. Since a stack frame is not necessarily aligned properly, the compiler has to generate code that aligns the stack pointer and positions the object relative to the stack pointer on the stack. Therefore, such objects can not be stored on the stack in functions that fork.

4.2.3 The `membar()` Memory-Barrier

Since the compiler generates code for function calls that only saves the caller saved registers, the compiler needs to be also instructed to save the callee saved registers when saving the state of a function instance at a fork. Furthermore, fork needs to be a memory barrier for the compiler, so that no operations can be reordered over the fork, and all writes are issued before the fork happens. EMPER uses the `membar()` macro, shown in Listing 4.8, that functions as a memory barrier for the compiler. The macro takes a function call as the argument. The inline assembler statement after the call clobbers the callee saved registers and `memory`, causing the compiler to assume these registers will be used, and their values will not be preserved by the statement. Therefore, the compiler has to create code that saves the registers and restores the values afterward. Since the call is the only other statement in the scope, the compiler saves the registers at the beginning of the scope, before the call statement, and restores the values afterward[8], resulting in a call that preserves the caller and callee saved registers. Furthermore, clobbering of `memory` acts as a read and write memory barrier. It tells the compiler that the assembler statement will read or write memory locations other than the input, output, or clobbered registers. To preserve consistency, the compiler has to write back all values hold in registers to memory. The compiler also has to assume that the values might have changed after the statement, so if values from such variables are needed, they have to be loaded again from memory. Effectively, this instructs the compiler to generate a function call that serves as a memory barrier and saves all registers. A function call with this macro can be used as the basis to implement continuations and thus fork.

```
#define membar(call) \
do { \
    call; \
    asm(    "nop" : : : "rbx", "r12", "r13", "r14", "r15", "memory" ); \
} while (0);
```

Listing 4.8 – The `membar()` macro.

4.2.4 The `Continuation` Class

The `Continuation` class implements the core functionality for continuations of a stack frame that can stop execution and save the state of a function and restore it at a later point and resume execution at the same position in the function. Listing 4.9 shows the code of this class, used to implement fork and join in the `Fibril` class. It can also be used to implement functionality similar to the `setjmp()` and `longjmp()` functions of the C standard library.

[8]The code for this macro was copied from the source files of Fibril [YMC16]. It provides the described behavior. However, it seems to be unspecified whether registers are saved before the function call or just before the **`asm`** statement (after the call) [Gcc].

```
class Continuation {
    void* bp;
    void* sp;
    void* ip;

    inline Continuation() {
        register void* rbp asm("rbp");
        register void* rsp asm("rsp");
        bp = rbp;
        sp = rsp;
        ip = nullptr;
    }

    void execute(void* rsp) {
        asm(    "mov %0, %%rsp \n\t"
                "mov %1, %%rbp \n\t"
                "jmp *%2 \n\t"
                : : "r" (rsp), "r" (bp), "r" (ip) : "memory");
    }

    /* ... */
}
```

Listing 4.9 – The `Continuation` class.

The class has three fields `bp`, `sp`, and `ip`, which store the saved base pointer, stack pointer and instruction pointer. The constructor initializes the base and stack pointer with the values of the current stack frame. By using the **`register`** keyword with an inline assembler statement, the compiler can be instructed to place a variable in a specific register. When the variable is read without writing it first, the last value of the register can be accessed. Doing this with the `rbp` and `rsp` registers, the base and stack pointer, allows reading the values of the current stack frame and initializing the respective class members. Since the continuation has to save the stack frame of a function and not of the constructor itself, it has to be marked as inline, or else the compiler could decide to make the constructor a dedicated function with its own stack frame. That would initialize the members with the wrong values.

The `execute()` function is used to resume the continuation. It loads the base pointer register `rbp` and the stack pointer register `rsp`, before jumping to the position pointed to by the saved instruction pointer `ip`. The base pointer is set to the saved value in `bp`. However, to support building a cactus stack, the function accepts a value for the stack pointer as the argument. That way, a worker can pass a pointer to the top of its own stack, rather than the stack pointer saved in the continuation. If a worker needs to load `rsp` with the saved value `sp`, it can pass the value to the function as the argument.

Since the value of `ip` has to be a pointer to the instruction where execution should continue, it is only initialized with the **`nullptr`** in the constructor. It must be set to a proper value for each point in a function where execution should be continued later. Therefore, the `Fibril` class that uses the continuation has to set the value for every fork and join.

4.2.5 Building the Cactus Stack

The `Fibril` class combines the `membar()` macro with the `Continuation` class to implement the `fork()` and `join()` methods that can be called by the user and the `run()` method that is invoked by EMPER internally to execute stolen tasks. Listing 4.10 shows the code for the basic fork-join functionality. Since the constructor calls the constructor of `Continuation`, that needs to be inlined in the forking function, `Fibril`'s constructor also needs to be inlined, as described in the previous subsection. The `fork()` function is only a wrapper that uses the `membar()` macro to call the actual fork function `forkImp()`, a C++ lambda expression, in a way that saves all registers and works as a memory barrier. `fork()` can be inlined as well, to save one function call and improve performance. `forkImp()` first sets the saved instruction pointer in the continuation to the next instruction after the call. It uses a GCC builtin-function that reads the return address of its stack frame since the return address points exactly to the instruction following the call in the caller. Therefore, it may not be inlined and uses the function attribute `noinline`. This enables a potential thief of the continuation to resume the function instance at the correct position following the fork. To enable a thief to steal the continuation, the `Fibril` object has to be scheduled. The scheduler is invoked and pushes a pointer to the object to the bottom of the executing worker's work-stealing dequeue, where it can be stolen by other workers. After that, the function that should be forked off, `func`, which is passed to the `forkImp()` function as an argument, can be executed.

When `func` returns, the executing worker has to try to pop the `Fibril` object off the bottom of its work-stealing dequeue to check whether it was stolen by another worker. If it was not stolen and the worker succeeds, it can return from the `forkImp()` function and thereby resume execution of the function instance that invoked the fork. However, if the object was stolen and the attempt to pop it failed, the worker cannot return, since another worker already resumed execution at the point where the return address points to. The worker then has to try to resume execution by calling `resume()` method of the scheduler with the `Fibril` object as the argument. The scheduler then tries to resume the object by invoking its `tryResume()` method, which will be explained in Section 4.3 and Section 4.4.

```
class Fibril {
    Continuation continuation;

    inilne Fibril() : continuation() { /* ... */ }

    void run() {
        continuation.execute(worker.tos);
    }

    inline void fork(/* func, ... */) {
        auto forkImp = [](Fibril *fr, /* func, ... */) __attribute__((noinline)) {
            fr->continuation.ip = __builtin_return_address();
            scheduler.push(fr);
            func(/* ... */);
            if (! scheduler.pop())
                scheduler.resume(fr);
        };

        membar(forkImp(this, /* func, ... */ ));
    }

    void join() {
        if (/* this was not stolen */)
            return;

        auto joinImp = [](Fibril *fr) __attribute__((noinline)) {
            fr->continuation.ip = __builtin_return_address();
            scheduler.resume(fr);
        };

        membar(joinImp(this));
    }

    /* ... */
}
```

Listing 4.10 – The basic `fork()` and `join()` implementation of the `Fibril` class.

The `run()` method is invoked by a thief that stole the `Fibril` object. It calls the `execute()` method of the continuation with a pointer to the top of the thief's stack to resume the execution of the stolen task. The worker's stack pointer register `rsp` is set to the top of its own stack and its base pointer register `rbp` is set to the value saved in the continuation, the base of the stack frame on the victim's stack.

The `join()` method serves as a wrapper function similar to `fork()`, that uses the `membar()` macro to call `joinImp()`. If the continuation was stolen at a previous fork in the caller function instance, then `joinImp()` has to synchronize with child tasks and wait for all child tasks to finish. In order to do so, it has to try to resume the execution in the same way `forkImp()` does when the `Fibril` object was stolen. If it was not stolen, then there cannot be any child task, and thus, join can return immediately.

```
fibril int foo()
{
    int i;
    Fibril frame;
    frame.fork(a);
    frame.fork(b);
    c();
    frame.join();
    return;
}
```

Listing 4.11 – The example function `foo()`.

To illustrate how the class works, Listing 4.11 shows a simple function that uses fork and join, and Figure 4.2 shows the corresponding stack layouts. The `foo()` function uses the `fibril` keyword to ensure the generated code uses the base pointer to address local variables. Initially the worker W_1 is executing the function. It has the stack frame of the function on its stack. The dummy variable `i` and the `Fibril` object `frame` are stored inside the stack frame. The initial state is identical to a normal, linear stack. The base pointer `rbp` of W_1 points to the base of the stack frame and the stack pointer `rsp`, to the end. When W_1 starts to execute the fork of function `a()`, it first saves the values held in registers onto the stack. Then the actual fork function, `forkImp()`, is called, and its stack frame is allocated. After the continuation is pushed into the worker's work-stealing dequeue, the function calls `a()`.

When worker W_2 steals the continuation and resumes it, the base pointer register of W_2 `rbp` is set to point to the stack frame of the function instance of `foo()` on the stack of W_1. The stack pointer of W_2 points to the top of its stack, leaving some space as linkage region, as explained in Section 4.7. W_2 can then start executing `foo()` at the first instruction after the fork of `a()`. Since the next statement is the fork of function `b()`, W_2 has to do the same steps as W_1 did before. First, W_2 saves its registers, because of the `membar()` call, then the stack frame of the `forkImp()` is allocated on the stack of W_2. The remaining steps are identical to what W_1 did, updating the `rip` of the continuation, pushing the `Fibril` object to W_2's work-stealing dequeue, and executing function `b()`. Both forks use the same continuation but have to update it and push it into different work-stealing dequeues.

The continuation can now be stolen again. In this example, the worker W_3 steals the continuation from W_2 and executes it. Again, W_3's base pointer `rbp` is set to point to the stack frame of `foo()` on W_1's stack, while the stack pointer of W_3 `rsp`, points to its own stack. W_3 will then call function `c()` normally, allocating the stack frame on its own stack. Function `foo()` has now three workers executing child tasks.

4.3 Bookkeeping of Child-Tasks

Continuations save the state of a function instance. Therefore, a function instance may not return before all child tasks have finished. This also conforms to strict fork-join parallelism. To prevent a function instance from returning premature, it has to synchronize with its child tasks by using a join statement after forking, before a return statement. Since EMPER uses a greedy scheduler, `Fibril` objects have to keep track of child tasks, so that the last worker to finish its work can resume the continuation at a synchronization point.

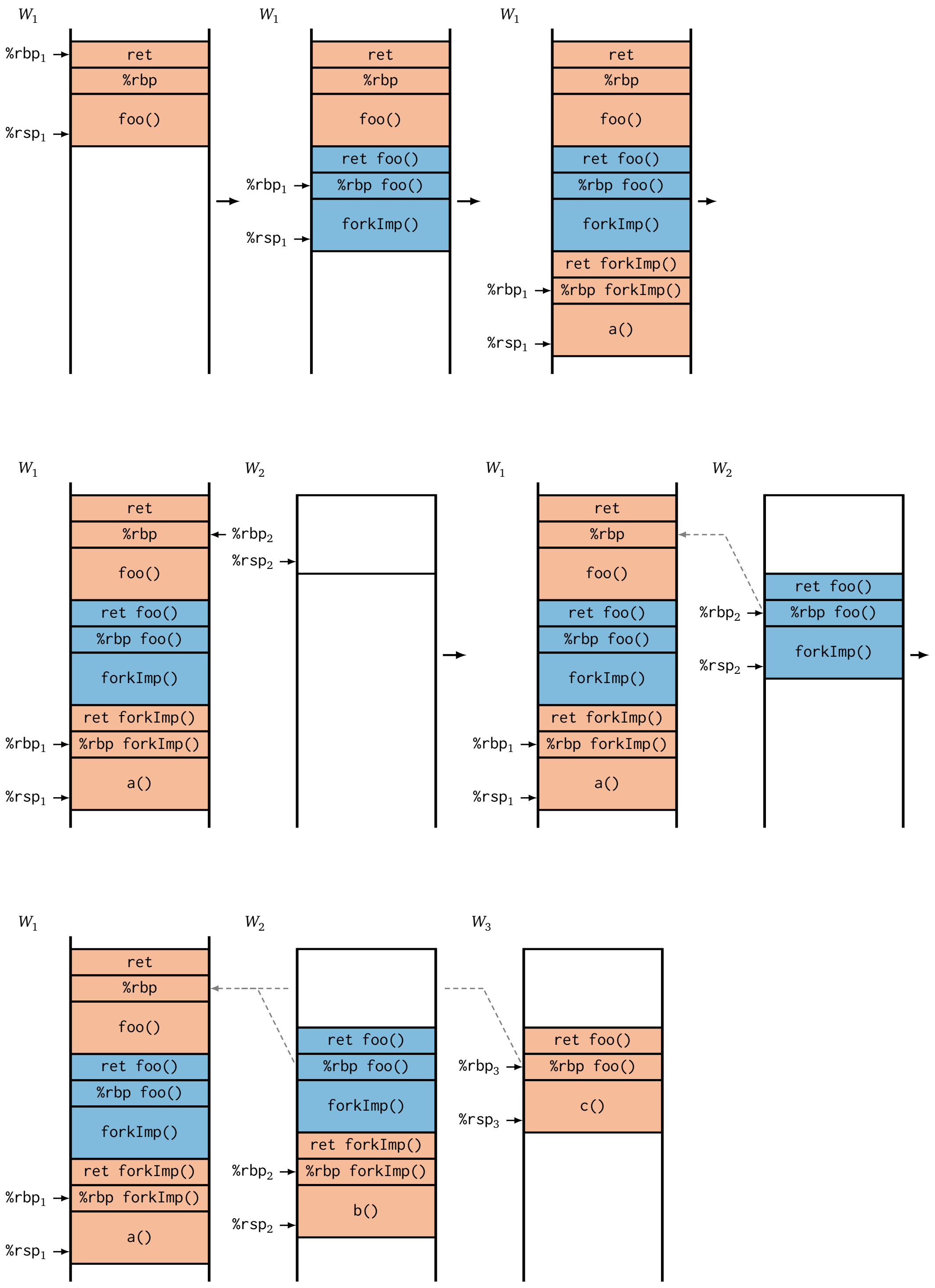

Figure 4.2 – Example of a cactus stack.

```
class Fibril {
    Continuation continuation;
    int activeChildren = 0;

    void run() {
        if (activeChildren == 0)
            activeChildren = 2;
        else
            activeChildren++;
        continuation.execute(worker.tos);
    }

    void join() {
        if (activeChildren == 0)
            return;
        /* ... */
    }

    void tryResume() {
        if (--activeChildren > 0) {
            return; /* start work-stealing */
        }
        /* resume continuation */
    }

    /* ... */
}
```

Listing 4.12 – The active children counter of the `Fibril` class.

Listing 4.12 shows the implementation of `Fibril` with a counter variable, `activeChildren`, of type **`int`** for active children. Initially, the counter is zero. When a task is stolen and the `run()` method is invoked, the counter is incremented before the continuation is executed. If the counter is still zero, the thief is the first to steal the `Fibril` object. In this case, the thief has to increment the counter by two, one time for itself and one time for the victim, since there are two workers executing. Otherwise, the thief only increments the counter by one, accounting for itself.

When a worker encounters a join, it first checks if the counter is still zero. In that case, no stealing happened, and the worker can continue executing. However, if the counter shows a value greater than zero, at least one steal occurred, and the worker has to try to resume the continuation. Similar to when a worker notices that the continuation was stolen at a fork, it invokes the scheduler that tries to resume the continuation by calling the `tryRsume()` method. This method decrements the active children counter by one. Then it checks if the counter has become zero again. If the counter is zero, the worker was the last to finish its work and can, therefore, resume the continuation. Otherwise, if the counter is still greater than zero, there are still active children. In that case the method returns, and the scheduler starts work-stealing to find a new task for the worker.

4.4 Context Management

In Section 3.3.4, the basics of how this implementation has to handle stacks or contexts, as they are referred to in EMPER, are explained. Listing 4.13 shows the corresponding implementation details. The `Fibril` class requires a reference to the context the saved stack frame is located on. Therefore, in the constructor, the context reference gets initialized with the pointer to the current context of the executing worker. In the `tryResuem()` method, the switching of workers' contexts can happen. If a worker was executing the last active task and the children counter reaches zero, the worker has to check whether its current context is the one the stack frame is located on. If that is the case, the worker can resume the suspended stack frame immediately. However, if the worker's context is not the context the stack frame is located on, the worker has to free its current context by handing it back to the context manager and make the context with the suspended frame its new context. That is necessary, so workers always maintain a correct reference to their working context. In order to resume the continuation, the worker invokes the `execute()` method of the continuation and passes the saved stack pointer `rsp` as an argument. After the method has finished, the base and stack pointer point to the same stack frame, and the worker can continue to execute the function instance normally. The branching of the cactus stack for this function instance and stack frame is thereby reversed.

Otherwise, if the worker is not the last and cannot resume the continuation, it has to check whether its current context is the stack the suspended frame is located on, to see if it has to switch contexts. If the worker's context is not the context the suspended frame is located on, the worker has nothing to do and can return from the method immediately. If the contexts are the same, the worker has to switch to a new context, since another worker will resume the context later and make it its context. Furthermore, it has to unmap the unused pages from the next page boundary below the suspended stack frame, pointed to by the saved stack pointer of the continuation, to the bottom end of the stack. In order to do so, the `unmap()` method of the context manager is called with the stack to unmap and the stack pointer, saved in the continuation, as arguments. The context manager uses the `madvise()` system call with the `MADV_FREE` flag to instruct the OS to unmap the physical pages, but keep the mapping in the virtual address space. Fibril [YMC16] used the `MADV_DONTNEED` flag that caused the OS to unmap the physical pages immediately, since `MADV_FREE` was not available, yet, at the time of publication. The `MADV_FREE` flag allows the OS to free the physical pages lazily at a later point if memory is needed elsewhere. This reduces unnecessary un- and remapping and should improve performance. If a page is accessed again, before it is unmapped, the OS will mark the page as being in use again and will not free it subsequently. When a page was unmapped by the OS and is accessed again afterward, a page fault is triggered, and the OS will map a new physical page to the page frame that was accessed. Finally, after unmapping the unused pages of the context, the worker allocates a new context to execute stolen tasks from the context manager, before the method returns to the scheduler.

```
class Fibril {
    Continuation continuation;
    int activeChildren = 0;
    void* context;

    inilne Fibril() : continuation() {
        context = worker.context;
    }

    void tryResume() {
        if (--activeChildren > 0) {
            if (context == worker.context) {
                contextManager.unmap(worker.context, continuation.rsp);
                worker.context = contextManager.alloc();
            }
            return; /* start work-stealing */
        }
        if (context != worker.context) {
            contextManager.free(worker.context);
            worker.context = context;
        }
        continuation.execute(continuation.rsp); /* resume */
    }

    /* ... */
}
```

Listing 4.13 – The context management of the `Fibril` class.

In the examples of Listing 4.11 and Figure 4.2, there can be multiple different scenarios of how the stack layouts change, depending on which steals happen and which worker returns last. If the continuation was never stolen from worker W_1, it would return from the fork of function `a()` like from a normal function call and could then fork off `b()` and call `c()` itself, and no parallelism would occur. If the continuation is only stolen once by W_2, there are different outcomes, depending on which worker finishes its work first. If W_2 returns from the fork of `b()`, and the continuation was not stolen from W_2, it will call `c()` and eventually return and call the `join()` method of the `Fibril` object. When W_1 is still executing `a()`, W_2 cannot resume the continuation. Since the now suspended stack frame of `foo()` is not located on W_2's context, the worker can start work-stealing and execute the next stolen task on its context. Figure 4.3 shows how the stack layouts change when worker W_1 returns from the fork of `a()` before W_2 reaches the join. First, W_1 returns from `a()` and finds the continuation was stolen and calls the scheduler, which calls the `tryResume()` method. Since W_2 has not decremented the counter of active children, W_1 cannot resume the continuation and has to start work-stealing. However, since the suspended task is located on W_1's context, W_1's context is the same context as pointed to by the `Fibril` object, W_1 has to unmap the unused pages on the stack and allocate a new context to execute stolen tasks on. When W_2 eventually reaches the join, it will be able to decrement the counter to zero and resume the suspended stack frame. It will unmap its current context and set it to the context with the suspended frame and then resume execution by setting its stack and frame pointer to the stack frame of `foo()`.

There are other possible scenarios. Worker W_1 could steal the continuation back from W_2 and execute function `c()`. Depending on which worker finishes last, W_1 could switch back to its initial context. Otherwise, W_2 will switch to the context with the suspended frame, as explained above. Continuing from the final situation of Figure 4.2 with three workers, there are even more potential outcomes, depending on the order in which the workers finish their tasks. In every case, the last worker to finish will resume `foo()` and possibly change the worker's context.

4.5 Scheduler Context

The previous sections explained the main functionality of the `Fibril` class. However, there is an important topic that was left out so far: Race conditions and synchronization. This section will explain race conditions resulting from the context management, introduced in Section 4.4, and how to solve them. The more general, but also more complex, synchronization of data races that occur in conjunction with shared variables and data structures will be discussed in Section 4.6.

When a worker is executing the `tryResume()` method, there are two situations where the worker has to switch its working context. Both context switches form a critical section, whereby two workers can end up racing for a given context. In both situations, the fundamental problem is that a worker is still using its context for execution of scheduling and the `tryResume()` method, while another worker already switches to the same context, producing collisions on the stack with undefined behavior. The first situation occurs when a worker has to free its current context in order to switch to the context of the continuation with the suspended stack frame. If the worker frees its context while it is still using the context, various things can happen. The context manager could give the context to another worker, that wants to allocate a new context, and start executing on the context, while it is still in use, or the context manager could hand it back to the memory allocator, which could unmap the memory region of the context. In both cases, undefined behavior and crashes will be the result. The other situation is when a worker has to switch to a new context because the current context has a suspended stack frame blocking it. The situation occurs when the owner of a stack is not the last one to finish its work. Then another worker will be the last one, the one to resume the continuation and switch to the context with the suspended frame. Since the owner of the stack has already decremented the counter, the resuming worker can switch to the stack as soon as it finishes its work, this includes the time before the owner of the context has switched to a new context. Again, possibly resulting in a collision on the stack with undefined behavior.

In both scenarios, the underlying problem is that a worker is making a context available for another worker or freeing it, while the worker is still using the stack, having the stack frames of function instances in execution located on the stack. A simple solution to this problem is to move to neutral ground before executing the critical section. In the case of EMPER, that means splitting the address space into an *application space* and a *scheduler space*. The application space consists of the working contexts that are used to allocate stack frames for application code that uses EMPER, while the scheduler space consists of the workers' stacks, the stacks of the pthreads that serve as workers. At a fork or join, before the worker invokes the scheduler and the `tryResume()` method, it has to switch back to the scheduler space and when a worker resumes a continuation at a join or after stealing it, the worker switches to the application space. The switch to the application space happens automatically since the worker sets its base and stack pointer to application contexts. In order to switch back to the scheduler space, the `Continuation` class is used. It implements the `setJmp()` and `longJmp()` methods, as shown in Listing 4.14. At the beginning of the workers' main routine, before the worker starts work-stealing and executing tasks, a `Continuation` object is instanced, and the `setJmp()` method is invoked to create an entry point. The scheduler's `resume()` method that is called by fork, if the continuation was stolen, and by join, as shown in Listing 4.10, calls the `longJmp()` method of the worker's continuation to switch back to the worker's main routine, on the worker's stack. The worker then tries to resume the suspended `Fibril` object by calling the object's `tryResume()` method. In case the resume is successful, the worker switches back to the application

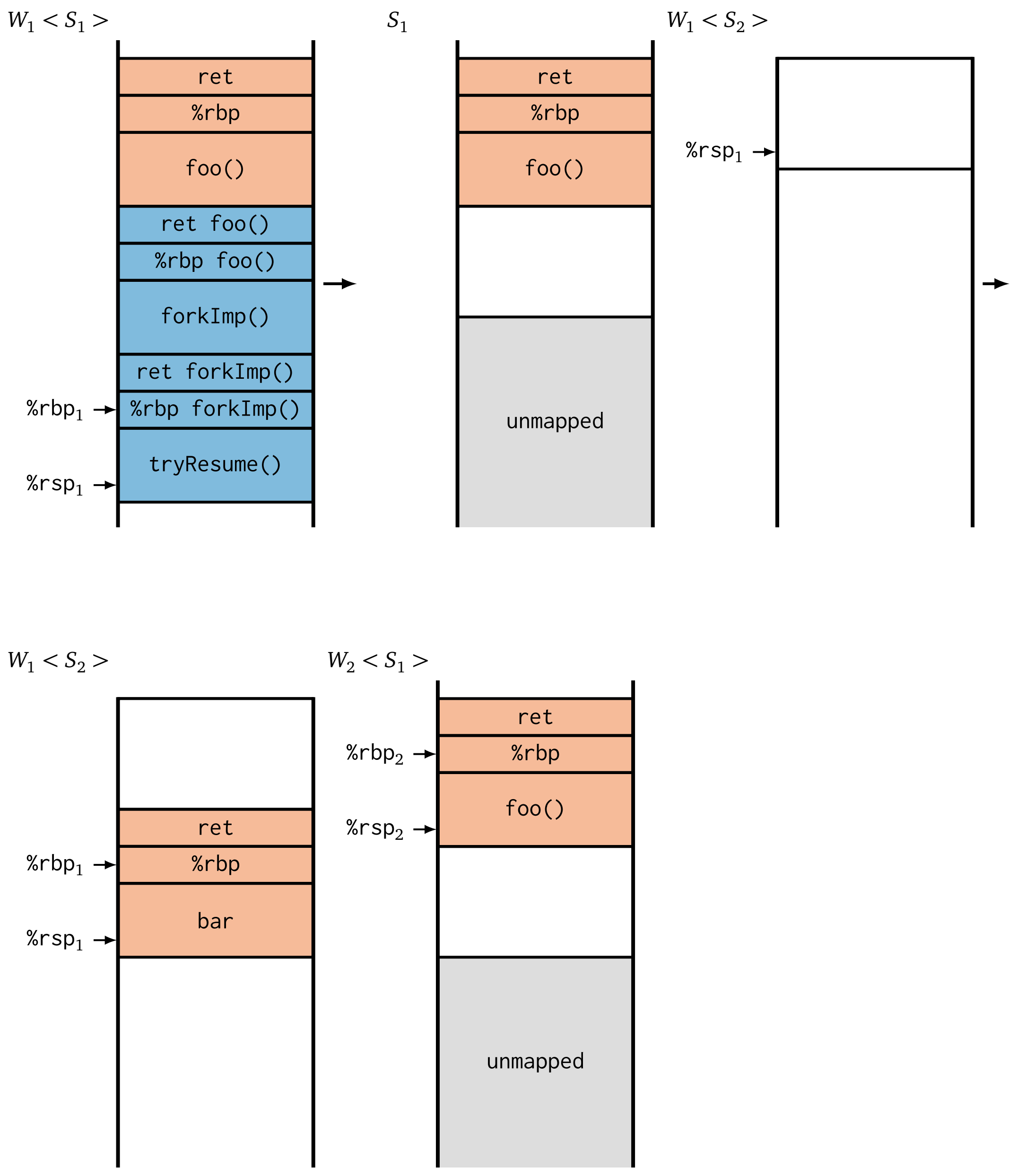

Figure 4.3 – Example of unmapping of unused stack pages.

space, as explained above. Otherwise, the method returns, and the worker invokes the scheduler to start work-stealing.

```
class Continuation {
    /* ... */

    __attribute__(noinline)
    setIp() {
        ip = __builtin_return_address();
    }

    void setJmp() {
        membar(setIp());
    }

    void longJmp() {
        execute(sp);
    }
}
```

Listing 4.14 – The `setjmp()` and `longjmp()` implementation.

4.6 Synchronization

The counter of active children is a shared variable and can be accessed by multiple workers simultaneously. Therefore, its access needs to be synchronized to avoid a data race situation. Furthermore, stealing or popping from a work-stealing dequeue is a critical section as well. The dequeue itself gets accessed in parallel and needs to be synchronized internally. However, since EMPER already offers a locked and a lock- and wait-free implementation, the synchronization of work-stealing dequeues is not within the scope of this thesis. But, the `Fibril` class uses stealing and popping of tasks as a signaling mechanism between workers, since the two actions are mutually exclusive. As shown in Listing 4.10, when a worker returns from a forked off function, it tries to pop the previously pushed continuation from its work-stealing dequeue. If it finds the task was stolen, the worker will try to resume it and thereby decrement the counter of active children. On the other hand, the thief that stole the task will run the task, and by doing so, increment the counter. Since the counter represents the level of parallelism and stealing increases parallelism, while trying to resume decrements it, the counter has to be in sync with steals and resume attempts. If the counter is out of sync, when a steal happens, there is a race for the counter variable. If the victim notices the steal and decrements and uses the value of the counter to decide whether to resume the function instance, before the thief has incremented the counter after the steal, the value used by the victim will be too low, potentially leading the victim to erroneously resuming the continuation. Therefore, stealing a task and incrementing the counter of active children form a critical section and need to appear as an atomic action to other workers.

Furthermore, while a worker unmaps the unused pages of a blocked context with a suspended frame, the context may not be resumed by another worker simultaneously. Otherwise, it is possible that pages get unmapped after the resuming worker has already allocated a new stack frame, resulting in erroneous and undefined behavior.

4.6.1 Lock-Based

The lock-based way of synchronization is relatively straight forward. Listing 4.15 shows where a lock must be used to prevent race conditions in the `Fibril` class. Any mutual exclusion locking mechanism can be used. Here, a mutex from the standard library is used. The counter of active children is protected by the mutex. In order to synchronize the critical section formed by stealing and incrementing the counter of active children, the thief has to lock the `Fibril` object inside the atomic steal operation of the work-stealing dequeue. Then when the `run()` method is executed, the counter can be incremented safely. The object can then be unlocked before the stolen task is executed. That way, the critical section appears as an atomic operation to other workers.

The `tryResume()` method has to protect the counter from parallel access and also ensure that if unused pages of a blocked context get unmapped, the context can not be resumed, before the operation is finished. In order to synchronize the operations, the lock is acquired before the counter is accessed, and released after the unmapping, if the worker cannot resume the context, or after decrementing the counter, if the worker can resume the context.

```
class Fibril {
    Continuation continuation;
    int activeChildren = 0;
    void* context;
    std::mutex mutex;

    void tryResume() {
        mutex.lock();
        if (--activeChildren > 0) {
            if (context == worker.context) {
                contextManager.unmap(worker.context, continuation.rsp);
                worker.context = contextManager.alloc();
            }
            mutex.unlock();
            return; /* start work-stealing */
        }
        mutex.unlock();
        if (context != worker.context) {
            contextManager.free(worker.context);
            worker.context = context;
        }
        continuation.execute(continuation.rsp); /* resume */
    }

    void run() {
        if (activeChildren == 0)
            activeChildren = 2;
        else
            activeChildren++;
        mutex.unlock();
        continuation.execute(worker.tos);
    }

    /* ... */
}
```

Listing 4.15 – The lock-based synchronization of the `Fibril` class.

4.6.2 Lock- and Wait-Free Algorithm

For the lock- and wait-free implementation of the synchronization of the critical sections formed by stealing and incrementing the counter of active children, and of preventing the resumption of a context while parts of it get unmapped, are separated. Listing 4.16 shows the corresponding code. The counter of active children itself becomes an atomic variable that ensures that basic operations like increment, decrement, subtraction, load and store are executed atomically. The counter gets initialized with the value `INT_MAX` the biggest positive number that fits in a variable of type **`int`** to account for every possible steal that could happen[9]. The variable gets decremented in the `tryResume()` method, similar to the basic implementation without synchronization. Since the decrement is an atomic operation, no further action is required. However, the counter cannot reach zero, because the assumption is that much less stealing and decrementing than `INT_MAX` will occur. Therefore, a second counter `steals` of type **`int`** is used to keep track of the number of steals that happen. This counter is initialized with zero and is incremented in the `run()` method. Since there is always only one worker executing a function instance at any point in time, although over the duration of a function instance's lifetime it can be resumed and executed by multiple workers, the `run()` method is also only executed by one worker at a time and acts as a serialized section. Thus there is no race condition when incrementing the counter. When a worker reaches the `join()` method and the `steal` counter is greater than zero, that is the continuation was stolen at least once, the worker has to try to resume the continuation. However, if the worker would only execute the `tryResume()` method and decrement the counter, it would not reach zero, even if the child task was the last to join. Therefore, the counter of steals and the counter of active children have to be synchronized, to obtain the actual number of active children, before the worker tries to resume the continuation. Since there cannot happen more steals at this point, it is safe to set the active children counter to its correct value. The active children counter served as a counter for finished child tasks so far since it was decremented for every worker that tried to resume the continuation. To calculate the correct value, it has to be set to the number of steals, the number of active children, minus the number of already finished child tasks. To accomplish that, its initial value, `INT_MAX`, has to be subtracted from it, and the number of steals, plus one for the initial worker, has to be added to it. Since two operations would expose an invalid intermediate state, it is done in one subtraction. First, the number of steals gets subtracted from `INT_MAX`. The result represents the number of steals that was accounted for but did not happen. Then, the result is subtracted from the active children counter in one atomic operation. After that, the counter shows the correct value and will eventually reach zero when a worker tries to resume the continuation.

In order to Synchronize the resumption of a context and the unmapping of parts of its context, an additional atomic variable `resumable` of type **`bool`** is required. This variable indicates if the continuation is resumable and is used to communicate between the worker that unmaps the context and the worker that wants to resume it. The initial value of the variable is `false`. When a worker has to unmap pages of its context and switch to a new one, it sets the value to `true`, after finishing the unmap operation. The worker that is the last to finish its work and has to resume the continuation has to read the value of the variable. If the value is already `true`, the unmapping has finished, and the worker can proceed to resume the continuation. However, if the value is still `false`, it means the unmapping is still in progress, and the worker cannot resume the context. If the worker had to wait, the algorithm would not be wait-free. Therefore, the worker will not wait, but start work-stealing to find other work, instead. In this case, the unmapping worker has to resume the continuation itself,

[9]In practice a 64-bit wide unsigned integer variable can be used. It is safe to assume that less than 2^{64} steals will happen. Furthermore, if that many steals could happen, the counter would be to small for every implementation, regardless of the type of synchronization.

after finishing the operation. Since this needs to be signaled to the worker, the variable is not just read, but instead, the value of the variable gets exchanged atomically. Thus, the unmapping worker also reads the value when setting it to `true`, and the worker that tries to resume the continuation also writes a value when checking the variable. When the resuming worker reads the variable, it sets its value to `true` itself. The unmapping worker will see that the value is not `false` anymore when setting the variable to `true`, which indicates that the resuming worker already checked the variable and started work-stealing. In this case, the unmapping worker has to resume the continuation itself.

```
class Fibril {
    Continuation continuation;
    void* context;
    std::atomic<bool> resumable = false;
    std::atomic<int> activeChildren = INT_MAX;
    int steals = 0;

    void tryResume() {
        if (--activeChildren > 0) {
            if (context == worker.context) {
                contextManager.unmap(worker.context, continuation.rsp);
                if (resumable.exchange(true) == true)
                    continuation.execute(continuation.rsp); /* resume */
                worker.context = contextManager.alloc();
            }
            return; /* start work-stealing */
        }
        if (context != worker.context) {
            if (resumable.exchange(true) == false)
                return; /* start work-stealing */
            contextManager.free(worker.context);
            worker.context = context;
        }
        continuation.execute(continuation.rsp); /* resume */
    }

    void run() {
        steals++;
        continuation.execute(worker.tos);
    }

    void join() {
        if (steals == 0)
            return;

        /* auto forkImp = ... */

        activeChildren.fetch_sub(INT_MAX - (steals + 1));
        membar(joinImp(this));
        resumable = false;
        steals = 0;
    }

    /* ... */
}
```

Listing 4.16 – The wait-free synchronization of the `Fibril` class.

4.7 Linking Regions

When a function is forked off, arguments have to be passed to the function. Therefore, the `forkImp()` function has to take the arguments of a function that is forked off and pass them on to the function when calling it. The `forkImp()` function also receives two arguments itself, a pointer to the `Fibril` object and a function pointer to the function to be forked off. The first six arguments are passed via registers, whereas further arguments have to be passed on the stack. The area on the stack where arguments are located is referred to as linking region. Figure 4.4 illustrates how the stack layout changes when a function takes more than six arguments and linking regions are introduced. It is a modified version of Figure 4.2 where function `a()` takes seven arguments and function `b()` takes eight arguments. When worker W_1 calls the `forkImp()` function to fork off function `a()`, it first pushes the memory arguments onto the stack in reverse order. The first argument to the `forkImp()` function is the pointer to the `Fibril` object. The second argument is the function pointer to `a()`. The remaining arguments are the arguments for function `a()`. Therefore, W_1 has to call the `forkImp()` function with a total of nine arguments, whereof the last three are passed on the stack. After moving the arguments to the registers and pushing them on the stack, the `forkImp()` function is invoked, and the stack frame is allocated, similar to the case without stack arguments. However, when the function returns, the caller function has to clean up the stack and pop the arguments. The `forkImp()` function itself has to forward the arguments to function `a()`. In order to do so, the arguments have to be moved to the correct registers and copied on the stack. Since the first two arguments for `forkImp()` are not passed to `a()`, there is only one stack argument out of the seven. After the first six arguments are in the correct registers and the last argument, `forkImp() arg9` of the `forkImp()` invocation, is pushed onto the stack as `a() arg7`, function `a()` is called, and the stack frame is allocated.

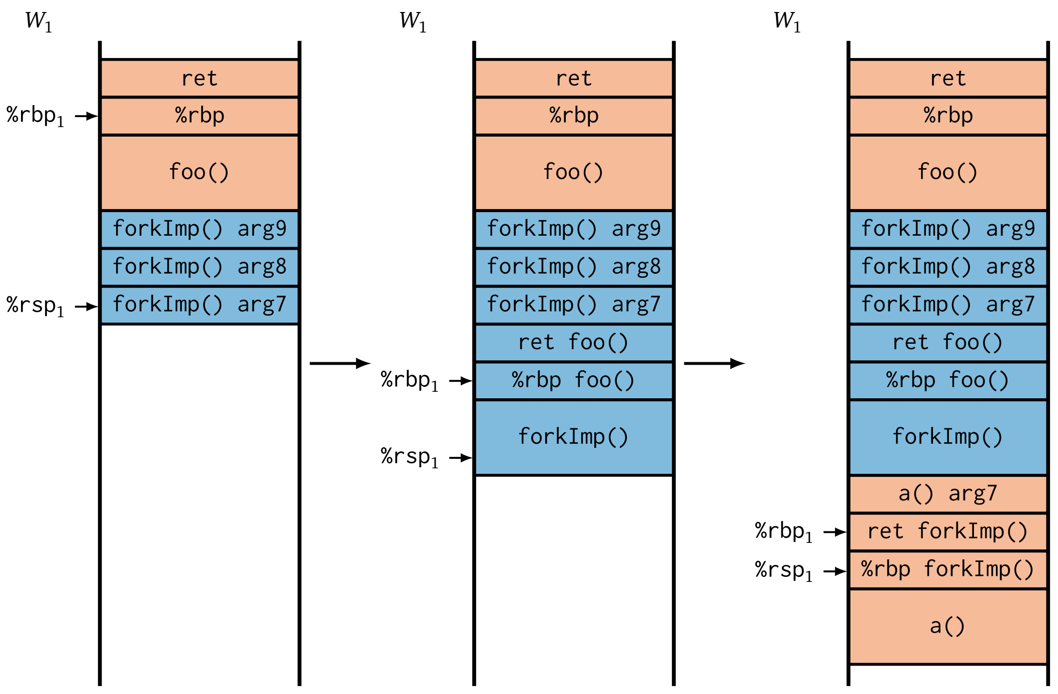

Figure 4.4 – Example of argument passing.

When a worker steals a continuation and executes it, it continues at the instruction directly following the call of the `forkImp()` function. However, the first instructions after a function with stack arguments will pop the stack arguments. That means the stack pointer `rsp` will get incremented. If the thief sets its stack pointer to the very top of its working stack when resuming the stolen continuation, it will increment the stack pointer, pointing to an area outside the allocated stack, resulting in undefined behavior and errors. Therefore, the worker has to leave some space to the top end of the stack and treat it as the linking region, that can then be freed from the stack when popping arguments. The `Fibril` class reserves an 128 byte large area as linking region. By choosing a fixed size, the effective number of arguments, a forked off function may take, is limited to the six arguments passed in registers plus the number of arguments that fit inside the linking region[10], minus the number of arguments used by the `forkImp()` function itself.

After stealing the continuation of `foo()` from worker W_1, in Figure 4.4, worker W_2, in Figure 4.5, first frees an area, equivalent to three stack arguments, on its stack. Then, when calling the `forkImp()` function itself to fork off function `b()` that takes eight arguments, it has to push the last four arguments for `b()` onto the stack in order to pass them to `forkImp()`. The remaining steps are executed analogously to W_1 forking off `a()`.

[10]On x86-64 a stack argument of numeric or pointer type is always 8 byte wide. Thus, 16 arguments fit in an 128 byte large linking region.

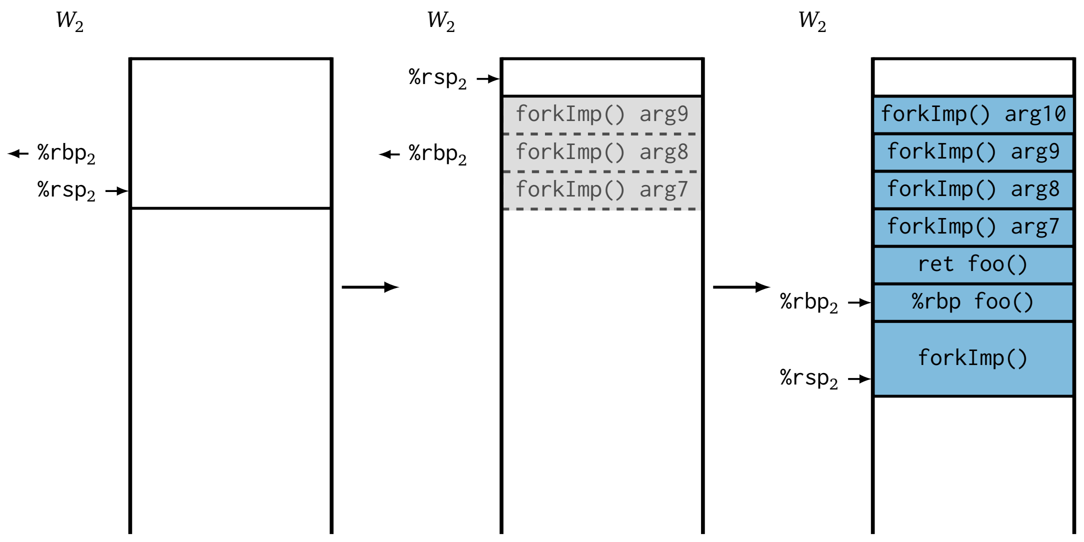

Figure 4.5 – Example of the linking region on a thief's stack.

5 ANALYSIS

In this chapter, the performance of EMPER's new lock- and wait-free continuation-stealing tasks will be evaluated. First, the lock-based version of the continuation-stealing tasks will be compared against EMPER's wait-free child-stealing tasks, that existed before the work of this thesis. Then the wait-free version of the continuation-stealing tasks will be compared against the lock-based version and then against Fibril [YMC16]. Furthermore, the impact unmapping of stacks has on performance will be evaluated. Then EMPER's wait-free continuation-stealing tasks will be compared against existing runtimes like Fibril, Intel Cilk Plus [Rob13], and TBB [CM08; Suk09]. Finally, suggestions for future work will be discussed.

For the evaluation, the twelve benchmarks described in Table 5.1 were used. These benchmarks have been used in previous publications about the cactus stack problem and are adopted from Fibril and adjusted for C++. The Source Lines of Code (SLOC) were counted using *SLOCCount* [Whe01].

5.1 General Setup

All benchmarks were run on a Non-Uniform Memory Access (NUMA) system with 4 Intel Xeon E7-4830 v3 CPUs, running at 2.10 GHz. Each CPU package has its own NUMA-node and 12 cores with 2-way Simultaneous Multithreading (SMT), giving the system a total of 96 hardware threads. The total available main memory is 512 GiB. The operating system used on the machine was

Benchmark	**Input**	**Description**	**SLOC**
cholesky	4000/40000	Cholesky factorization	455
fft	2^{26}	Fast Fourier transformation	3055
fib	42	Recursive Fibonacci	40
heat	2048×500	Jaccobi heat diffusion	149
integrate	10^4 ($\epsilon = 10^{-9}$)	Quadrature adaptive integration	59
knapsack	32	Recursive knapsack	97
lu	4096	LU-decomposition	274
matmul	2048	Matrix multiply	115
nqueens	14	Count ways to place N queens	48
quicksort	10^8	Parallel quicksort	66
rectmul	4096	Rectangular matrix multiply	291
strassen	4096	Strassen matrix multiply	377

Table 5.1 – Description of the 12 benchmarks.

Ubuntu Linux 18.04 with kernel version 4.15.0. The compiler used to compile all benchmarks, the Fibril [YMC16] and Intel Cilk Plus runtimes, and EMPER was GCC 7.4.0. GCC 7.4.0 has native support for Intel Cilk Plus and could be used to compile the Intel Cilk Plus version of the benchmarks as well as all the other versions. All code was compiled with optimization level `-O2`. For the TBB versions of the benchmarks version 2017 Update 7 of the TBB runtime was used. For the Intel Cilk Plus and Fibril benchmarks, the latest version available in the respective public repositories were compiled. The Fibril runtime was adjusted to also use the `MADV_FREE` flag for `madvise()`, to allow for a fair comparison.

All benchmarks were performed using 4 KiB memory pages and 64 KiB stacks. EMPER tries to pin worker threads to hardware threads by setting affinities towards hardware threads in `pthread` attributes. Each experiment was run eleven times in total, where the first run was a warm-up run. The execution time of the remaining ten runs was measured, and the mean execution time was recorded. The `MADV_FREE` flag was used in all experiments, if not specified differently.

5.2 EMPER Performance

Figure 5.1 shows the performance of EMPER's child-stealing tasks, as present in EMPER before the work of this thesis, and continuation-stealing tasks on 1–96 threads. The continuation-stealing tasks use lock-based synchronization for the work-stealing deques and continuations, similar to Fibril. The continuations are synchronized using a spin-lock, and the work-stealing dequeues use Dijkstra's protocol for mutual exclusion similar to Cilk's dequeues [FLR98], which use a spin-lock internally. In the following, lock-based synchronization for EMPER and Fibril refers to the described type of synchronization, if not specified differently.

Since workers are pinned to hardware threads in EMPER, the first 48 workers use separate cores, and the second 48 workers use the second SMT hardware thread of each core. Because of this, the scaling changes after 48 workers, since workers do not get assigned exclusively to cores anymore and have to share hardware resources. In some benchmarks, this only leads to less performance per additional core. However, in some benchmarks, the speedup stagnates or even decreases. This is most evident in `cholesky`, `lu`, and `quicksort`, where the performance of 96 workers can drop down to only 0.56× that of 48 workers.

In many benchmarks continuation-stealing greatly outperforms child-stealing, when using 96 threads. In `fib`, `integrate`, `knapsack`, `matmul`, and `nqueens`, continuation-stealing outperforms child-stealing by 24.8×, 16.4×, 13.3×, 7.9× and 20.2×, respectively. In the remaining benchmarks, continuation-stealing performs either better or similar to child-stealing, except for `strassen`, where the performance drops below that of child-stealing after 48 threads. In summary, while the performance increase can be explained partly by the difference in calling convention when forking, continuation-stealing is a great improvement over child-stealing.

However, Figure 5.1 shows only the lock-based version of the continuation-stealing tasks. Ideally, the performance can be increased further by using wait-free synchronization.

5.2.1 Wait-Free Performance

Figure 5.2 shows the performance of wait-free and lock-based synchronized versions of EMPER's continuation-stealing tasks on 1–96 threads. The "wait-free" version uses the lock- and wait-free algorithm of this thesis to synchronize the continuations and EMPER's wait-free work-stealing dequeues. The "locked (deque only)" uses wait-free continuations and lock-based dequeues, and the "locked" version uses lock-based synchronization for both.

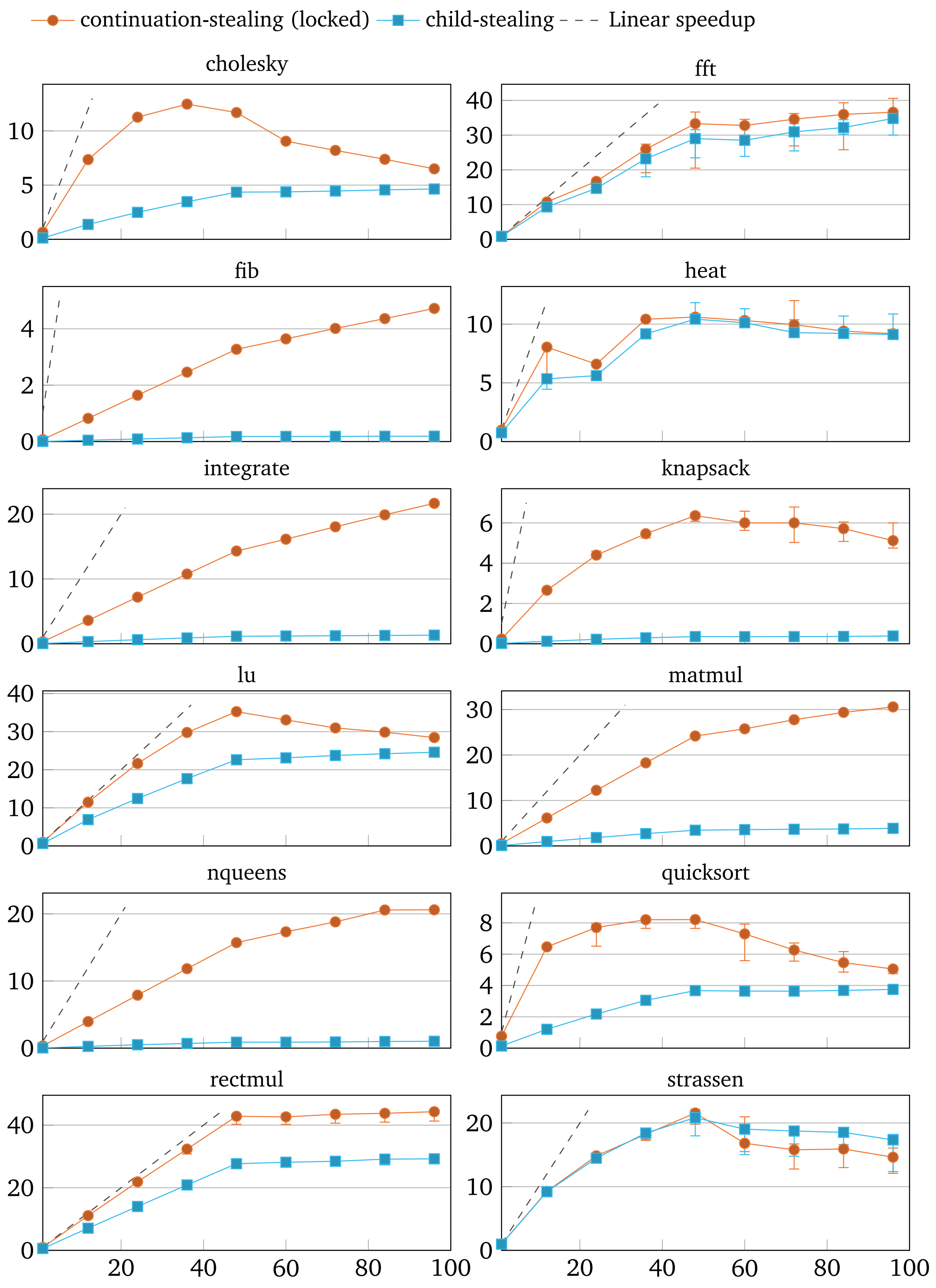

Figure 5.1 – Comparison between EMPER child-stealing and continuation-stealing tasks.

The X axis are the number of worker threads and the Y axis are the speedups T_{serial}/T_P, where T_P is the execution time using P worker threads.

The `integrate` benchmark shows that the wait-free version scales better than the locked versions from 1–48 threads and worse from 48–96 threads so that the locked versions almost catch up in performance. A possible explanation for this behavior is that the wait-free version generally utilizes the computational power of a core better than the locked version and, therefore, scales better from 1–48 cores. The idea of SMT is to allow for better utilization of the hardware resources of a CPU core by sharing some of the resources of one core among multiple hardware threads. Therefore, when the second SMT hardware thread of a core is used, the wait-free version, which already utilizes more than half of the computational resources of that core gets less additional resources than the first hardware thread offered. The version using spin-locks does not use the hardware as effective since time is spent waiting for locks. In this case SMT can be effective, and one hardware thread of a core can use more hardware resources, while the other one is waiting. Thus, resulting in better performance gains on 48–96 cores, compared to the wait-free version and compensating the worse scaling of blocking synchronization to some extend.

In the benchmarks shown, the wait-free version outperforms the locked versions by 1.06× in `integrate`, by up to 1.22× in `quicksort`, 1.32× in `knapsack` and 1.33× in `cholesky`. In the benchmarks omitted, the performance is within less than 1.1× the performance of each other. The version "locked (dequeue only)", using the wait-free algorithm with locked dequeue, only manages to outperform the fully locked version by a significant margin in `knapsack` by 1.22×. In all other benchmarks, the performance is only slightly better or very similar.

There are two possible explanations why the wait-free algorithm of this thesis alone, seemingly, does not improve the performance a lot. Firstly, it is possible that the critical section of the continuations, where wait-free synchronization is used, is not under much contention and, thus, has no big impact on performance in general. Secondly, the locks used to synchronize the work-stealing dequeues and the serialization caused by them could lead to mostly serialized entering of the critical sections of continuations and, thereby, reducing the contention of the critical section and reducing its impact on performance.

The lock-based synchronization of continuations requires that the lock is acquired from within the locked section of a work-staling dequeue when the continuation is stolen. Therefore, it is not possible to use wait-free work-stealing dequeues in combination with the lock-based version of the continuations. As a result, the difference in performance caused by the wait-free algorithm of this thesis, compared to the locked version, cannot be measured and analyzed directly.

However, the wait-free synchronization of continuations enables the use of wait-free work-stealing dequeues for scheduling. The fully wait-free version, using both wait-free continuations and dequeues, is an improvement over the locked versions and significantly outperforms them.

5.2.2 Comparison with Fibril

Figure 5.3 shows a comparison of the performance of EMPER's wait-free continuation-stealing tasks and Fibril. Except for the `quicksort` benchmark, where EMPER outperforms Fibril by 1.28×, Fibril is faster or performs similar to EMPER. In `cholesky`, `fib`, `knapsack`, and `nqueens`, Fibril's performance is 1.48×, 1.36×, 1.3× and 1.24× that of EMPER, respectively.

However, this outcome can be explained by differences in the runtimes. Fibril is written in C and is very minimalistic, with all the focus on performance. In contrast, EMPER is written in C++ and uses C++ standard library functionality and heavy-weight constructs like `std::function`. EMPER puts much focus on extensibility and maintainability, besides performance. Furthermore, EMPER has additional functionality like child-stealing tasks and supports multiple scheduling strategies. Despite these differences, EMPER's performance comes close to that of Fibril.

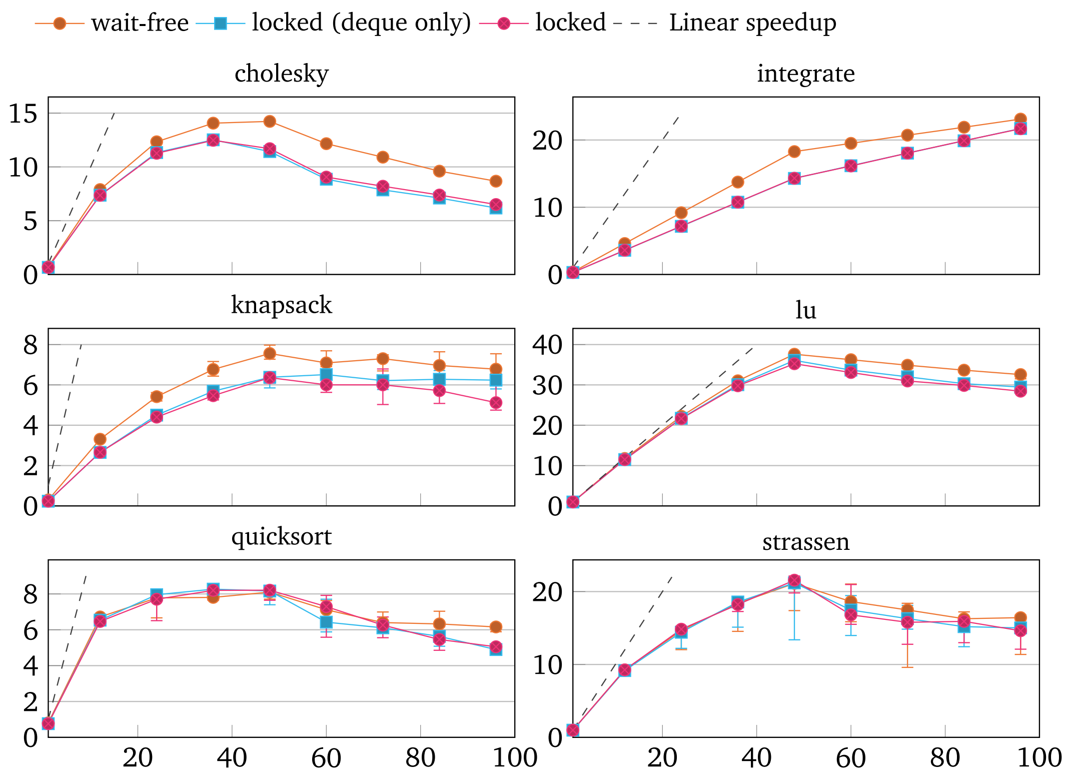

Figure 5.2 – Comparison of synchronization types in EMPER.

The X axis are the number of worker threads and the Y axis are the speedups T_{serial}/T_P, where T_P is the execution time using P worker threads.

Furthermore, the lock- and wait-free algorithm for the synchronization of continuations, presented by this thesis, and the wait-free work-stealing dequeues, as used in EMPER, can be implemented in Fibril to improve its performance further. Section 5.3 compares the performance of several runtimes, including a version of Fibril that employs the wait-free approach of this thesis.

5.2.3 Impact of `madvise()`

Figure 5.4 shows a comparison of EMPER's wait-free continuation-stealing tasks using `madvise()` with `MADV_FREE`, `MADV_DONTNEED`, or no `madvise()`. Yang and Mellor-Crummey [YMC16] stated that the overhead of unmapping unused stack pages and the thereby increased amount page faults, is negligible. However, some of the experiments of this thesis show differing results, where performance is decreased notably.

In `cholesky`, `heat`, `knapsack` and `lu`, using the `MADV_FREE` flag instead of `MADV_DONTNEED` improved the performance by 1.7×, 1.18×, 1.46× and 1.27×, respectively. Forgoing `madvise()` improved the performance in these benchmarks further by 2.52×, 1.23×, 1.59× and 1.37×, respectively, compared to the version with `MADV_FREE`. In the remaining benchmarks, all three versions perform nearly identically.

It is difficult to find the exact reasons for the drop in performance of these benchmarks, since the amount of steal and unmap operations, and page faults, shown in Figure 5.5 do not indicate

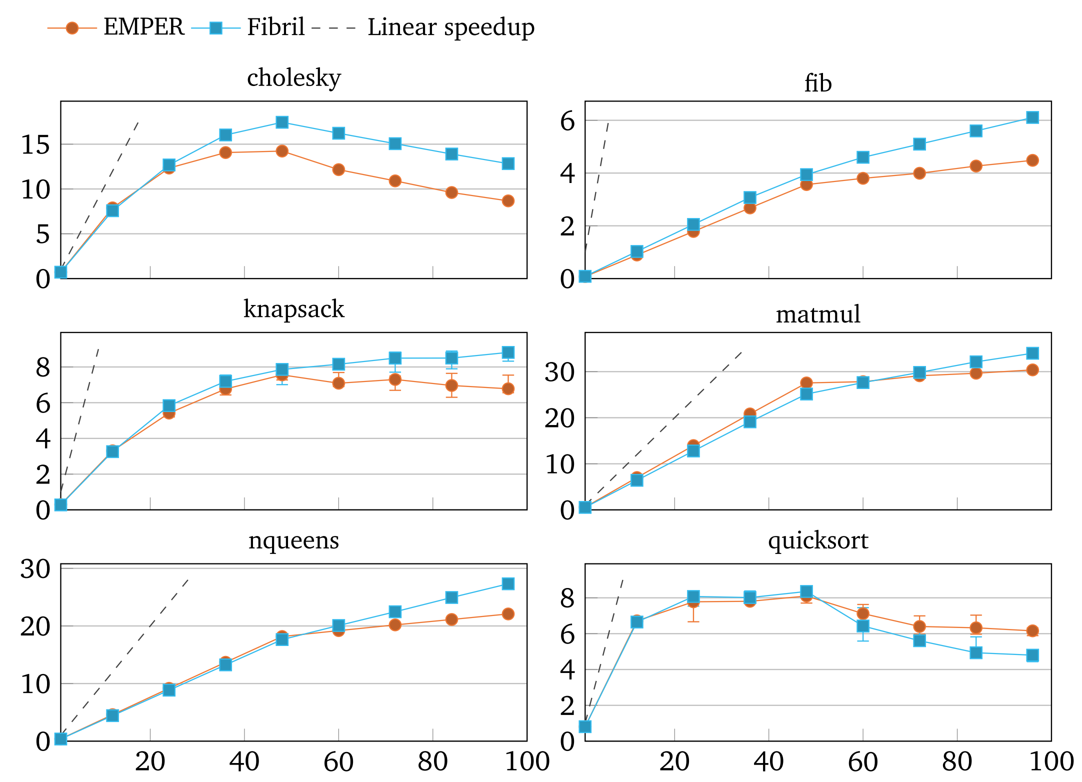

Figure 5.3 – Comparsion between EMPER and Fibril.

The X axis are the number of worker threads and the Y axis are the speedups T_{serial}/T_P, where T_P is the execution time using P worker threads.

one single cause. Generally, the unmap operation itself introduces additional time spend executing `madvise()`. In the case of `cholesky`, the amount of unmaps is the highest of all benchmarks. For some benchmarks, the amount of page faults increases significantly. However, in some benchmarks, it decreases. Furthermore, unmaps can lower the parallelism of an application. If a worker spends time unmapping pages, delaying the resumption of a task, in a case where otherwise the task would be continued and would create a new task, and where no other work is available, workers have to stall and wait for new tasks to be created. This also applies to page faults delaying task creation. The amount of steals is lower in all experiments using `madvise()`, compared to those without.

Unmapping unused stack pages can decrease performance significantly. Reducing the amount of unmaps could prevent this and further improve the performance of a runtime using this cactus stack implementation, in some cases. However, further study of this matter is might be.

5.3 Comparison of Runtimes

Figure 5.6 shows a comparison of the performance of EMPER, Fibril, Cilk Plus, and TBB on 1–96 threads. EMPER uses wait-free continuation-stealing tasks. Fibril is shown in two variations, using wait-free and lock-based synchronization. “Fibril” is the original version using lock-based synchronization. “Fibril (wait-free)” is altered to use the wait-free algorithm of this thesis to

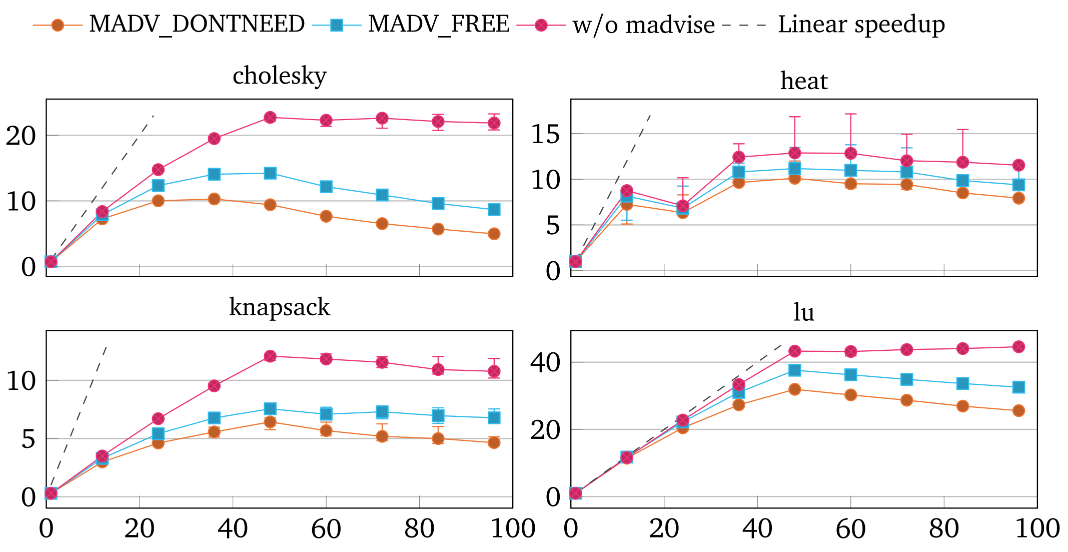

Figure 5.4 – The impact of `madvise()` in EMPER.

The X axis are the number of worker threads and the Y axis are the speedups in T_{serial}/T_P, where T_P is the execution time using P worker threads. All benchmarks use EMPER's wait-free continuation-staling tasks.

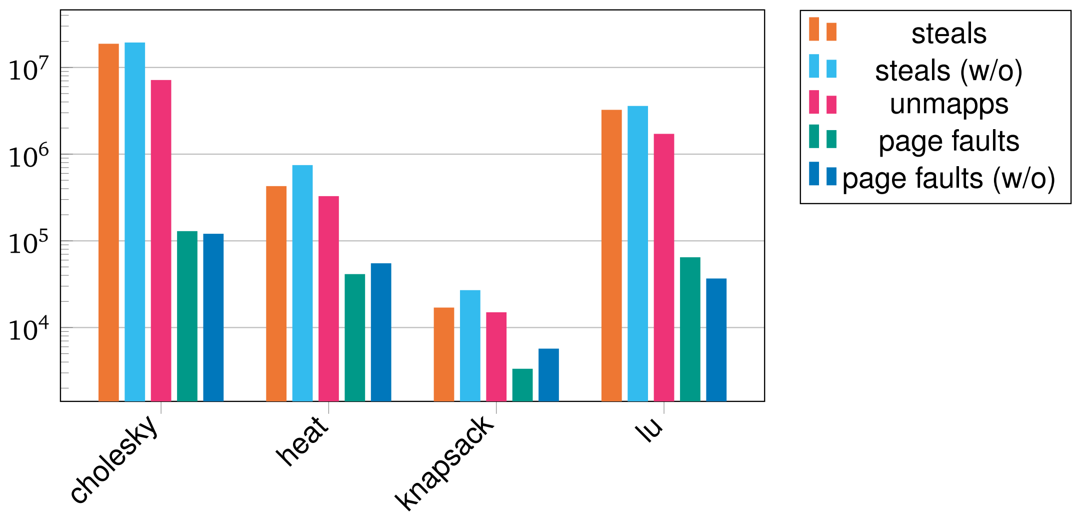

Figure 5.5 – Key operations on 96 threads.

synchronize its continuations and EMPER's wait-free work-stealing dequeues for the scheduler. In both versions, the memory manager for stacks uses a global pool of stacks that is synchronized with a lock, making it not fully wait-free.

EMPER outperforms Cilk Plus and TBB in most benchmarks. In `fib`, `integrate`, `matmul`, `nqueens`, and `quicksort`, EMPER's performance is 1.61×, 1.6×, 1.6×, 1.54× and 1.25× that of Cilk Plus and 9.27×, 9.26×, 4.8×,2.54× and 3.78× that of TBB, respectively. Additionally, EMPER outperforms TBB in `knapsack` by 3.68×. However, in `cholesky` and `lu`, EMPER's performance drops down to 0.52× and 0.81× that of Cilk Plus, and in `heat` and `strassen`, to 0.83× and 0.8× that of Cilk Plus, and 0.78× and 0.88× that of TBB, respectively.

Fibril performs worse than Cilk Plus in `cholesky`, `heat`, and `lu`, three of the four benchmarks where `madvise()` has the most significant impact on performance. Cilk Plus does not unmap stacks, indicating that both EMPER and Fibril could potentially outperform Cilk Plus in these benchmarks, if the negative effect, unmapping stacks has on the performance, could be reduced.

The wait-free version of Fibril performs very similarly to the original Fibril in most experiments, using 96 worker threads, except for `integrate`, where the wait-free version outperforms the lock-based version by 1.23×. In `knapsack`, the wait-free version performs better on 48 threads than the locked version on 96 threads by 1.21× but drops down to only 1.05× the performance on 96 threads. That shows that wait-free synchronization can improve performance, notably. Since Fibril uses a global pool of stacks that uses lock-based synchronization even in the wait-free version of Fibril, there might be room for further improvement.

5.4 Future Work

The implementation and evaluation showed that there is room for further improvement of performance, but also usability. Adding dedicated compiler support, similar to Cilk Plus, could have multiple advantages. Firstly, the compiler could allocate a `Fibril` object automatically and pass it to the fork and join methods. Additionally, the compiler could enforce that variables in the stack frame are addressed using the base pointer and warn or abort compilation if it is not possible for some reason. This could make the use of fork and join easier and more robust. Secondly, this would allow compiler-based optimizations, as described by Schardl, Moses, and Leiserson [SML17], further improving performance in some cases. Moreover, the overhead of the calling convention of fork and join could be reduced. Fibril and EMPER use a function call to save the state of a stack frame and the position in code, where execution should continue upon resumption. However, this entails an additional function call and copying or moving of function arguments. A compiler could produce more efficient code that reduces some of these overheads while maintaining interoperability.

EMPER and Fibril use a memory manager for stacks, to reduce allocations. These memory managers use small private buffers for stacks per worker. Fibril uses an additional global pool to balance between workers. However, Fibril uses lock-based synchronization for the pool. This lock could become a global bottleneck when under high contention. A wait-free implementation could potentially improve performance.

On systems with multiple NUMA-nodes, a NUMA-aware runtime could further improve performance by reducing access to data that is not local to a NUMA-node. A randomized work-stealing scheduler that favors stealing from workers in the same NUMA-node could reduce the migration of tasks across nodes. A memory manager for stacks that uses a global pool per NUMA-node could reduce the migration of stacks.

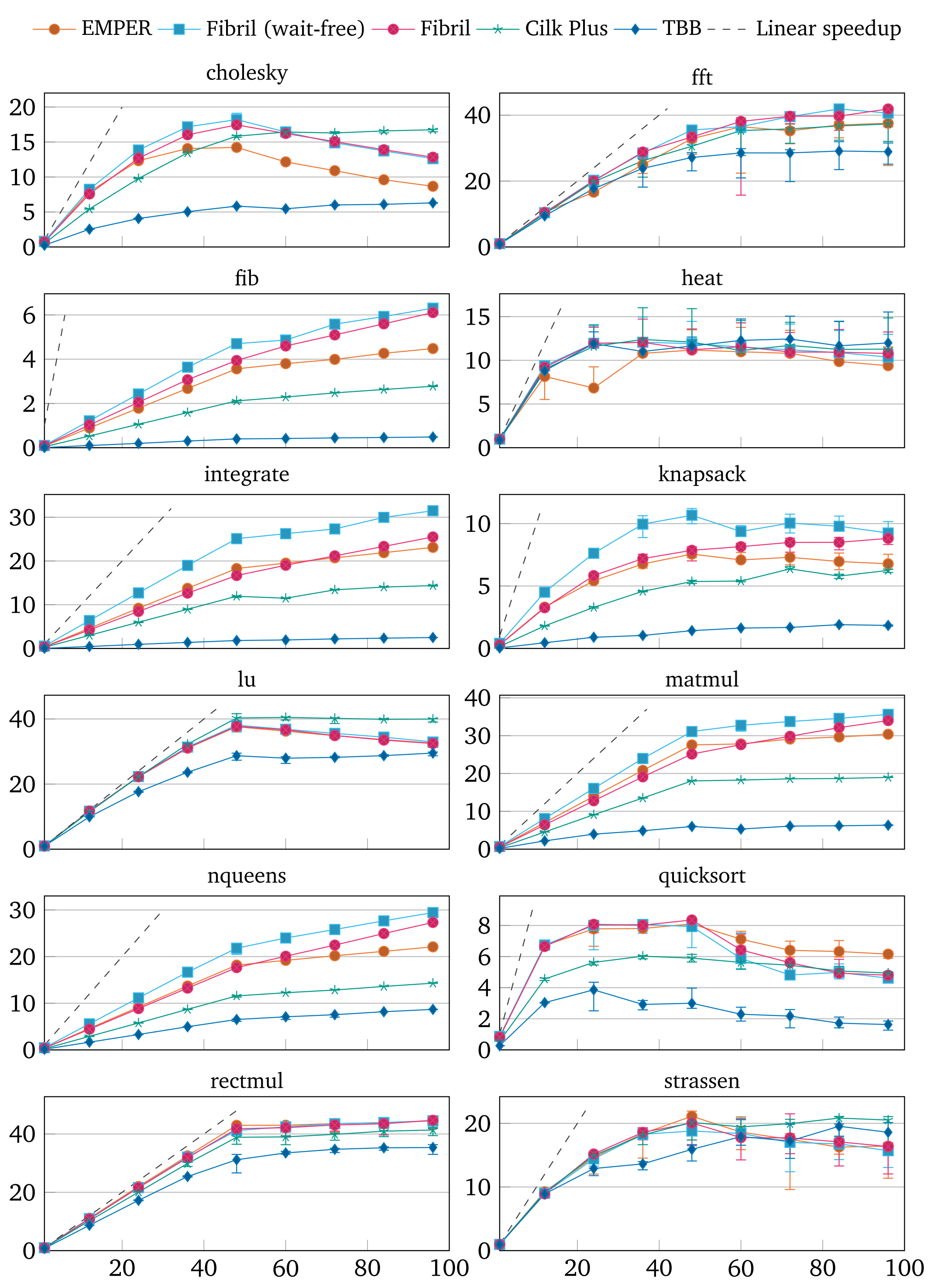

Figure 5.6 – Comparsion of micro-parallelism runtimes.

The X axis are the number of worker threads and the Y axis are the speedups T_{serial}/T_P, where T_P is the execution time using P worker threads.

Finally, since `madvise()` can reduce performance in some cases, reducing the amount of unmaps could stabilize performance. However, further research could be needed to determine when unmapping is useful and how to decide efficiently at runtime, whether to unmap stacks or not.

CONCLUSION 6

In this thesis, a fully lock- and wait-free implementation of continuation-stealing tasks and efficient cactus stack were presented. Continuation-stealing is a way to implement lazy task creation, which reduces the overhead resulting from the creation and bookkeeping of tasks, compared to child-stealing. The results showed that continuation-stealing greatly out-performs child-stealing in most cases. Wait-free synchronization of critical sections can improve performance compared to lock-based synchronization. This thesis presented a wait-free algorithm for the synchronization of continuations that allows the usage of wait-free work-stealing dequeues in combination with continuation-stealing. The results showed that the wait-free approach improved scalability. Furthermore, the wait-free approach of this thesis could be employed in Fibril [YMC16] and further improved its performance, as shown in Figure 6.1. Generally, wait-free synchronization scales better over many cores, but SMT compensates for the worse scaling of locks to some extend. However, in some cases the use of SMT results in a reduction of performance. That appears to be linked to the use of `madvise()` to unmap unused stack pages, in some cases. That partly contradicts the statement of Yang and Mellor-Crummey [YMC16] that the unmapping of stack pages has a negligible impact on performance.

Fork-join microparallelism runtimes have more room for improvement. Reducing the amount of unmap operations could stabilize performance in some applications. NUMA-awareness could reduce the migration of tasks and stacks on NUMA-systems. A wait-free memory manager for stacks with a shared pool could reduce allocations of stacks by balancing them among workers. Dedicated compiler support, similar to Intel Cilk Plus [Rob13], could be used to improve usability and further improve performance by allowing better compiler optimization and an improved parallel calling convention.

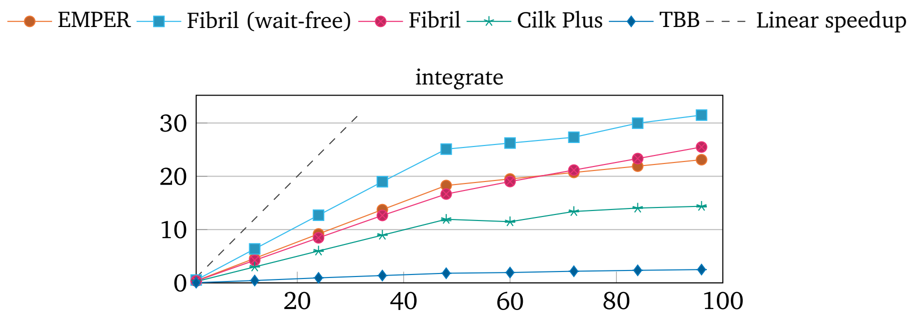

Figure 6.1 – Comparsion of micro-parallelism runtimes.

The X axis are the number of worker threads and the Y axis are the speedups T_{serial}/T_P, where T_P is the execution time using P worker threads.

LIST OF ACRONYMS

PC	Personal Computer
OS	Operating System
EMPER	Extensible Micro-Parallelism Experimentation Runtime
API	Application Programing Interface
ISA	Instruction Set Architecture
ABI	Architecture Binary Interface
GCC	GNU Compiler Collection
TBB	Intel Threading Building Blocks
NUMA	Non-Uniform Memory Access
SMT	Simultaneous Multithreading
SLOC	Source Lines of Code
DAG	Directed Acyclic Graph
TLMM	Thread-Local Memory Mapping

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF ALGORITHMS

REFERENCES

[BL99] Robert D. Blumofe and Charles E. Leiserson. “Scheduling Multithreaded Computations by Work Stealing.” In: *J. ACM* 46.5 (Sept. 1999), 720–748. ISSN: 0004-5411. DOI: 10.1145/324133.324234. URL: https://doi.org/10.1145/324133.324234.

[CM08] G. Contreras and M. Martonosi. “Characterizing and improving the performance of Intel Threading Building Blocks.” In: *2008 IEEE International Symposium on Workload Characterization*. 2008, pp. 57–66. DOI: 10.1109/IISWC.2008.4636091.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. “The Implementation of the Cilk-5 Multithreaded Language.” In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI ’98. Montreal, Quebec, Canada: Association for Computing Machinery, 1998, 212–223. ISBN: 0897919874. DOI: 10.1145/277650.277725. URL: https://doi.org/10.1145/277650.277725.

[Fri+09] Matteo Frigo et al. “Reducers and Other Cilk++ Hyperobjects.” In: *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA ’09. Calgary, AB, Canada: Association for Computing Machinery, 2009, 79–90. ISBN: 9781605586069. DOI: 10.1145/1583991.1584017. URL: https://doi.org/10.1145/1583991.1584017.

[Gcc] *GCC, the GNU Compiler Collection*. accessed 2019-12-10. URL: https://gcc.gnu.org/.

[Hal12] Pablo Halpern. *Strict Fork-Join Parallelism*. 2012. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1645.pdf.

[Lee+10] I-Ting Angelina Lee et al. “Using Memory Mapping to Support Cactus Stacks in Work-Stealing Runtime Systems.” In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT ’10. Vienna, Austria: Association for Computing Machinery, 2010, 411–420. ISBN: 9781450301787. DOI: 10.1145/1854273.1854324. URL: https://doi.org/10.1145/1854273.1854324.

[Lei09] Charles E. Leiserson. “The Cilk++ Concurrency Platform.” In: *Proceedings of the 46th Annual Design Automation Conference*. DAC ’09. San Francisco, California: Association for Computing Machinery, 2009, 522–527. ISBN: 9781605584973. DOI: 10.1145/1629911.1630048. URL: https://doi.org/10.1145/1629911.1630048.

[Mat+14] Michael Matz et al., eds. *System V Application Binary Interface AMD64 Architecture Processor Supplement*. Draft Version 0.99.7. 2014. URL: https://www.uclibc.org/docs/psABI-x86_64.pdf.

[MKH91] E. Mohr, D. A. Kranz, and R. H. Halstead. “Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs.” In: *IEEE Trans. Parallel Distrib. Syst.* 2.3 (July 1991), 264–280. ISSN: 1045-9219. DOI: 10.1109/71.86103. URL: https://doi.org/10.1109/71.86103.

[Rob13] A. D. Robison. “Composable Parallel Patterns with Intel Cilk Plus.” In: *Computing in Science Engineering* 15.2 (2013), pp. 66–71. ISSN: 1558-366X. DOI: 10.1109/MCSE.2013.21.

[Rob14] Arch Robison. *A Primer on Scheduling Fork-Join Parallelism with Work Stealing*. 2014. URL: http://open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3872.pdf.

[SML17] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. “Tapir: Embedding Fork-Join Parallelism into LLVM’s Intermediate Representation.” In: *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’17. Austin, Texas, USA: Association for Computing Machinery, 2017, 249–265. ISBN: 9781450344937. DOI: 10.1145/3018743.3018758. URL: https://doi.org/10.1145/3018743.3018758.

[Suk09] Jim Sukha. “Brief announcement: a lower bound for depth-restricted work stealing.” In: Jan. 2009, pp. 124–126. DOI: 10.1145/1583991.1584025.

[WC93] David B. Wagner and Bradley G. Calder. “Leapfrogging: A Portable Technique for Implementing Efficient Futures.” In: *SIGPLAN Not.* 28.7 (July 1993), 208–217. ISSN: 0362-1340. DOI: 10.1145/173284.155354. URL: https://doi.org/10.1145/173284.155354.

[Whe01] David Wheeler. *SLOCCount*. https://www.dwheeler.com/sloccount/. 2001.

[YMC16] Chaoran Yang and John Mellor-Crummey. “A Practical Solution to the Cactus Stack Problem.” In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’16. Pacific Grove, California, USA: Association for Computing Machinery, 2016, 61–70. ISBN: 9781450342100. DOI: 10.1145/2935764.2935787. URL: https://doi.org/10.1145/2935764.2935787.