2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-4066-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPS49936.2021.00044

Nowa: A Wait-Free Continuation-Stealing Concurrency Platform

Florian Schmaus, Nicolas Pfeiffer
Wolfgang Schröder-Preikschat
Friedrich-Alexander-University Erlangen-Nürnberg (FAU)
Erlangen, Germany
{schmaus,pfeiffer,wosch}@cs.fau.de

Timo Hönig
Ruhr University Bochum (RUB)
Bochum, Germany
timo.hoenig@rub.de

Jörg Nolte
Brandenburg University of Technology Cottbus-Senftenberg (BTU)
Cottbus, Germany
joerg.nolte@b-tu.de

***Abstract*—It is an *ongoing challenge* to efficiently use parallelism with today's multi- and many-core processors. Scalability becomes more crucial than ever with the rapidly growing number of processing elements in many-core systems that operate in data centres and embedded domains. Guaranteeing scalability is often ensured by using fully-strict fork/join concurrency, which is the prevalent approach used by concurrency platforms like Cilk. The runtime systems employed by those platforms typically resort to lock-based synchronisation due to the complex interactions of data structures within the runtime. However, locking limits scalability severely. With the availability of commercial off-the-shelf systems with hundreds of logical cores, this is becoming a problem for an increasing number of systems.**

This paper presents Nowa, a novel wait-free approach to arbitrate the plentiful concurrent strands managed by a concurrency platform's runtime system. The wait-free approach is enabled by exploiting inherent properties of fully-strict fork/join concurrency, and hence is potentially applicable for every continuation-stealing runtime system of a concurrency platform. We have implemented Nowa and compared it with existing runtime systems, including Cilk Plus, and Threading Building Blocks (TBB), which employ a lock-based approach. Our evaluation results show that the wait-free implementation increases the performance up to $1.64\times$ compared to lock-based ones, on a system with 256 hardware threads. The performance increased by $1.17\times$ on average, while no but one benchmark exhibited performance regression. Compared against OpenMP tasks using Clang's `libomp`, Nowa outperforms OpenMP by $8.68\times$ on average.

***Index Terms*—scheduling, concurrency platform, wait-free**

I. Introduction

Driven by the rapidly rising number of computation cores, the way programmers can express concurrency in their code and how this concurrency is efficiently transformed into parallelism becomes decisive. Since it is cumbersome to introduce support for parallelism into serial programming languages [1], new programming languages are designed with primitives supporting parallelism from the start. As a result, concurrency control is part of the language specification. An example of this is the Go programming language. Besides the programming language layer, Go also includes a runtime system, which is *the* essential driver of the parallel execution of the concurrency expressed in the Go programs. The combination of a parallel programming language and an accompanying runtime system is called a *concurrency platform*.

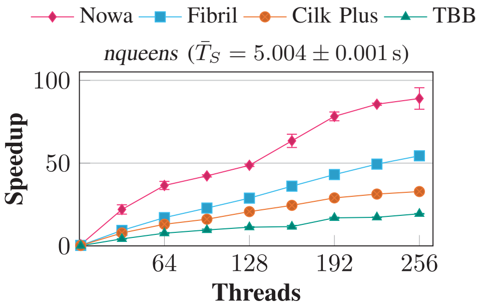


Figure 1: Comparison of runtime systems of different concurrency platforms. Our wait-free Nowa system exhibits better speedup than the lock-based ones.

The choice of concurrency platform has a direct impact on the execution time of a parallel application. Figure 1 shows a comparison of concurrency platforms, including established ones like Cilk Plus or Threading Building Blocks (TBB). However, the performance of existing concurrency platforms is often limited by the lock-based synchronisation used within the platform's runtime system. In contrast, as can be seen in Figure 1, our Nowa runtime system unlocks additional performance on systems with hundreds of hardware threads, by combining our wait-free [2] approach with a work-stealing queue algorithm, optimised for scalability. As to the growing number of cores at the hardware level, it is also safe to assume that the relevance of concurrency platforms will continue to increase in the future.

All concurrency platforms' joint property is that they consist of a *programming-language layer* and a *runtime system*. The programming-language layer's goal is to reduce the developer's burden of concurrent programming. This is done by providing various language constructs to define and coordinate concurrent tasks. Those constructs may deliberately limit the amount of concurrency that can unfold at a given point. For example, fully-strict concurrent computations [3] require all child tasks to finish before a parent task itself can finish.[1]

[1] Contemporary sometimes referred to as "Structured Concurrency".

978-1-6654-4066-0/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPS49936.2021.00044

```
unsigned fib(unsigned n) {
  if (n < 2) return n;
  unsigned a = spawn fib(n - 1);
  unsigned b = fib(n - 2);
  sync;
  return a + b;
}
```

Listing 1: Fibonacci function with concurrency keywords

While this may be seen as an undesirable restriction at first, it does not practically limit the achievable parallelism of an application, if the application contains a sufficient level of concurrency. It does, however, simplify reasoning about the program, thus enabling the derivation of space-, time- and communication-bounds [4].

Listing 1 shows keywords that a concurrency platform may introduce. Here, the implementation of the calculation of the n-th Fibonacci number is extended by two additional keywords: spawn and sync. Call expressions annotated with spawn indicate that the caller may continue to execute in parallel with the callee. A function whose body contains a spawn statement is called a *spawning function*. All potentially spawned tasks of a spawning function join at the sync expression (or at the return from the function). However, the spawn keyword merely expresses the potential for parallelism. The decision to lift this annotated concurrency into real parallelism can be made dynamically at runtime.

If fib() were naively implemented using POSIX threads, then the large amount of created threads would probably harm the system's performance. In contrast, when a concurrency platform is used to parallelise a computation, the programming-language layer and the runtime system cooperate to make ideal use of the available computational resources, without causing overutilisation of the same. This collaboration enables *dynamic task parallelism* and is a decisive characteristic of some concurrency platforms. This relieves developers from pondering about the negative impact of parallel overdecomposition. Examples of concurrency platforms include Cilk-5 [5], MIT's OpenCilk [6], Cilk Plus [7], OpenMP, TBB, X10 [8], and Go. They usually structure parallelism using the fork/join model and employ randomised work-stealing [4].

This paper presents **Nowa**, our **no**n-blocking **wa**it-free approach for tasks coordination in continuation-stealing concurrency platforms. To the best of our knowledge, this is the first wait-free approach. Our implementation of the Nowa runtime system employs randomised work-stealing, allows for dynamic task parallelism and employs a practical solution to the cactus stack problem [9]. In our empirical study, the Nowa runtime system yielded promising results of up to 1.64 × the performance of lock-based runtime systems, on a system with 256 hardware threads.

The remainder of this paper is organised as follows: The next section provides background information about the basic building blocks of concurrency platforms and related work. In Section III we describe the mechanics of continuation-stealing concurrency platforms, which are essential to understand our approach. In Section IV we present our wait-free approach for continuation-stealing platforms. We present an evaluation of our approach in Section V and conclude in Section VI.

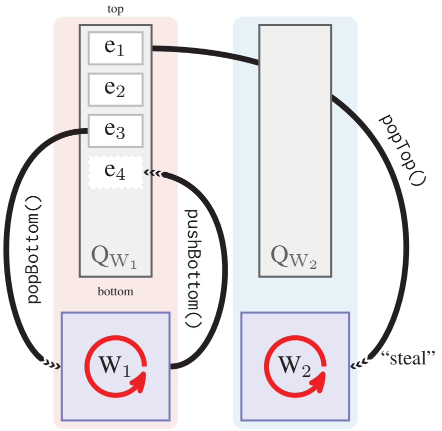


Figure 2: Workers and associated work-stealing queues

II. Background

The properties of randomised work-stealing make it particularly appealing for concurrency platforms. In contrast to scheduling with a global "ready-list", work-stealing provides every worker with a private queue, which other workers can steal from once their queue runs out of work. This way, conflicts due scheduling are distributed over workers, reducing contention and preventing a single hotspot from forming and becoming a bottleneck. Furthermore, there are proven bounds on computation and space requirements, guaranteeing linear speedup for an ideal work-stealing scheduler [4]

The concrete incarnation of a worker depends on the kind of concurrency platform. If the platform's runtime system solely resides in user-space, which is currently the case for most platforms, then workers are implemented based on kernel-level threads. If the platform is based on a spatially-multiplexing operating system [10], then *workers are CPU cores* that are exclusively assigned to the application.

A. Work-Stealing Queues

While every fully-synchronised queue could be used for work-stealing, a queue algorithm exploiting the unique properties of work-stealing is able to increase scheduling performance by reducing contention. We will refer to such queues as *work-stealing queues*.

One can summarise the common properties of work-stealing queues as follows: First, they are double-ended queues (dequeues), where one end is referred to as *bottom* and the other as *top*. The operations provided by work-stealing queues on their ends are not symmetric: While it is possible to remove items from both ends, it is only possible to append at the bottom of the queue. The bottom end is used exclusively by the queue's worker—remember that there is a queue for every worker—in a stack-like manner, where items are pushed to and popped. Consequently, the operations are named pushBottom() and popBottom(). Work-stealing attempts are solely performed on the other end of the queue, the top end. Thieves use the popTop() operation to steal work from other workers' queues, the victims. Figure 2 depicts the anatomy of work-stealing queues for thief and victim.

All queue operations must be partially multithread-safe. That is, popTop() can be used concurrently with itself and at most one of the two bottom operations on the same queue instance. However, since pushBottom() and popBottom() are only ever used by the same worker, work-stealing queues do not need to support concurrent bottom operations. Work-stealing queue algorithms exploit these properties.

B. Child-Stealing versus Continuation-Stealing

One of the most important aspects of a concurrency platform is the employed work-stealing scheme [11]. It can be either child-stealing or continuation-stealing (Figure 3), and influences, among other things, the order in which tasks are executed.

In *child-stealing*, a spawning function makes its child tasks available to be stolen. The example in Figure 3b shows that, at the fork point, W_1 pushes its child task τ_c into its work-stealing queue and continues executing τ_p. This allows W_2 to steal the child task τ_c from W_1. While this scheme is very intuitive, it has multiple drawbacks. First, due to the potentially large amount of child tasks, those may need to be dynamically allocated. Hence the runtime system inherits the performance characteristics of the dynamic memory allocator, which often employs locks. Secondly, child-stealing tends to cause more frequent stack switches, which are expensive.

With *continuation-stealing*, the runtime system offers the continuation after the spawned child task to thieves. While this may appear as a subtle change, it has a strong impact. Most notably, if the continuation is not stolen, which is the typical case, the worker can proceed without a stack switch (Figure 3c). Furthermore, since, in fully-strict continuation-stealing, there is only at most one pending continuation per spawning function, the storage for the continuation instance can be allocated on the stack without requiring the dynamic memory allocator. Figure 1 shows an example where a platform that employs child-stealing, TBB, performs much worse than the continuation-stealing ones.

C. The Cactus-Stack Problem

The last missing central piece of fork/join concurrency platforms is the *cactus stack* [12], a special parallel call stack that forms as a result of the parent-child relationship between function instances and their stack frames, respectively. Stack frames of forked off functions are allocated on separate linear stacks than their parent stack frame but maintain a reference to it. This forms a tree-like data structure, where the linear stacks are the nodes, and the references are the edges.

Suspended stack frames (stack frames of function instances waiting at a sync point) at the bottom of a linear stack cause the stack to be blocked. Consequently, the worker executing on the blocked stack has to use a new empty stack for executing another task, until the sync condition of the suspended stack frame is fulfilled, and the frame's execution can continue.

In theory, this can lead to an impractically large stack space (S_P) consumption, with $S_P \leq DPS_1$ pages of physical memory for a given program, where P is the number of threads used, S_1 is the maximum number of pages of stack space used by the serial execution of the program, and D is the forking depth, the maximum possible number of spawn points on any path from the root to a leaf in the cactus stack. Attempts to find a solution for the cactus stack with a tighter bound for memory consumption, however, resulted in either sacrificing performance or interoperability between serial and parallel code. Finding a solution that satisfies all three criteria – low memory footprint, high speedup, and interoperability between serial and parallel code – is known as the *cactus stack problem* [13].

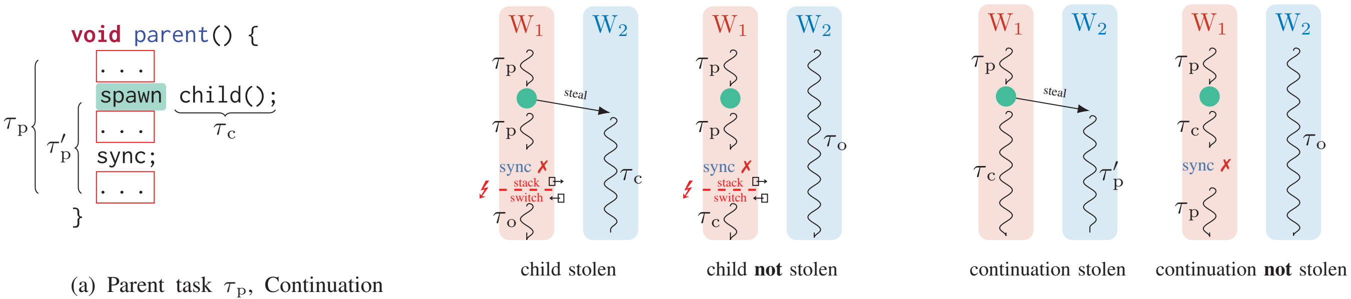
D. Related Work

To solve the cactus-stack problem, Yang and Mellor-Crummey presented a "practical solution" to the cactus stack problem, which they implemented in their runtime system *Fibril* [9]. Whenever a stack frame is suspended, the operating system is informed that the stack space below the suspended stack frame is not used anymore and physical pages are freed, while the virtual mapping is preserved, thus lowering only the *physical* space consumption. Cilk-5 [5] forwent serial-parallel interoperability to limit space consumption in favour of good performance. Cilk Plus [7] does not limit interoperability, but the number of usable stacks, thereby preventing workers from stealing once the limit is reached. Cilk-M [13] uses thread-local memory mappings to build cactus stacks that workers perceive as linear stacks.

Tapir [14] pushes the knowledge about concurrency established by fork-join parallelism into the compiler's intermediate representation to enable further optimizations. Acar *et al.* present a dynamic Scalable Non-Zero Indicator to coordinate nested parallelism [15]. While their approach is not limited to fully-strict parallelism, it depends on dynamic memory allocation.

One of the earliest described work-stealing queue algorithm is the *THE queue*[2] used by Cilk-5 [5]. It is implemented as a bounded buffer with top and bottom indices. The THE queue already exploited the partial thread-safety requirement to elide locking if the queue's top and bottom pointer are non-conflicting. That is, the top and bottom indices do not refer to the same queue element. Later, Arora *et al.* presented a non-blocking work-stealing queue algorithm, henceforth referred to as *ABP queue* [16]. The ABP queue uses atomic compare-and-swap (CAS) instructions to synchronise concurrent queue operations lock-free. Every popTop() attempt requires an atomic operation to arbitrate the access. However, just as the

[2] This queue algorithm is referred to as "THE protocol" in the according publication [5]. THE is an acronym for **T**ail, **H**ead, and **E**xception.



```
void parent() {
    ...
    spawn child();
    ...
    sync;
    ...
}
```

(a) Parent task τ_p, Continuation (of parent task) τ'_p, and Child τ_c

(b) Child-Stealing

(c) Continuation-Stealing

Figure 3: Child-Stealing versus Continuation-Stealing

THE queue can elide locking, the ABP queue is able to omit the CAS instruction for popBottom(). Finally, pushBottom() never requires any atomic instruction. Modern architectures, however, may require a memory fence.

The main disadvantage of the ABP queue is that its effective capacity may decrease dynamically. Since the underlying array is not used as a ring buffer and pushBottom() and popTop() only ever increment the indices, space freed by popTop() cannot be used for adding new elements. The ABP queue mitigates this by resetting the top and bottom value if it becomes empty. However, the reduced effective-capacity condition may persist for an extended period until the mitigation mechanism kicks in. In 2005 Chase *et al.* presented a work-stealing queue [17] that does not have this drawback. Their queue algorithm, which we will refer to as *CL queue*, is based on 64-bit counters that double as generation counters and indices for a ring buffer.

The publications introducing the THE, ABP, and CL queue only briefly consider memory models and, thus, required memory barriers. The first CL queue algorithm utilizing C11's memory model was published in 2013 [18] to the best of our knowledge. Shortly after, Norris *et al.* discovered a bug in the published implementation using model checking [19]. Lock-free data structures are often implemented using overly strong ordering parameters [20]. Concurrency platform designers should strive to use relaxed memory barriers where they are safe, which can significantly increase performance.

III. Continuation-Stealing Concurrency Platforms

A. The DAG Model of Continuation-Stealing

The conceptual link between a concurrency platform's programming-language layer and its runtime system is a directed acyclic graph (DAG). We use the following DAG model based on three types of vertices:

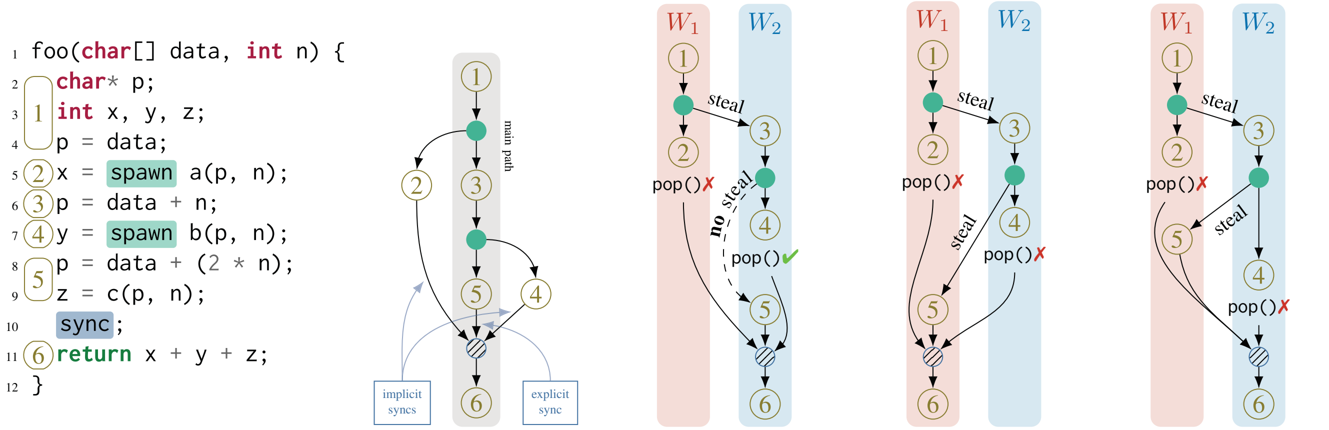
○ *Strand Vertices* for a strand of serial execution.
● *Spawn Vertices* for a spawn-point within the program.
▨ *Sync Vertices* for a sync-point within the program.

Every strand consists of a sequence of one or more instructions not involving any parallel control. Hence a strand vertex never forks into two sub-strands. Instead, such forks are represented by spawn vertices. Just like spawn at the programming-language level, a spawn vertex merely denotes the possibility that its sub-strands could be executed in parallel. Sync vertices ensure that the *sync condition* holds before allowing the control-flow to proceed past them. In fully-strict parallelism, the sync condition is that the number of active parallel strands N_τ, spawned by the current spawning function, is zero: $N_\tau = 0$. Since there is already an active strand upon entering the spawning function, the maximum value of N_τ is the number of spawn call sites plus one. Due to the dynamic nature of spawn points, N_τ is only incremented if a continuation is actually stolen, and hence might never exceed its initial value.

Figure 4a shows an example of a function using spawn and sync to express concurrency. The function's DAG is shown in Figure 4b, where the spawn and sync vertices appear as filled dots. In our model, every spawn vertex has an indegree of 1 and an outdegree of 2, and every sync vertex has an indegree of > 1 and outdegree of 1. The two outgoing edges of a spawn vertex consists of precisely one *child edge* and one *continuation edge*. A child edge leads to a strand representing a spawned function, whereas a continuation edge leads to the control-flow continuation after the spawning function call. Every spawning function's DAG also has one *main path*: The path beginning from the function's start vertex to the end vertex only following continuation edges.

B. Dynamic Strand-to-Worker Mapping

The concurrency platform's runtime systems dynamically maps a fork/join program's DAG to the available workers. During runtime, any worker processing a spawn point pushes the continuation after the spawning call site into its work-stealing queue, admitting the continuation to be stolen by other workers. The worker then calls the spawned child function using the standard calling convention. Once the child function returns, the worker attempts to pop work from its work-stealing queue using popBottom() (Figure 5, line 4). If the pop operation returns a work item, then, due to the LIFO nature of the bottom end of work-stealing queues, this item *must* be the continuation that the worker previously pushed. The worker can now safely discard this continuation and proceed with the control-flow because it leads directly to the discarded



```
foo(char[] data, int n) {
  char* p;
  int x, y, z;
  p = data;
  x = spawn a(p, n);
  p = data + n;
  y = spawn b(p, n);
  p = data + (2 * n);
  z = c(p, n);
  sync;
  return x + y + z;
}
```

(a) Spawning function foo() (b) DAG of foo() (c) Potential Mapping 1 (d) Potential Mapping 2 (e) Potential Mapping 3

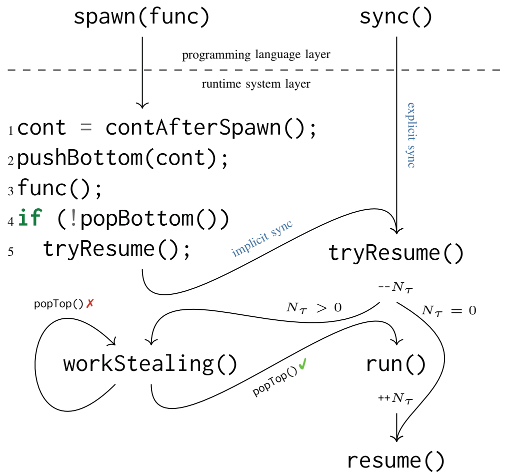
Figure 4: The spawning function foo(), its DAG, and resulting potential strand-to-worker mappings (further exists)

continuation. If the pop operation returns no work item, then the continuation must have been stolen. In this case, the worker has to perform an *implicit sync* operation (Figure 5, line 5).

Figures 4c to 4e demonstrate the outcome of three possible sequences of events resulting from the execution of foo() (of Figure 4a), by showing the according strand-to-worker mapping. In *Figure 4c*, the second worker, W_2, steals continuation #3 from the first worker (W_1). Upon finishing executing a(), W_1 attempts to pop the continuation #3 from its work-stealing queue, which he just pushed into the queue before executing 2. W_1 finds its queue empty, i.e. popBottom() (denoted in Figure 4 as pop() for brevity) does not return the continuation: W_1 now knows the continuation was stolen and performs an implicit sync operation. This operation yields a negative result, which means that the worker is now out of work and needs to resort to work-stealing. Meanwhile W_2 executed the stolen continuation #3 and then, after pushing continuation #5, spawned b() (strand #4). Upon returning from b(), popBottom(), invoked by W_2, yields the continuation 5, which the worker discards and simply continues. After finishing #5, the control-flow leads W_2 into an explicit sync point. Here W_2 finds that W_1 already finished the only spawned function a(), and the sync point is ready to proceed.

In *Figure 4d*, W_2 steals continuation #3. However, unlike the previous example, W_1 steals continuation #5 from W_2. This leads to W_2 finding its work-stealing queue empty after executing b(), resulting in an implicit sync, which returns a negative result and hence leads W_2 to the quest for work. W_1 performs an explicit sync operation due to encountering the sync statement after executing 5, which returns a positive result: W_1 is free to proceed with strand 6.

Finally, in *Figure 4e*, nearly the same sequence of events, as in Figure 4d, happened. However, W_2 spend slightly more time with strand #4. Thereby strand #4 becomes the last finishing strand before the sync point. This leads eventually to W_2 being the worker where the sync condition holds. Thus, strand #6 after the explicit sync point is executed by W_2, not by W_1, as in Figure 4d.



```
1 cont = contAfterSpawn();
2 pushBottom(cont);
3 func();
4 if (!popBottom())
5   tryResume();
```

Figure 5: Coordination logic within the runtime system and the programming-language layer interface

The typical function interaction of a continuation-stealing runtime system is shown in Figure 5, which shows the central role of tryResume(): Every invoking control flow has either reached an explicit or implicit sync point. Hence one of the actions performed in this function is to decrement N_τ. Then, the resulting value of N_τ is inspected to see if the sync condition holds. Depending on the outcome, the worker either resumes execution after the explicit sync point, or goes over to steal work. If the sync condition is not satisfied at an explicit sync point, the spawning function's stack is suspended (cf. Section II-C). As soon as popTop() returns a continuation while performing work-stealing, the worker invokes the continuation's run() function to resume the same. Since this means that a new parallel task has been forked, N_τ is incremented by run() before calling the continuation's

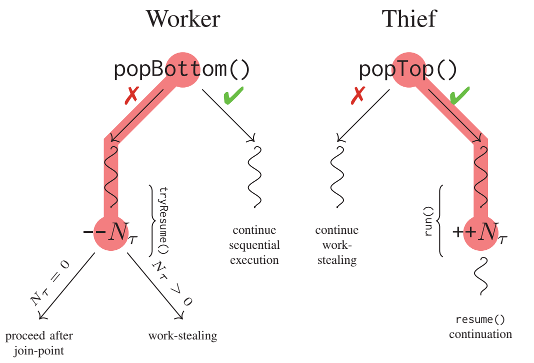


Figure 6: The data race between worker and thieves

```
1  void random_steal(worker_t * w) {
2    worker_t * victim; fibril_t * f;
3    while (1) {
4      victim = workers[rand() % NPROCS];
5      lock(victim->deque);
6      f = steal(victim->deque);
7      if (f == NULL) unlock(victim->deque);
8      else break;
9    }
10   lock(f); unlock(victim->deque);
11   if (f->count++ == 0) {
12     f->count += 1;
13     f->stack = victim->stack;
14   }
15   unlock(f);
```

Listing 2: Excerpt showing the usage of locks in Fibril [9]

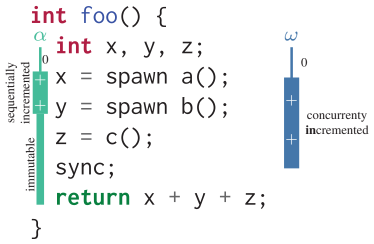
resume() function.

C. The Data Race between Worker and Thieves

Within the just described scheme, employed by continuation-stealing runtime systems, there is a *data race* that is not obvious at first glance: Any pop operation on a queue and the potential subsequent modification of N_τ needs to be performed atomically. Figure 6 shows the critical sections that need to be guarded by a mutex to eliminate the data race. Otherwise, a worker who found the queue empty after calling popBottom() may observe N_τ to be zero after decrementing it, while there are, in fact, still active parallel tasks. Such a worker would assume that the sync condition holds when this is not the case.

This could happen because a thief may already have stolen the continuation from the worker's queue, using popTop(), but did not yet increment N_τ. As a result, a worker may erroneously assume that the sync condition holds and continues after the sync point. After that, the outcome of the program execution is undefined.

Listing 2 shows how the runtime system of Fibril [9] prevents this data race. It uses two locks, one for the queue and one for the data structure containing N_τ (fibril_t.count in Fibril's case). However, both locks are briefly simultaneously acquired in line 10 of Listing 2, combining the two individual locks into the large critical section, displayed in Figure 6.



```
int foo() {
  int x, y, z;
  x = spawn a();
  y = spawn b();
  z = c();
  sync;
  return x + y + z;
}
```

Listing 3: Spawning function anatomy wrt. α and ω

IV. Wait-Free Synchronisation of Strands in Concurrency Platforms

We now present Nowa, our wait-free approach to strand coordination in continuation-stealing runtime systems. The core idea of Nowa is the transformation of the hazardous race condition presented in Section III-C into a benign one. This transformation lifts the requirement for acquiring a lock during the critical sections shown in Figure 6, allows to modify any counter via atomic instructions and, ultimately, turns the **whole operation wait-free** [2]. We will show that one can initialise the counter used by the sync-condition with an arbitrarily large value and later reset it to N_τ at explicit sync points, enabling the transformation of the race condition.

A. Decomposition of N_τ

One central insight Nowa is based on is that N_τ of a spawning function can be decomposed into two counters. The first being the number of actually *forked tasks* α, the second being the number of *joined tasks* ω. At any point in time, the number of active parallel tasks is equal to the number of forked tasks minus the number of joined tasks, which leads to Equation 1.

$$N_\tau = \alpha - \omega \tag{1}$$

Listing 3 shows the anatomy of a spawning function. We observe that α is only ever incremented after the first spawn statement until the last spawn statement. After that, the value of α becomes effectively constant. On the other side, ω is also initially zero and will be potentially incremented from the first spawn statement until a subsequent explicit sync statement.

B. The Wait-Free Nowa Approach

Nowa introduces a new variable ${N_\tau}'$ defined as

$${N_\tau}' = I_{\max} - \omega \tag{2}$$

where $I_{\max}$ is the maximal value of the datatype of the sync-condition counter. During the first phase, before the explicit sync point is reached, Nowa uses ${N_\tau}'$ instead of N_τ as the sync-condition counter. Since ω is zero at the start of a spawning function, Nowa initialises the sync-condition counter with $I_{\max}$. As before, every time a parallel task finishes, the sync-condition counter is decremented. However, since it was initialised with $I_{\max}$, workers will not erroneously observe the sync condition. Once the explicit sync point is reached, Nowa will restore the value of N_τ in the sync-condition counter, henceforth allowing workers to observe the sync condition.

Nowa exploits four invariants found in fully-strict fork/join parallelism. Invariant **I**: N_τ can only become zero once the explicit sync point is reached. Until then, there is at least one task still running: The one heading towards the explicit sync point. Invariant **II**: α is only ever incremented by the same single control-flow (which may be executed by different workers, but never in parallel) and hence does not need to be synchronised. The α incrementing control flow is along the *main path* (as defined in Section III-A). Invariant **III**: Once the explicit sync point is reached, no more steals will happen, and α becomes immutable. Workers only fork new parallel tasks along the main path, and once the explicit sync point is reached, the only control-flow on the main path is suspended. Invariant **IV**: Tasks trying to join with their peers only need to know if there are still active parallel tasks. The exact number of outstanding active tasks, however, is irrelevant for this operation. Joining tasks are only interested in a *boolean is-positive indication* of N_τ.

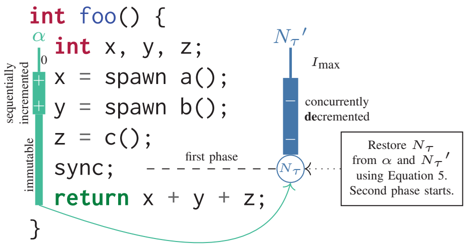
It is sufficient for workers in the first phase to observe any positive value at the sync-condition counter due to Invariant **I** and Invariant **IV**. Since Nowa initialises the counter with $I_{\max}$, a non-positive value could only be observed if more than $I_{\max}$ tasks spawned. As soon as the second phase is entered, by reaching the explicit sync point, we need to restore the actual value of the counter of active parallel tasks (N_τ) to give workers a chance to observe the sync condition. Note that N_τ can be safely used as the value for the sync-condition counter within the second phase, as the race condition between workers and thieves no longer exists because of Invariant **III**.

To see how the restoration of N_τ is possible, we have to complete Equation 1 by $I_{\max}$, yielding Equation 3. Now we solve this equation for N_τ.

$$\begin{aligned} N_\tau + I_{\max} &= \alpha - \omega + I_{\max} && (3) \\ N_\tau &= \alpha - \omega + I_{\max} - I_{\max} \\ &= I_{\max} - \omega - I_{\max} + \alpha \\ &= (I_{\max} - \omega) - (I_{\max} - \alpha) && (4) \\ &= N_\tau{}' - (I_{\max} - \alpha) && (5) \end{aligned}$$

By substituting $I_{\max} - \omega$ with $N_\tau{}'$, as per Equation 2, we obtain Equation 5, which can be used to restore N_τ from $N_\tau{}'$ and α upon entering the second phase.

Nowa's runtime system creates a stack object for every called spawning function. This object contains the state required for the coordination of the potentially spawned strands. Instead of a mutex and a counter for N_τ, Nowa's state includes a field for α and an atomic integer member for the sync-condition counter. As per Invariant **II**, the type for α does not need to be atomic. The sync-condition counter is initialised by the runtime system with $I_{\max}$, because the sync-condition counter represents $N_\tau{}'$, not N_τ, in the first phase. This initialisation happens together with the initialisation of the other members upon entering the spawning function.



```
int foo() {
  int x, y, z;
  x = spawn a();
  y = spawn b();
  z = c();
  sync;
  return x + y + z;
}
```

α
0
sequentially incremented
immutable
$N_\tau{}'$
$I_{\max}$
concurrently **de**cremented
first phase
N_τ
Restore N_τ from α and $N_\tau{}'$ using Equation 5. Second phase starts.

Listing 4: Spawning function anatomy wrt. α, N_τ and $N_\tau{}'$

Listing 4 shows how our wait-free approach uses $N_\tau{}'$ as the initial value of the sync-condition counter. Joining tasks are atomically decrementing its value, before checking if the sync-condition holds. Within the first phase, workers will observe a positive sync-condition counter. Once the control flow reaches the explicit sync point, the value of N_τ is restored atomically in the sync-condition counter using Equation 5. Subsequent checks for the sync-condition will now be based on N_τ.

C. Summoning Synergy Effects

A runtime system of a concurrency platform is a multi layered structure, with the work-stealing queues at its core. The layer wrapping this core is responsible for coordinating the concurrent strands. Most runtime systems use lock-based synchronisation within the outer layer, and the partially-locked THE queue at the core. Nowa removes the locks of the outer layer with its wait-free approach. As a result, all parallel strands can instantaneously and simultaneously break through the outer layer towards the core, where they face the work-stealing queue. To avoid those queues from becoming a bottleneck, we also use a highly-optimised variant of the CL queue within the Nowa runtime system. As explained in Section II-A, unlike the typically used THE queue, the CL queue is completely lock-free. Ultimately, due to the combination of the wait-free Nowa approach and the lock-free CL queue, a synergy effect manifests.

V. Evaluation

We implemented Nowa in a runtime system and compared Nowa with Fibril, Cilk Plus, and TBB. We used Fibril as starting point for Nowa, so the comparison with Fibril shows the direct impact of our wait-free approach. Cilk Plus and TBB are state of the art concurrency platforms and, therefore, put the performance of Nowa into context.

The twelve benchmarks described in Table I were used to evaluate and compare the performance of Nowa, Fibril, Cilk Plus and TBB. These benchmarks have been used in previous publications about the cactus stack problem and Cilk and are adopted from Fibril. The source lines of code (SLOC) were counted using D. Wheeler's *SLOCCount* tool.

Benchmark	Input	Description	SLOC
cholesky	4000/40000	Cholesky factorization	454
fft	2^{26}	Fast Fourier transformation	3054
fib	42	Recursive Fibonacci	40
heat	4096×1024	Jaccobi heat diffusion	149
integrate	10^4 ($\epsilon = 10^{-9}$)	Quadrature adaptive integration	59
knapsack	32	Recursive knapsack	164
lu	4096	LU-decomposition	269
matmul	2048	Matrix multiply	114
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	621

Table I: Description of the 12 benchmarks

All benchmarks were run on a NUMA system with two AMD EPYC 7702 CPUs at 2.00 GHz with boost enabled up to 3.35 GHz. Each CPU package has its own NUMA-node and 64 cores with 2-way simultaneous multithreading (SMT), totaling 256 hardware threads. The total available main memory was 503 GiB. The evaluation was performed using Ubuntu Linux 18.04 with kernel version 4.15.0, GCC 7.5.0, Intel Cilk Plus RTS 7.4.0, and TBB 2017 Update 7. All code was compiled with `-O2`, as the effects of a higher optimization level are often indistinguishable from random noise [21]. The execution time measurements were performed from within the applications to avoid the inclusion of OS and loader induced process ramp-up time. Furthermore, we ensured that the system was quiescent during the measurements.

We used the unmodified benchmarks from Fibril except for *knapsack* and *strassen*. In *knapsack* a bug was fixed that resulted in a minimal problem size after the first run. And in *strassen* the serial portion was reduced and dynamic memory allocations were replaced by preallocated memory to minimise deviations due to memory management outside the runtime system's control. Nowa and Fibril were adjusted to not unmap unused stack space, due to stack frame suspension, using `madvise()`. This adjustment allows for a fair comparison with Cilk Plus, OpenMP and TBB, which do employ this technique. Furthermore, our results in Section V-B question the worthwhileness of this technique.

The evaluation was performed using 4 KiB memory pages and 1 MiB stacks. Besides the execution time using n threads (T_n), we also measured the serial execution time (T_S) using the serial elision [5] of every benchmarked application. For every combination of n (respectively S) and runtime system, we ran the benchmark 51 times in total, with the first run being a warm-up run. To compare the runtime systems via their speedup values, we calculated the 50 serial executions' arithmetic mean per benchmark ($\bar{T}_S$). This mean serial execution time is used to determine 50 speedup values ($\mathscr{S}_n^1 \cdots \mathscr{S}_n^{50}$) for each runtime and benchmark using n threads, that is:

$$\mathscr{S}_n^m = \frac{\bar{T}_S}{T_n^m}$$

From those, we finally calculate the speedup's geometric mean $\overline{\mathscr{S}}_n$ and standard deviation, shown as line point and

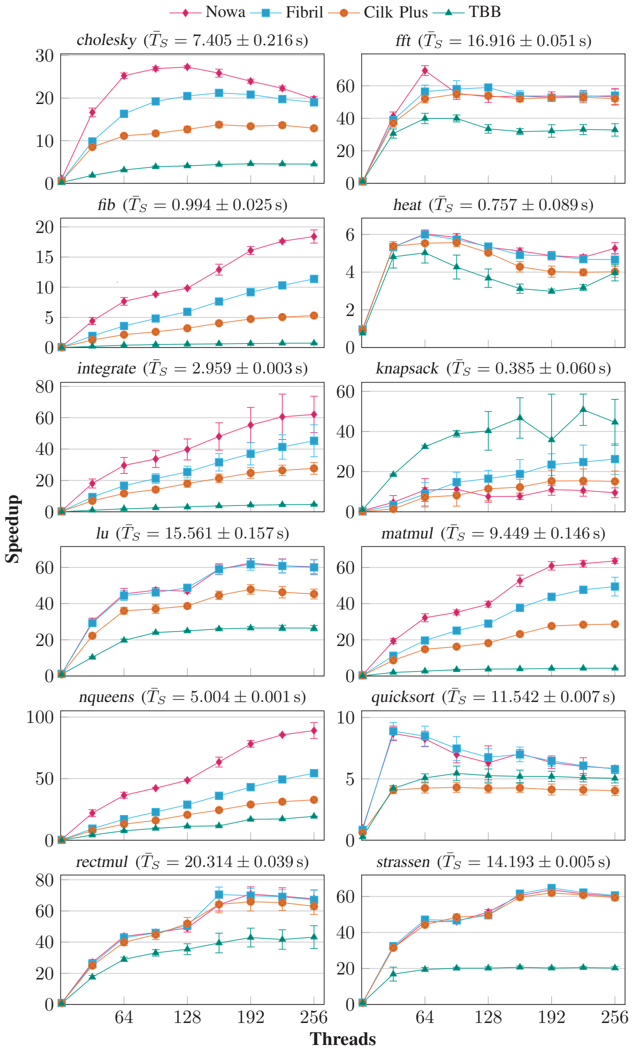


Figure 7: Speedup using 1–256 hardware threads

error bars in the graphs. Since we take the identical value for the serial execution in the speedup calculation across all runtime systems, the derived speedup values are comparable between the runtime systems. A higher speedup of a runtime A than runtime B also denotes a lower execution time than B. Hence, when we provide values for the average performance changes between runtime systems, we calculate the geometric mean of the speedup ratios of the related runtime systems.

A. Comparison of Runtime Performance

Figure 7 shows a comparison of the speedups achieved by Nowa, Fibril, Cilk Plus, and TBB on 1–256 threads. Nowa performs very similar to Fibril on 256 threads in *cholesky*, *fft*, *heat*, *lu*, *quicksort*, *rectmul*, and *strassen*, indicating that the benchmark itself or the data processed might be the bottleneck in these benchmarks. In the case of *cholesky*, the achieved speedup drops down from around 28 on 128 threads to about 20 on 256 threads. Tests have shown that this benchmark relies heavily on allocating and deallocating the stacks used to compute the work. Nowa and Fibril use small per worker buffers of stacks and a global pool to recirculate stacks that changed ownership in the course of work-stealing. When put under stress by many workers, this single global pool can become a bottleneck and limit the achievable speedup. Improvements to the pool can dampen performance derogation in such instances. Furthermore, Nowa achieved speedups roughly comparable to Cilk Plus in *fft*, *rectmul*, and *strassen*, and to TBB in *quicksort*, on 256 threads.

The *knapsack* benchmark is the only benchmark where Nowa performed significantly worse than Fibril, Cilk Plus, and TBB. On 256 threads, the performance of Nowa was 0.36 × that of Fibril, 0.63 × that of Cilk Plus, and 0.21 × that of TBB. However, this benchmark tries to solve the 0/1 knapsack problem by employing a branch-and-bound technique. Hence it spawns a new task for every branch to traverse the branches of the search tree recursively. To bound the number of traversed branches, the recursion stops at branches where an approximation shows that no better result than the current best can be found. Therefore, the required work until the algorithm stops (and thus, the execution time) depends heavily on the executed tasks' ordering. As explained in Section II-B, the employed work-stealing strategy influences the ordering. For example, TBB uses child-stealing and executes forked-off functions in reverse order. Once we switch the order of spawn statements in *knapsack*'s source code, the continuation-stealing runtime systems, including Nowa, outperform TBB. That means that *knapsack* would benefit from a code annotation hinting the concurrency platform which strand of execution to prefer after a spawn. It is, to our knowledge, the only benchmark from the evaluated benchmark suite where this is the case. Nevertheless, for comparison, we decided to show the results using the initial order of spawn statements. The random nature of randomised work-stealing also influences executed tasks' ordering and leads to high variance of run-time in this benchmark. We also found that the memory layout of the *knapsack* executable has a tremendous impact on performance, changing the benchmark's run-time by up to an order of magnitude.

Nowa outperforms Fibril on 256 threads in *fib*, *integrate*, *matmul*, and *nqueens* by 1.62 ×, 1.37 ×, 1.29 ×, and 1.64 ×, respectively. In the *fib*, *integrate*, and *nqueens* benchmarks, the work of a created task is very small, and there is no shared data, making the runtime system the limiting factor of the achievable speedup. Because of this, it is a useful tool for measuring the performance of the runtime system itself.

On average, Nowa improves performance by 1.06 × compared to Fibril, by 1.5 × compared to Cilk Plus, and by 3.02 × compared to TBB, on 256 threads. Since we find *knapsack* to be ill-suited as a benchmark, excluding the results of *knapsack* from calculating the average performance differences between runtime systems is more conclusive. Therefore, we exclude *knapsack* form now, when providing the average speedup ratio of two runtime systems. Without the *knapsack* benchmark, the average speedup increase achieved by Nowa over Fibril, Cilk Plus, and TBB is 1.17 ×, 1.62 ×, and 3.84 ×, respectively. Furthermore, the minimum and maximum performance of Nowa was 0.99 × and 1.64 × the performance of Fibril, 1.01 × and 3.47 × the performance of Cilk Plus, and 1.16 × and 25.3 × the performance of TBB.

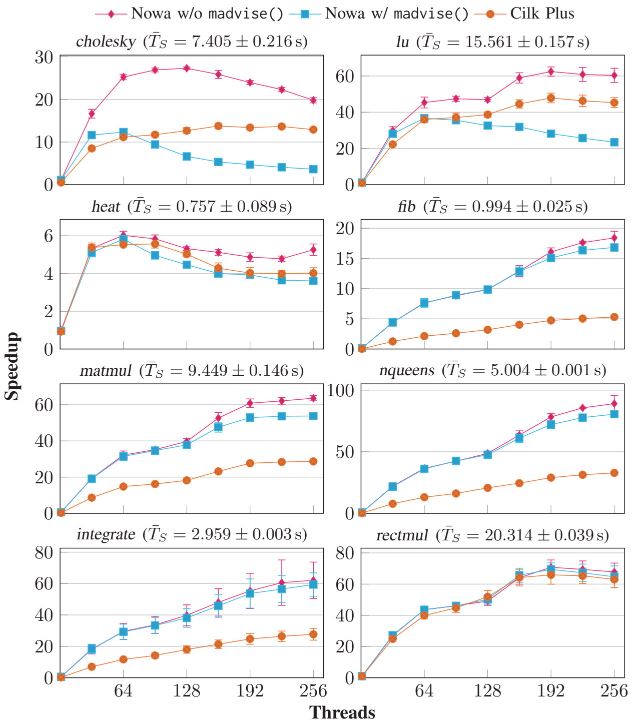


Figure 8: Comparison of the impact of madvise()

B. Analysis of the Practical Cactus-Stack Solution

Yang *et al.* [9] presented a practical solution to the cactus-stack problem by instructing the kernel to free the physical page backing of the unused portion of a suspended stack. This is done using the madvise() syscall with the proper advice argument, MADV_DONTNEED or MADV_FREE in the case of Linux. The original results of the empirical study published by Yang *et al.*, suggested that the use of madvise() with MADV_DONTNEED has no significant impact on performance.

This stands in contrast with our findings. As they are indicating that the use of madvise() is associated with a, in some benchmarks, *significant*, performance penalty. Even the use of MADV_FREE, which allows the kernel to free pages lazily, only improves this by a small margin. Figure 8 shows a comparison of the performance of Nowa with and without the use of madvise() with MADV_FREE. The speedup of Cilk Plus is added for reference. The benchmarks *cholesky* and *lu* showed the highest performance derogation of 0.18 × and 0.39 ×, respectively. On average, the performance decreased by 0.73 ×. Repeating the experiment with Fibril showed similar results with average performance decrease of 0.75 ×.

As shown in Table II, the use of madvise() only lowers memory consumption slightly. Considering the harmful impact on performance, the low memory savings in our benchmarks

	Max RSS (MiB)		
Benchmark	`madvise()`✗	`madvise()`✓	△ **RSS**
cholesky	117	109	−8
fft	2,073	2,073	0
heat	82	82	0
lu	146	145	−1
matmul	63	65	2
quicksort	785	783	−2
rectmul	3,831	3,773	−58
strassen	7,638	7,638	0
fft, *integrate*, *knapsack*, *nqueens*	7	7	0

Table II: RSS usage of Nowa wrt. the use of `madvise()`

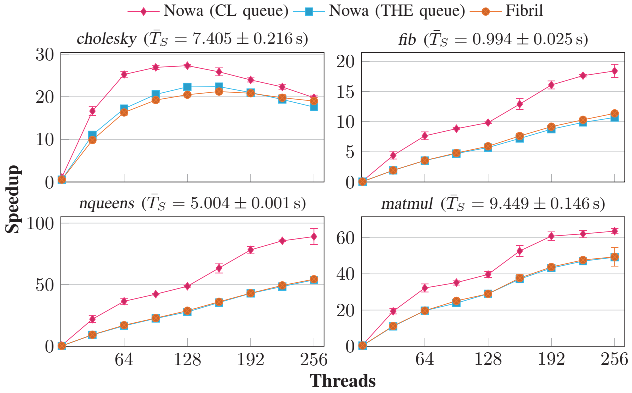


Figure 9: Comparison of the CL queue versus THE queue

and the fact that modern systems have plenty of available main memory, we wonder whether the benefits of unmapping unused stack space justifies its disadvantages.

C. The Work-Stealing Queue's Impact on Performance

As explained in Section IV-C, Nowa generates synergy effects by combining the wait-free approach with the CL work-stealing queue. To investigate the impact of this synergy, we created a second variant of Nowa which uses the THE queue, just like Fibril, instead of the CL queue, but otherwise still employs the wait-free Nowa approach for strand coordination. We then ran the benchmarks with those two Nowa variants.

Figure 9 shows an excerpt of the results of this comparison. If Nowa is using the THE queue, then the performance is comparable to Fibril's. However, as soon as Nowa makes use of the CL queue, the proper performance potential is unlocked in most benchmarks. For some benchmarks, like *fib*, this may even account for a performance increase of up to 1.72 ×. Generally, the performance of our Nowa runtime system using the THE queue was similar to Fibril, or even decreased slightly. This decrease is an indication that the THE queue becomes the bottleneck. However, only the wait-free aspect of our Nowa approach allows the use of the fully lock-free CL queue, which would not be possible without it, thus, removing the bottleneck and unleashing performance.

D. Applicability

Besides Fibril [9], where our approach is applicable since we use Fibril's source code as the starting point for Nowa, we examined Intel's Cilk Plus runtime system [7] as per revision 39ad15a[3], and MIT's OpenCilk [6] Cheetah runtime as per revision cb3d271[4]. Both runtimes use a similar locking approach as Fibril. Therefore we conclude that the scalability of both runtime systems is limited due to the lock and that they would benefit from our wait-free approach.

E. Comparison with OpenMP

As it is a dominant concurrency platform, especially in HPC, we also compared Nowa against OpenMP tasks. For this, we modified the benchmarks to use the `omp task` and `omp taskwait` pragmas, as spawn/fork respectively sync/join primitives.

We included two OpenMP runtime systems in our evaluation: `libgomp` from GCC 7.5.0 and `libomp` from Clang 11.1.0. Every benchmark was compiled with GCC, allowing a fair comparison between all OpenMP and non-OpenMP runtime systems. In the case of `libomp`, this means that we instructed GCC to link against `libomp`. Due to the increased execution times when using `libgomp`, we reduced the number of successive benchmark runs to 13, including one warmup run, and report the values of those 12 runs (instead of 50 + 1 runs). Furthermore, we only examined the execution times using 1, 64, 128, 192, and 256 hardware threads to keep the overall time to carry out the evaluation within reasonable bounds.

To mimic Nowa's and Fibril's runtime systems' behaviour, we extended the `omp task` construct by the `untied` clause. Most continuation-stealing runtime systems do no restrict the target thread after a task's suspension, hence avoiding starvation scenarios. To show the impact of tied and untied tasks, we also evaluated `libomp` using tied tasks.

Figure 10 puts the two OpenMP runtime systems' performance in perspective with the already presented execution times from Nowa and TBB. Unlike Figure 7, we decided to use a logarithmic scale for the plots. Mainly to accommodate `libgomp`'s performance, which is consistently low, often exhibiting a *speedup below one*. While `libomp` does better in terms of speedup, it is slightly below the speedup achieved by TBB. Especially when tied tasks are used, `libomp` is elevated into the performance region of TBB in case of *cholesky*, *fft*, *matmul*, and *rectmul*. Only in *strassen* `libomp` outperforms TBB. For 7 of the 12 benchmarks, using tied tasks seems to improve the performance significantly, even though them being tied to a thread limits the freedom of the runtime system's scheduler. In the case of *strassen*, `libomp` using tied tasks comes close to the speedup of Nowa.

The only benchmark where an OpenMP runtime system is able to outperform Nowa is *quicksort*, which could be due to scheduling artefacts (similar to the case of *knapsack*). But on average, Nowa is 8.68 ×, faster compared to `libomp` using untied tasks, and 5.47 × when using tied tasks.

[3] bitbucket.org/intelcilkruntime/intel-cilk-runtime 39ad15 2018-01-24
[4] github.com/OpenCilk/cheetah cb3d27 2020-08-29

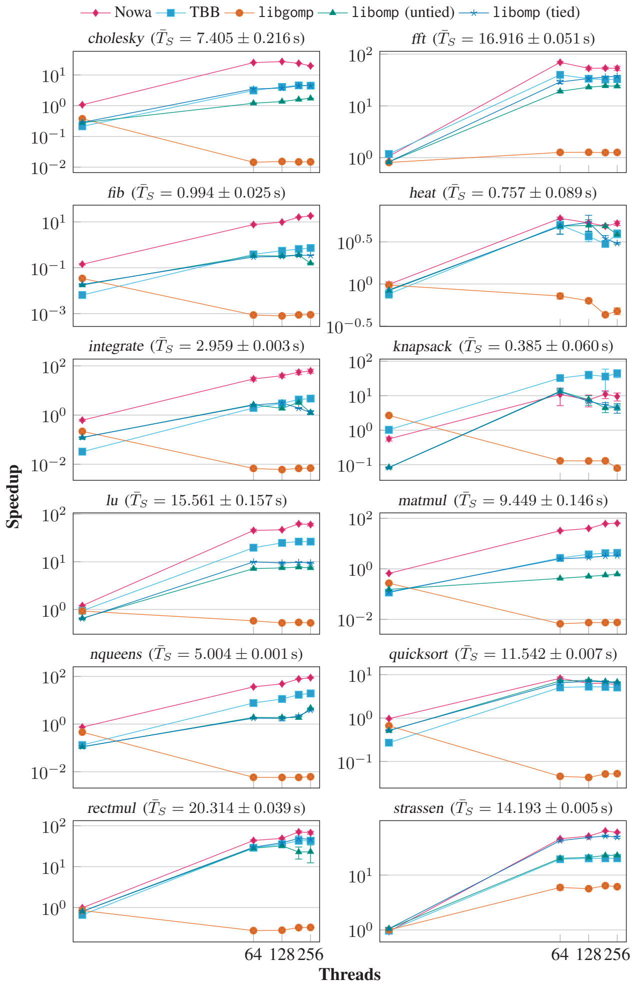


Figure 10: Nowa compared against OpenMP

	Nowa	OpenMP (`libomp`)	
		untied tasks	tied tasks
cholesky	0.37 ±0.01 s	4.28 ±0.06 s	1.66 ±0.01 s
fft	0.32 ±0.07 s	0.71 ±0.03 s	0.45 ±0.02 s
fib	0.05 ±0.01 s	6.30 ±0.13 s	2.86 ±0.04 s
heat	0.14 ±0.01 s	0.20 ±0.01 s	0.25 ±0.01 s
integrate	0.05 ±0.01 s	2.37 ±0.21 s	2.33 ±0.36 s
knapsack	0.04 ±0.01 s	0.09 ±0.02 s	0.09 ±0.03 s
lu	0.26 ±0.02 s	2.09 ±0.10 s	1.64 ±0.06 s
matmul	0.15 ±0.00 s	15.41 ±0.27 s	2.85 ±0.11 s
nqueens	0.06 ±0.01 s	1.07 ±0.03 s	1.32 ±0.11 s
quicksort	1.98 ±0.07 s	1.71 ±0.04 s	1.81 ±0.05 s
rectmul	0.30 ±0.03 s	0.98 ±0.54 s	0.44 ±0.04 s
strassen	0.24 ±0.01 s	0.62 ±0.03 s	0.29 ±0.02 s

Table III: Execution times using 256 hardware threads

To provide a more in-depth insight into the effects of untied and tied tasks, we show in Table III the average execution times and their standard deviations of Nowa and libomp with untied and tied tasks using 256 hardware threads. For *fib*, one of the most challenging benchmarks for a runtime system, Nowa is 116.69 × faster than `libomp` using untied tasks. The time `libomp` requires to execute *fib* more than halves once we switch from untied to tied tasks. Nevertheless, even when using tied tasks, Nowa can execute *fib* 52.92 × faster than `libomp`. It is also worth noting that for some benchmarks, like *heat*, *nqueens*, and *quicksort*, using `libomp` with tied tasks increases the execution time compared to united tasks.

The evaluation reveals that `libgomp` has a severe performance issue when used with fine-grained task parallelism: Nowa can achieve 486.93 × the speedup of `libgomp`, on average. Despite `libomp` performing significantly better than `libgomp`, potentially due to its internal work-stealing scheduling, both OpenMP runtime systems cannot achieve the performance of Nowa, when used for task parallelism.

VI. Conclusion

We presented Nowa, a novel wait-free approach to coordinate concurrent strands in concurrency platforms. Nowa was implemented in a runtime system, which allows for dynamic task parallelism and can employ the practical solution to the cactus-stack problem, all strictly wait-free. Contrary to a previous publication, we find that applying the practical solution to the cactus-stack problem incurs a significant performance penalty. Therefore our evaluation raises the question, whether this solution should be applied at all. Our evaluation shows that Nowa improves the performance by 1.17 × on average compared to Fibril, by 1.62 × compared to Cilk Plus, by 3.84 × compared to TBB, by 486.93 × compared to OpenMP tasks using GCC's `libgomp`, and by 8.68 × compared to OpenMP using Clang's `libomp`, on 256 hardware threads. We further showed that this performance gain is reinforced by Nowa's adept combination of its wait-free approach with a lock-free work-stealing queue implementation based on relaxed memory orderings. Our analysis shows that existing concurrency platforms could also benefit from Nowa.

Artifact Appendix

The source code of the Nowa runtime system and the used benchmark suite is available online:

`https://gitlab.cs.fau.de/i4/manycore/nowa`

It is also available as doi:10.5281/zenodo.4550680. We explictly welcome feedback and bug reports. Our experiments' resulting dataset, which is the base for the presented evaluation, is available as doi:10.5281/zenodo.4550676.

Acknowledgment

We thank Andreas Fried (KIT) for his feedback. This work is partly funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) — Project Number 146371743 — TRR 89 Invasive Computing and under grants SCHR 603/10-2 and NO 625/7-2 (COKE).

References

[1] H.-J. Boehm, "Threads cannot be implemented as a library", *SIGPLAN Not.*, vol. 40, no. 6, Jun. 2005. DOI: 10.1145/1064978.1065042.

[2] M. Herlihy, "Wait-free synchronization", *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, Jan. 1991. DOI: 10.1145/114005.102808.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system", *SIGPLAN Not.*, vol. 30, no. 8, Aug. 1995. DOI: 10.1145/209937.209958.

[4] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing", *J. ACM*, vol. 46, no. 5, Sep. 1999. DOI: 10.1145/324133.324234.

[5] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language", in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98, Montreal, Quebec, Canada: Association for Computing Machinery, 1998. DOI: 10.1145/277650.277725.

[6] T. B. Schardl, I.-T. A. Lee, and C. E. Leiserson, "Brief announcement: Open Cilk", in *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '18, Vienna, Austria: Association for Computing Machinery, 2018. DOI: 10.1145/3210377.3210658.

[7] Intel. (2020). "Intel Cilk Plus", [Online]. Available: http://www.cilkplus.org.

[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing", *SIGPLAN Not.*, vol. 40, no. 10, Oct. 2005. DOI: 10.1145/1103845.1094852.

[9] C. Yang and J. Mellor-Crummey, "A practical solution to the cactus stack problem", in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '16, Pacific Grove, California, USA: Association for Computing Machinery, 2016. DOI: 10.1145/2935764.2935787.

[10] F. Schmaus, S. Maier, T. Langer, J. Rabenstein, T. Hönig, W. Schröder-Preikschat, L. Bauer, and J. Henkel, "System software for resource arbitration on future many-* architectures", in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2020. DOI: 10.1109/IPDPSW50202.2020.00160.

[11] A. Robison, "A primer on scheduling fork-join parallelism with work stealing", *The C++ Standards Committee, Tech. Rep., WG21 paper N*, vol. 3872, 2014.

[12] E. A. Hauck and B. A. Dent, "Burroughs' B6500/B7500 stack mechanism", in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS '68 (Spring), Atlantic City, New Jersey: Association for Computing Machinery, 1968. DOI: 10.1145/1468075.1468111.

[13] I.-T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson, "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*, ser. PACT '10, Vienna, Austria: Association for Computing Machinery, 2010. DOI: 10.1145/1854273.1854324.

[14] T. B. Schardl, W. S. Moses, and C. 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*, ser. PPoPP '17, Austin, Texas, USA: Association for Computing Machinery, 2017. DOI: 10.1145/3018743.3018758.

[15] U. A. Acar, N. Ben-David, and M. Rainey, "Contention in structured concurrency: Provably efficient dynamic non-zero indicators for nested parallelism", *SIGPLAN Not.*, vol. 52, no. 8, Jan. 2017. DOI: 10.1145/3155284.3018762.

[16] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors", in *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '98, Puerto Vallarta, Mexico: Association for Computing Machinery, 1998. DOI: 10.1145/277651.277678.

[17] D. Chase and Y. Lev, "Dynamic circular work-stealing deque", in *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '05, Las Vegas, Nevada, USA: Association for Computing Machinery, 2005. DOI: 10.1145/1073970.1073974.

[18] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli, "Correct and efficient work-stealing for weak memory models", *SIGPLAN Not.*, vol. 48, no. 8, Feb. 2013. DOI: 10.1145/2517327.2442524.

[19] B. Norris and B. Demsky, "CDSchecker: Checking concurrent data structures written with C/C++ atomics", *SIGPLAN Not.*, vol. 48, no. 10, Oct. 2013. DOI: 10.1145/2544173.2509514.

[20] P. Ou and B. Demsky, "Checking concurrent data structures under the C/C++11 memory model", *SIGPLAN Not.*, vol. 52, no. 8, Jan. 2017. DOI: 10.1145/3155284.3018749.

[21] C. Curtsinger and E. D. Berger, "Stabilizer: Statistically sound performance evaluation", in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 219–228. DOI: 10.1145/2451116.2451141.