FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Florian Schmaus

Porting Cilk to the OctoPOS Operating System

Masterarbeit im Fach Informatik

27. Oktober 2015

Please cite as:
Florian Schmaus, “Porting Cilk to the OctoPOS Operating System” Master’s Thesis, University of Erlangen, Dept. of Computer Science, October 2015.

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

Porting Cilk to the OctoPOS Operating System

Masterarbeit im Fach Informatik

vorgelegt von

Florian Schmaus

geb. am 28. Februar 1984
in Bamberg

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Dipl.-Inf. Christoph Erhardt**
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat**
Beginn der Arbeit:	**1. April 2015**
Abgabe der Arbeit:	**15. September 2015**

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.

(Florian Schmaus)
Erlangen, 27. Oktober 2015

ABSTRACT

Concurrency platforms allow developers to utilize all available cores, ideally in an efficient manner. It is becoming more and more important to make use of as many available cores as possible in order to speed up the executed program, because the clock rate of processors is not improving any more. Instead, manufactures are forced to increase the number of cores in their systems in order to stay competitive. Examples of existing concurrency platforms include Intel's Threading Building Blocks (TBB), OpenMP, Java's Fork/Join API, Habanero Scala and the family of Cilk language extensions for C/C++. Cilk is known for its minimally invasive yet powerful extension of C and provably good work stealing scheduling algorithm. Another concurrency platform is OctoPOS, an operating system designed from ground up for future many-core architectures and whose execution model avoids heavyweight threads in an attempt to provide good scalability, low latencies and little jitter. This thesis shows how the Cilk runtime, which is responsible for distributing the parallel task created by Cilk, can be replaced by OctoPOS with the help of a modified LLVM version called CilkO. Developers using CilkO are able to write parallel programs using Cilk keywords instead of having to use the rather complex OctoPOS API. Furthermore OctoPOS is fully aware when a system call will block, thus developers do not have to avoid potentially blocking calls in their code in an attempt to build a deadlock free program. The evaluation of the generated parallel code by CilkO shows good performance and scalability for massive amounts of reasonably sized parallel tasks.

KURZFASSUNG

Nebenläufigkeitsplatformen erlauben es Entwicklern alle verfügbaren Prozessoren, idealerweise effizient, zu nutzen. Diese wurde in den letzten Jahren immer wichtiger, da es den Prozessorenherstellern kaum noch möglich ist die Taktrate ihrer Prozessoren zu erhöhen. Stattdessen sind sie gezwungen immer mehr Prozessoren in ein System zu verbauen, um dadurch die Leistung zu steigern und Wettbewerbsfähig zu bleiben. Beispiele für existierende Nebenläufigkeitsplatformen sind Intels Threading Building Blocks (TBB), OpenMP, Javas Fork/Join API, Habanero Scala und die Familie der Cilk Spracherweiterungen für C/C++. Cilk is bekannt für seine einfach zu verwendende Spracherweiterung und einen ausgezeichneten Algorithmus um Arbeit an die verfügbaren Prozesorren zu verteilen. Eine weitere Nebenläufigkeitsplatform ist OctoPOS, ein Betriebsystem welches von Anfang an für zukünftige Vielkernarchitekturen entwickelt wurde, und dessen Ausführungsmodell schwergewichtige Fäden gänzlich vermeidet, um dadurch gute Skalierbarkeit, kurze Wartezeiten und möglichst geringe Abweichungen zu erreichen. Diese Masterarbeit zeigt wie die Cilk Laufzeitbibliothek, welche für das verteilen der möglichst parallel auszuführenden Arbeit verantwortlich ist, durch OctoPOS ersetzt werden kann. Dies geschieht mit der Hilfe einer modifizierten LLVM Version, genannt CilkO. Entwicklern erlaubt CilkO parallele Programme unter Zuhilfenahme der Cilk Schlüsselwörter zu schreiben, und somit die eher komplexe OctoPOS API zu umgehen. Zudem ist es ihnen erlaubt eventuell blockierende Systemaufrufe zu verwenden, da OctoPOS weiß wenn der Aufruf blockieren wird. Die Auswertung des generierten parallelen Code von CilkO zeigt gute Leistung und Skalierbarkeit wenn eine große Anzahl paralleler Aufgaben mit einer angemessenen Größe abgearbeitet wird.

CONTENTS

1 INTRODUCTION

We are starting to experience the effects of having to pay for our lunch. When Sutter warned us in [Sut05] that "The free lunch is over", he described a future where developers are required to use parallel programming techniques in order to speed up their programs. Until the mid-2000s, developers could afford to be lazy: The steady increase of the clockrate of microprocessors would cause a speed up of their programs. And that speed up came for free, without requiring any assistance or intervention by the developer. But by the beginning of 2005 it was foreseeable that physical limitations would not allow manufactures of microprocessors to further increase the clock rate. Instead, in order to improve their products, they needed to adapt their approach: manufacturers are now forced to increase the core count per processor. This, in turn, forces developers to explicitly write parallel code in order to utilize the available cores.

The number of cores available to a single system is still continuously on the rise. Recent research [Tei+11] suggests that we are on the edge of the shift from multi-core to many-core systems. And since the future will allow for massively parallel MPSoCs (Multi-Processor System-on-a-Chip), we will soon require novel paradigms, frameworks and language extensions in order to allow developers to write concurrent code that takes advantage of the available cores.

Examples for already existing concurrency frameworks, which are also called *concurrency platforms*, on the programming language level include Intel's Threading Building Blocks (TBB), OpenMP, Java's Fork/Join API [Lea00] and Habanero Scala [Ima11][IS12]. Recent development also lead to the creation of programming languages with built-in support of concurrency and distribution. Examples include X10[Cha+05] and Chapel[Cha13]. Meanwhile the trend of adding new concurrency paradigms to existing languages continues. For example just recently Oracle introduced with Java 8 the concept of "Streams" which can be processed in parallel.

Within the research of *Invasive Computing*, introduced by "Invasive Computing: An Overview," the *OctoPOS* operating system emerged. It was designed from ground up for future many-core architectures, with the goal to incorporate the execution model within the OS. This execution model avoids heavyweight threads in an attempt to provide good scalability, low latencies and little jitter. Instead the unit of execution is called *i*-let, which is basically a fine-grained code snippet. OctoPOS implements novel concepts to enable efficient high-level micro-parallelism support by an operating system, and as a result, can be viewed as concurrency platform.

Another prominent concurrency platform is the family of Cilk C-language extensions. Research and development of Cilk started in the 1990s at the MIT Laboratory for Computer Science. At that time, Cilk essentially extended C with keywords to express the fork-join idiom and parallel for-loops. One major contribution of Cilk is the the provably good work stealing scheduling algorithm [FLR98]. This algorithm is used in the runtime system and allows a Cilk program to be executed in parallel on the available processing elements with minimal scheduling overhead. While work-stealing schedulers are known to provide good performance, the Cilk runtime exists as an additional layer between the parallel Cilk program and the regular OS, and thus, imposes overhead.

We noticed that there are many similarities between the execution model of OctoPOS and Cilk's runtime scheduler. What if one replaced the Cilk runtime with OctoPOS? The initial expectation is that removing the runtime layer between the parallel program and the operating system should yield at least a performance advantage. Thus this thesis attempts to provide answers to the following questions:

1. Is it possible to replace the Cilk runtime with OctoPOS?
2. If so, how could this be achieved and implemented?
3. What are the benefits in doing so?

2 FUNDAMENTALS

The following chapter provides a detailed overview of the technologies upon which this master thesis is built. The first section starts with explaining the theory behind how parallelism can be modelled and measured. The chapter then continues with the topmost item of the technology stack, Cilk, followed by an introductory section about the compiler framework used, LLVM. We will have explored our way down the technology stack with the final section about the low-level invasive run-time system *OctoPOS*, which concludes this chapter.

2.1 Parallelism

In order to understand the potential goals and the evaluation results of this thesis, we need to make an excursion into basic concepts of *parallelism*.

The most basic term is the **speedup** S a parallel program can experience when run on a P-processor machine. It is thus defined as the ratio of the execution time of a program run on a single-processor machine T_1 to the program run on a P-processor machine T_P:

$$S = \frac{T_1}{T_P} \tag{2.1}$$

If it is possible to obtain a speedup proportional to P, then **linear speedup** is achieved. If $S = P$, then **perfect linear speedup** is obtained. In the uncommon case where the speedup is greater than P, **superlinear speedup** was achieved. This is possible for example because of cache and other processor effects.

One of the earliest and most fundamental observations of parallelism is that the fraction p of a computation that can be run in parallel provides an upper bound of the possible speedup S. Suppose that 50% of a program can be run in parallel, while the other 50% can not. Then, even with an infinite number of processors, the achievable speedup is at most 2. This is called **Amdahl's Law** (2.2).

$$S \leq \frac{1}{1-p} \tag{2.2}$$

While this provides some insight into what is are theoretically achievable, it does not quantify parallelism, i.e. it does not allow us to determine what concurrency frameworks like Cilk should be able to offer.

A good overview how to quantify, model and measure parallelism is given in chapter 27 of [Cor+09]. The following summarizes the relevant concepts and laws taken from the book.

The *Directed Acylic Graph (dag)*[1] *model of multithreading* [BL93] is often used to analyze parallel programs. As the name suggests, it models a parallel program as a directed acyclic graph. That is, a graph where every edge has a direction and, no matter how one follows those directed edges, it is not be possible to visit the same edge twice, because there are no cycles. Every sequence of instructions of the parallel program that does not include an instruction introducing parallelism is called a **strand** and modeled as vertex of the dag. Edges in the dag indicate dependencies between the instructions.

For the evaluation of my work, it is important to measure parallelism, e.g. how it scales with the number of processors. Two metrics are important to reason about the performance of a parallel program. The first, **work**, denotes the total amount of time spent in all instructions of the parallel program. The second, **span**, corresponds to the amount of time it takes to execute the longest chain of dependencies in the dag. This path in the dag is also called the **critical path** or **depth** of the dag.

We can also define work and span without looking at the instructions of a program: work is the running time of the program on one core and span is the running time the program on an infinite number of cores (Assuming a perfect scheduler without overhead.). Henceforth span is denoted as T_∞ and work as T_1.

The amount of work a parallel program yields is important, since it provides a lower bound on the program's execution time T_P on P processors. Equation (2.3) is called the **Work Law**.

$$T_P \geq \frac{T_1}{P} \tag{2.3}$$

It is trivial to conclude that this law holds if we think of a theoretical model where each processor executes at most one instruction per unit time, which also means that P processors can execute at most P instructions per unit time. This leads to the conclusion that it takes P-processors at least T_1/P time to compute T_1. Rewriting the inequation to $T_1/T_P \leq P$ also gives an upper bound for the speedup on P processors.

Another important boundary is imposed by the **Spawn Law**.

$$T_P \geq T_\infty \tag{2.4}$$

The Equation (2.4) states that an finite number of processors cannot outperform an infinite number of processors, because every machine with an infinite number of processors can mimic a P-processor machine.

[1] Note that although DAG is an acronym, it is usally written in lower case in literature and I decided to follow this convention.

With the definitions of T_1 and T_∞ at hand, we define **parallelism** as the ratio of work to span. Another way to think of parallelism is that it gives the average amount of work along each step of the critical path.

$$\mathscr{P} = \frac{T_1}{T_\infty} \tag{2.5}$$

Assuming an ideal parallel computer (Parallel Random Access Machine, PRAM) with a zero-overhead greedy scheduler, the **Greedy-Scheduling Theorem** [BL99] gives an upper bound for the execution time using P processors.

$$T_P \leq \frac{T_1}{P} + T_\infty \tag{2.6}$$

This equation resembles **Brent's Theorem** (or "Brent's Lemma", "Brent's Law") [Gra69][Bre74]. The first term on the right-hand side of Equation (2.6) is the work term: because of the work law (2.3), the time to execute the computation cannot be any better than $\frac{T_1}{P}$. The second term originates from the critical path.

An important observation found in [BL99][Lei09] is that if the parallelism $\mathscr{P}$ exceeds the number P of processors by a sufficient margin, then Equation (2.6) guarantees nearly perfect linear speedup. This can be easily shown: if we transform the assumption $\mathscr{P} \gg P$ using (2.5) to $T_\infty \ll \frac{T_1}{P}$ we see that the work term dominates the span term. What follows is that the running time is $T_P \approx \frac{T_1}{P}$, which equals a nearly perfect linear speedup $\frac{T_1}{T_\infty} \approx P$.

To summarize: We define **parallel slackness** as

$$PS = \frac{T_1}{T_\infty P} = \frac{\mathscr{P}}{P} \tag{2.7}$$

that is the factor by which the parallelism of a computation exceeds the number of processors. Then we first find if that achieving perflect linear speedup is impossible if $PS < 1$, because using Equation (2.7) we transform this inequation to $\frac{T_1}{T_\infty} < P$. Which means that the parallelism is less then the number of available processors. And also shows that if the slackness is greater then one, then the number of processors becomes the limiting constraint. We also find that the impact of T_∞ on the execution time is decreasing with an increasing parallel slackness.

2.2 Cilk

Most of the programming languages we use today, including C and C++, were not designed with parallelism in mind. The increasing availability of SMP systems in the last decades is accompanied with a rising desire of developers for well-defined and easy-to-use constructs allowing them to express parallelism in their code. Cilk, pronounced silk ("nice threads"), is such a construct, which consists of linguistic extensions ot the C language and a runtime (library). One reason Cilk was the chosen as parallel-programming language extension used in this thesis is the increasing popularity it has gained in the recent years.

2.2.1 History

Work on Cilk started at MIT's Computer Science and Artificial Intelligence Laboratory (CSAIL) Supertech Research Group under the leadership of Prof. Charles E. Leiserson in the early 1990s. The Cilk-1 system was released in September 1994. The latest available version is 5.4.6 and sometimes referred to as MIT Cilk or Cilk-5. One of the various papers about Cilk, "The Implementation of the Cilk-5 Multithreaded Language" by Frigo, Leiserson, and Randall [FLR98] was awarded SIGPLAN's "Most Influential PLDI Paper Award" in 2008 [Acm]. Cilk-5 was the first Cilk flavor which provided simple linguistic extensions such as `spawn` and `sync` to ANSI C.

Cilk Arts, Inc., a venture-founded start-up founded by Leiserson and Frigo, started in 2006 the development of a commercial Cilk implementation supporting C++, called Cilk++ [Lei09]. Besides full support for C++ and exceptions, it removed the restriction that functions going to get spawned are required to be specially annotated as Cilk functions. Also the essential `spawn` and `sync` keywords (see Section 2.2.2) were renamed to `cilk_spawn` and `cilk_sync`, to avoid conflicts with existing labels. Cilk++ also added "reducer hyperobjects" which allow programmers to concurrently access non-local variables in a lock-free manner [Fri+09].

In 2009, Intel Corporation acquired Cilk Arts and merged the Cilk technology with its Array Notation (AN) technology. The result, Cilk Plus, was released by Intel in 2010, while simultaneously making the Cilk Plus specification freely available. Intel announced, two years after the acquisition of Cilk Arts, that it would add Cilk Plus support to GCC, and shortly after, completed the initial implementation in 2012. The same was done with LLVM [Cor14], although the changes where not included upstream. Cilk Plus is also proposed as standard to the C++ standard body.

2.2.2 Language Extensions

In its simplest and purest form, Cilk consists of three new keywords extending the C language:

`cilk_spawn` Used to create parallelism. Functions preceded by this keyword *may* continue to get executed in parallel with the control flow invoking the function, and are called *spawned functions*.

`cilk_sync` Acts as a local barrier and joins together the parallelism forked by `cilk_spawn`. That is, the runtime ensures that all functions spawned before the `cilk_sync` have completed and returned before executing any statement after the `cilk_sync`.

`cilk_for` The parallel counterpart of the looping construct `for` in C/C++. Allows loop iterations to be run in parallel and can be seen as syntactic sugar which mixes `cilk_spawn` and `cilk_sync` together with a for-loop. Using `cilk_for` enforces certain restrictions to the loop's initializer, condition and increment statements.

One important aspect of those keywords is that they do not command parallelism. They only add *logical parallelism*, which means they grant the permission to run certain parts of code in

parallel, not that this parallelism is always created (for example by spawning a thread). Not only is thread creation a relativ heavy task, spawning threads in contemporary programming languages also introduces the hazardous possibility that the cost of constructing and managing the thread is greater than the computation time of the task itself. When Cilk is used most of the overhead introduced with creating parallelism is only imposed when the parallelism is really realized (usally because an idle core/worker has become available). The decision to eventually take the leap to create a new parallel strand is performed by Cilk's runtime and especially its work stealing scheduler (Section 2.2.5).

We are able to map the dag model of multithreading to the programming model established by Cilk's keywords using the following rules: 1. Every `cilk_spawn` of a function creates two edges from the strand before the spawned function: One edge to the first instruction of the spawned function, and the other to the first instruction after the spawned function. 2. A `cilk_sync` creates edges from the final instruction of each spawned function to the instruction immediately after the `cilk_sync`.

```
1 uint64_t fib(uint32_t n) {
2   if (n < 2)
3     return n;
4   uint64_t a = cilk_spawn fib(n-1);
5   uint64_t b = fib(n-2);
6   cilk_sync;
7   return a + b;
8 }
```

Listing 2.1 – The Fibonacci function using Cilk

Listing 2.1 shows an example of a Cilk-enhanced recursive function computing the n-th Fibonacci number. The `cilk_spawn` annotated function spawns the invocation of `fib(n-1)` which is then potentially run in parallel with the execution of `fib(n-2)`. To ensure that the spawned function has been completed and the result is available in variable `a` of line 4 and since the return value of the `fib` function is computed from both results of the recursively invoked `fib` functions, a `cilk_sync` is added before it is computed and the function returns.

2.2.3 Design, Goals and Contributions

The runtime scheduler ensures that Cilk programs are executed in a *faithful* manner, which means that parallel code retains its serial semantics when run on one processor.

Furthermore, simply eliding the Cilk keywords from the source yields the *serial elision*, the ordinary non-parallel C representation of the Cilk enhanced program. For `cilk_spawn` and `cilk_sync` eliding means substituting them with the empty string, and for `cilk_for` eliding means substiuting it with the string "for", so that it becomes the ordinary keyword representing for-loops. Those substituions are done using C preprocessor macros which expand to the desired string.

[BL99] proves that the expected time to execute a parallel program on P processors using Cilk's work-stealing scheduling is bounded as shown in Equation (2.8).

$$T_P = \frac{T_1}{P} + \mathcal{O}(T_\infty) \tag{2.8}$$

The important part of Equation (2.8) is the $\mathcal{O}(T_\infty)$, which says that the total execution time T_P is not only dependent on the number of processors P but also influenced by the length of the critical path. And since this length not only consists of the number of tasks executed within the critical path, but also overhead the scheduler induces to perform this tasks, the T_∞ is wrapped in $\mathcal{O}()$.

Comparing with Equation (2.6) one finds that Cilk's scheduler executes the program within the expected time depending on the number of available processors.

Furthermore Cilk was designed following the *work-first principle*, which states that overhead in the work path should be minimized and scheduling overhead should contribute to the critical path instead of the work path when possible. Therefore Cilk's scheduler guarantees that the cost of stealing contributes only to the critical-path overhead and not to the work overhead. And in order to minimize the work overhead, instead of always using locking, Cilk uses a mutal-exclusion protocol inspired by [Dij65] called the *THE*[2] protocol, which allows the Cilk scheduler to manage the worker's queues of ready jobs using a work-stealing algorithm.

While the work-first principle may sound counterintiutive at first, the reason that it should be followed can be illustrated. First we define the *span overhead* to be the smallest constant $c_\infty > 0$ such that we can rewrite Equation (2.8) to

$$T_P \leq \frac{T_1}{P} + c_\infty T_\infty \tag{2.9}$$

If we now assume a program with $PS \gg c_\infty$, which, if we remeber the defintion of PS from 2.7, translates to the number P of processors being much smaller than the program's parallelism $\mathscr{P}$. It now follows that $\frac{T_1}{P} \gg c_\infty T_\infty$, and we see that the span overhead c_∞ has little effect on T_P if sufficient parallel slackness exists. But it does determine how much parallelism must exist to ensure linear speedup.

To achieve this scheduling performance, most Cilk implementations utilize a *two-clone strategy*. Two helper functions are created for every spawned function. One function, the so called *fast-clone*, is used for the common case, i.e. when no work was stolen. The other one, the *slow-clone* is only executed if the continuation is executed on a worker different from the one which created it. Cilk was inspired using this approach [Lee+10] by the "lazy task creation" strategy of Kranz, Halstead, and Mohr [KHM89]. Cilk Plus employs a similar approach to the two-clone stragegy, also consisting of a slow and fast path, which will be explained in Section 2.2.6.

2.2.4 Stack Memory: The Cactus Stack

When a function is spawned, it needs to share the same view on the already existing stack as the spawning function at the time of the spawn, but needs its own stack space for the new activcation

[2] An acronym for **T**ail, **H**ead and **E**xception.

frame on which the spawned function operates. This means that the existing stack needs to be shared between the spawned function and the continuation, but the expansion of the stack must not be shared. Thus Cilk creates two stack expansions at every spawn by using a a **cactus stack**[3][HD68] for stack-allocated memory, where the extant activation frames form a tree.

Figure 2.1 shows an example cactus stack for five functions A(), B(), C(), D() and E() and the invocation tree shown on the left: First A spawns B and C, then C spawns D and E. All functions have a different view of the stack, but some memory regions of the stack are shared. In this particular example, the memory region of A is shared by all five functions and the region of C is shared by C, D and E.

The space used by a cactus stack is bounded by the number of processors. If S_1 is the space used for the execution of the serial elision of a Cilk program, then the stack space S_P consumed during a P-processor execution satisfies Equation (2.10).

$$S_P \leq PS_1 \tag{2.10}$$

2.2.5 Runtime and Work Stealing Scheduler

Another distinguishing feature of Cilk, besides the language extension, is the runtime used, which includes a provably efficient work-stealing scheduler [FLR98]. It is the duty of the runtime scheduler to dynamically distribute strands to the available processing elements. Hence the scheduler is important to the overall efficiency of the parallel program.

Two basic approaches how to distribute work can be distinguished: **Work stealing** and **work sharing**.

When work sharing is used, tasks are added to a global queue from which all worker threads try to obtain their work. That single queue is also the main disadvantage of work sharing: since all threads add and get their tasks from this queue it becomes the scalability bottleneck. It even gets worse if the tasks become more leightweight, since this means the requests to the queue become more frequent.

Work stealing does not use a global queue and therefore does not suffer from these shortcomings. Instead of a single global queue, every worker maintains its own queue of ready tasks, which is

[3]Also called spagehtti stack, or suguaro stack (after a kind of cactus).

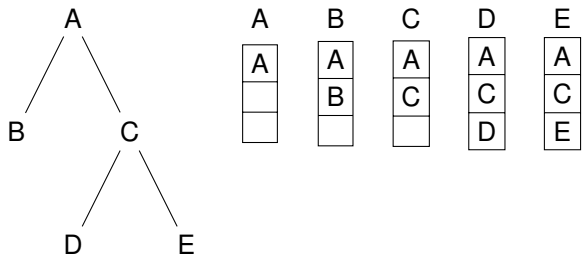

Figure 2.1 – An example Cactus Stack

operated as a stack by the worker. This means that work originating from the worker is pushed to the tail of the queue, from which the worker also pops new work off when it finished the current strand. If the worker's queue is empty, it becomes a thief trying to steal work from the head of the queue belonging to other workers. The queue is therefore used as stack by the local worker and as deque[4] by the thief and victim.

One major advantange of work-stealing is that no task migration between workers is going to happen when all workers have work to perform. And since workers are usually bound to a processor, this increases the efficency gained through the processor's cache. Furthermore work stealing keeps the communication between processors at a minimum.

Another questions that needs to be answered when implementing work scheduling is what the current worker executes after a strand is spawned and which strand is made available to thieves for stealing. The two possiblities are **Child Stealing** or **Continuation Stealing** (sometimes called "Parent Stealing")[Rob14].

```
1 stmt1;
2 a = cilk_spawn fib(n-1);
3 b = fib(n-2);
4 cilk_sync;
5 return a + b;
```

Listing 2.2 – Parallel Cilk Code

Considering Listing 2.2 and Figure 2.2 the concurrency platform could either make `fib(n-1)` available for stealing or the continuation starting just before `fib(n-2)`. In general, child stealing is easier to implement. That is one reason why concurrency platforms designed as library, like Microsoft's Parallel Patterns Library or Intel Threading Building Blocks, use this approach per default.

But child stealing produces a lot parallel tasks in the worker's queue if no stealing actually happens. Imagine executing the code snippet of Listing 2.3 on fully saturated system, i.e. the queues of every worker are sufficiently full, so that no work stealing will happen during the execution of the

[4] A double ended queue

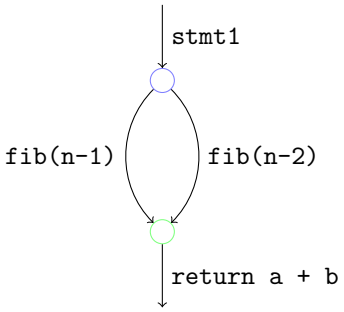

Figure 2.2 – dag for Listing 2.2

snippet. Using child stealing will add n elements to the queue of the worker executing the for-loop until the worker reaches the `cilk_sync`. If this happens it will start working on the tasks from the queue that the worker just added itself.

```
for (int i = 0; i < n; ++i)
    cilk_spawn f(i);
cilk_sync
```

Listing 2.3 – Spawning within a for-loop body

Using continuation stealing in this case means that the worker calls `f(0)` after it created a task for the continuation and placing the continuation's task into its queue. Thus the queue size only increases by a constant factor of 1, compared to n when child stealing is used. The other $n-1$ tasks will be created, one after another, once the continuation gets executed.

In other words, continuation stealing exposes only exaclty one continuation for stealing. This provides the notable advantage that the queue's utilization is minimized, compared to when child stealing is used, resulting in all tasks up to the next `cilk_sync` being created and put into the queue.

2.2.6 Continuation Stealing in Cilk Plus

Cilk Plus implementens an efficent approach to enable continuation stealing using non-local jumps, i.e. `setjmp()` and `longjmp()`. A statement like

```
cilk_spawn foo();
```

will get transformed to the pseudo code shown in Listing 2.4.

```
jmp_buf buf;
if (setjmp(buf)) {
  push(&buf);
  foo();
  bool empty = pop();
  // Leave frame if queue empty. Does not return.
  if (empty) cilkrts_leave_frame();
}
```

Listing 2.4 – Pseudo C Code of Continuation Stealing in Cilk Plus

After the space for the jump buffer is allocated, the stack context is save using `setjmp()`. Notice how a caller using `longjmp()` on the buffer will execute the continuation, since the long jump will return at the `setjmp()` invokation returning a nonzero value, thus omitting the body of the `if` statement. But when `setjmp` is used the first time, i.e. to initialize the jump buffer, it will return

zero, thus executing the `if` statement's body. The body first pushed the pointer to the buffer on top of the work-stealing queue. This essentially exposes the continuation for work stealing. Next the spawned function `foo()` is executed and after it returns the current control flow tries to remove the topmost item of the work-stealing queue (line 5). Since the local worker uses its work-stealing queue in a stack like fashion, there are exactly two outcomes of the `pop()` operation: either the continuation has not been stolen, in which case it has now become unavaiable for stealing and the current thrad of execution continues executing, i.e. it will execute what was the continuation. Or the continuation has been stoolen in the meanwhile, in which case the work-stealing queue has to be empty. This means the current thread of execution needs to abort here, since someone else is going to execute the continuation.

The overhead of a `cilk_spawn` in the case where the continuation is not stolen is thus the sum of the cost of the `setjmp()`, `push()` , `pop()` operation together with the cost of the `if` statement and allocating the jump buffer.

2.3 The LLVM Compiler Infrastructure

The LLVM (formerly known as "Low-Level Virtual Machine") compiler infrastructure provides a highly modular compiler and framework that supports various input languages and target architectures. Started in 2000 as a research infrastructure project at the University of Illinois [Lat02], it gained major traction when Apple Inc. hired the LLVM project lead, Chris Lattner, to incorporate LLVM within Apple's development systems. Another contributing factor to LLVM's success is its clean design and easy to understand structure and code, when compared to competitors like GCC. Unlike GCC, LLVM uses exclusively C++ (instead of mostly C) and is modular (instead of monolithic). This has attracted a lot of people, not only experienced compiler programmers and scientists, but also relatively inexperienced students which choose LLVM as foundation for their research, as it can be seen on the high number of scientific papers involving LLVM. The efforts put into LLVM where rewarded by the ACM, which awarded Lattner, together with professor Vikram Adve and Evan Cheng, the Software Systems Award 2012 for LLVM [CM13], a highly distinguished recognition of notable software that contributed to science.

Compared with other compilation approaches, LLVM's outstanding features include the low-level code representation used. This representation is called *Intermediate Representation* (IR), which is sometimes referred to as "LLVM Assembly" as it shares many similarities with assembly languages. IR provides 1. a language-independent type system that exposes the primitives commonly used to implement high-level language features; 2. useful instructions for typed address arithmetic; and 3. a mechanism that can be used to implement the exception handling features of high-level languages uniformly and efficiently [LA04]. The main motivation for LLVM's IR comes from the idea that optimizations can not only be applied at compile time, but also at installation time, runtime and idle time. The fact that IR can also be stored on disk, usually in a format called *Bitcode*, is a key enabler of lifelong program optimizations and offers an alternative way to encode entire programs.

The following subsections describe the high level architecture of LLVM and the required components which are needed to understand this thesis: the general architecture and compilation pipeline of LLVM, Clang's Abstract Syntax Tree (AST) and the Intermediate Representation (IR).

2.3.1 Architecture and Compilation Pipeline

LLVM is designed like a traditional static compiler with a three phase design: frontend, optimization and backend [Lat11]. The frontend typically transforms the source code into a new representation which gets fed into the Optimizer, whose result is then again feed into the backend. The backend, also known as the code generator, then maps the output of the Optimizer to the target instruction set.

Figure 2.3 gives an overview of the architecture of LLVM. On the left we see some of the available LLVM frontends. Every frontend produces IR which is going to get consumed by LLVM's core. Within the LLVM core, the IR is analyzed and usually optimized by applying passes. The backends on the right side are responsible for producing the machine code, depending on the selected target machine's architecture.

The clean distinction between the three compiler phases into frontend, core, and backend, makes it possible to plug in additional front- and backends. This allows to extend LLVM conveniently by further input languages or target machine architectures, by simply adding another front or backend to LLVM. One of the most popular frontends is *Clang* (C-language family), which provides support for C, C++ and Objective-C as input language.

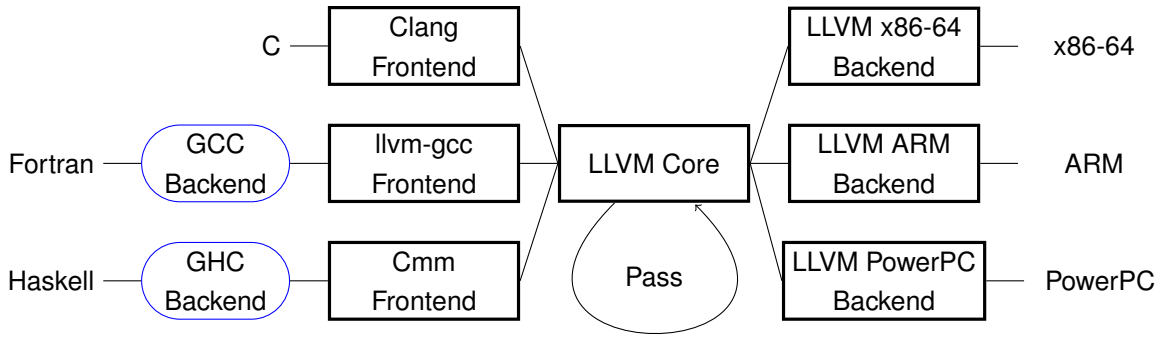

Figure 2.3 – Architecture of LLVM

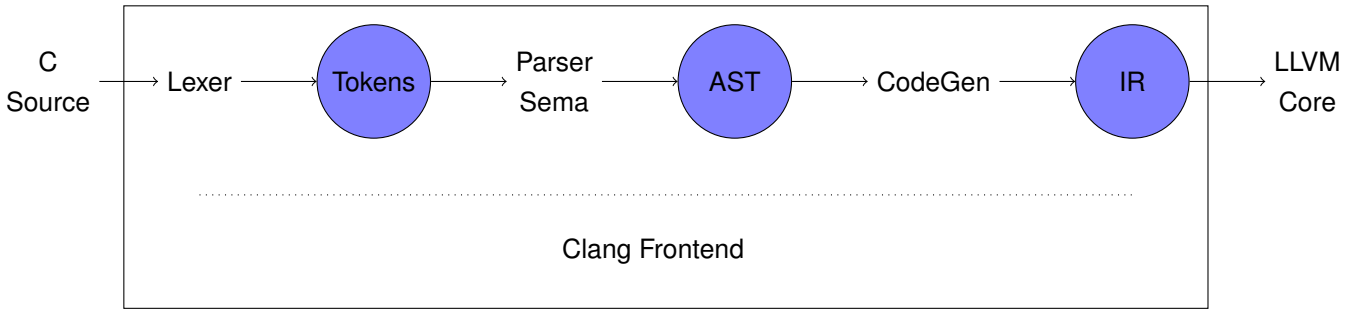

Figure 2.4 – Architecture of Clang

Most of the work described in this thesis is done within the Clang frontend, therefore we will discuss the internals and architecture of Clang in the following paragraphs. Figure 2.4 shows a detailed view of its processing pipeline.

The very first step, the *Lexical Analysis* is performed by the Lexer which splits the source code's textual input into tokens. The Lexer first removes irrelevant characters like comments or white spaces and decides if the remaining words and symbols are either part of the input language's reserved keywords or if they denote an identifier and then creates the corresponding tokens. It should be noted for completeness that Clang's C/C++ pre-processor works together with the lexer, and both interact with each other continuously. This enables detailed error and debug messages even when pre-processor macro expansion is involved.

After the Lexical Analysis is finished, the *Syntactic and Semantic Analysis* follows. A recursive-descent Parser transforms the tokens which were produced by the lexer into basic elements of the source language. While creating elements like operators, operands or statements, the Parser also performs a syntactic analysis. After such an element was detected by the Parser, the Sema library performs semantic analysis, which ensures that the element is valid within the current context and does not violate the language type system. Since the semantic analysis directly follows the parsing phase, Clang's Parser library directly invokes the Sema library [5].

LLVM consists of further projects besides core LLVM and Clang, which again provide multiple libraries. LLVM is thus not only a compiler, but also a framework providing libraries which can be used as building blocks for further projects. Those librarires are then usually glued together to form another tool like debuggers or static code analyzers.

The most notable LLVM projects besides LLVM core and Clang include:

Extra Clang Tools Tools built using Clang's tooling APIs. Those include: the *Clang C++ Modernizer*, used to automatically convert C++ code written against old standards to use features of newer C++ standards where appropriate, the *clang-tidy* linter tool to help diagnose and fix typical programming errors, the *modularize* tool for checking whether a set of headers provides the

[5] This was not always the case. The Parser library would talk to an abstract "Action" interface, but this was changed when C++ support was added to Clang.

consistent definitions required to use modules, and the *pp-trace* tool to trace preprocessor activity.

compiler-rt The compiler runtime, consisting amongst other things of 1. the *builtins* library providing low-level target-specific hooks; 2. various *sanitizer runtimes*, like the AddressSanitizer or MemorySanitizer; 3. the *profile* library used to collect coverage information.

libc++ The C++ Standard Library, targeting C++11.

LLDB A "next-generation high performance" debugger, which can be used as replacement for the GNU Project Debugger (GDB).

2.3.2 Clang's Abstract Syntax Tree (AST)

The heart of the Clang (C-Language) frontend of LLVM is the Abstract Syntax Tree (AST), to which every inputed source file gets transformed to after the lexer has tokenized the input. The AST is produced by Clang's Sema library, it is thus done right after the semantic analysis phase, which checks the token stream for semantic errors and, if there are none, creates an AST node for the current token(s). AST nodes represent declarations, statements, and types. Hence Clang organizes the AST Nodes in a class hierarchy with three distinct super classes: `Stmt`, `Decl`, and `Type`, as can be seen in Figure 2.5. Every AST node must inherit from exactly one of these super-classes.

An interesting design decision made in Clang was to model expressions as statements. While the initial intention was to keep the number of AST node superclasses low, more recently this is recognized as minor design flaw [Kli] because not every expression is also a statement.

Clang employs its own type system based on how types can be expressed in C/C++. The Clang type object instances are then transformed, together with the knowledge about the target architecture, to LLVM's IR types.

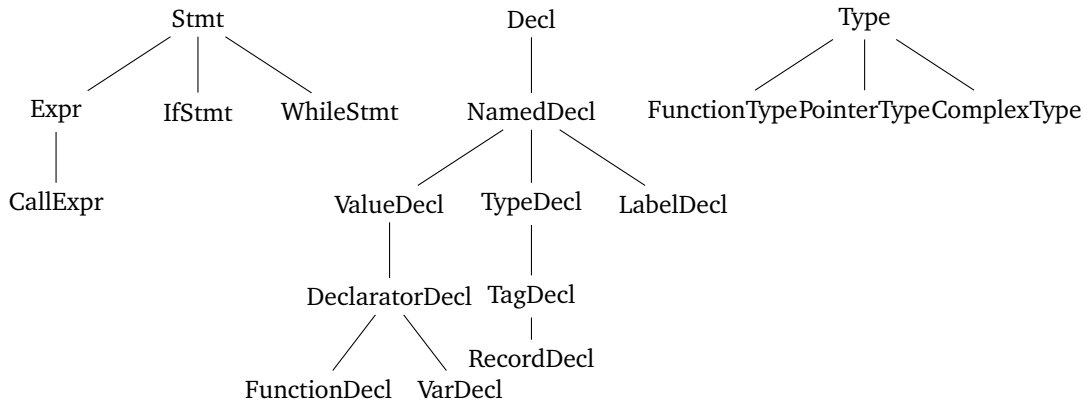

Figure 2.5 – Clang AST Node Hierarchy

2.3.3 LLVM's Intermediate Representation (IR)

The link between the frontends and backends is LLVM's IR used in LLVM core: frontends produce the IR, while backends consume it. The majority of target-independent optimizations are performed on IR within the LLVM core.

LLVM's IR can be characterized as somewhere between low-level programming languages like C, and architecture specific assembly languages. While IR can be used as target independent code, like it is done by Google's *Portable Native Client (PNaCl)* and discussed in [Kan11], it is usually used as non-architecture-agnostic code within LLVM (For example: Clang emits different intermediate code depending on the configured target.). IR may not be suitable as format for portable code because the LLVM project gives no gurantee that the IR will be stable or backwards compatible accross major versions, further arguments against are given in [Goh11].

Unlike C, IR is strongly typed with a simple type system. But just like C the calling convention is abstracted: LLVM IR uses the *call* keyword to represent a simple function call and *ret* to return the control flow from a function back to the caller. Optionally calls (and functions) can be annotated with a calling convention to be used. This is one example where IR may not be target independent.

LLVM IR shares many similarities with assembly. It is especially similar to RISC instruction sets: IR supports linear sequences of simple instructions like add, subtract, compare and branch, while all those instructions are in three address form obtaining their input and producing their output in different registers. Besides the typical RISC instructions a few specialized and rather complex instructions exists. Such as *getelementptr*, which is used to get the address of a subelement of an aggregate data structure. As IR is not designed for a particular hardware architecture, it is able to provide an infinite number of registers, which are prefixed with a % character in IR's textual representation.

A unique property of IR, compared to most programming languages and assembly, is that the code must be in Static Single Assignement (SSA) form, which requires that each register is assigned exactly once and every variable is defined before it is used. Requiring IR to be in SSA form is not suprising, since it simplifies and improves the results of a variety of compiler optimizations, such as register allocation.

Listing 2.5 shows the example IR code for the serial elision of Listing 2.1. A typical property of IR is that every identifier is accompanied by type information (e.g. `i64`). And since IR is strongly typed, every type conversion, for example from `i32` to `i64` needs to be performed explicity, as can be seen in line 6 in Listing 2.5. This is unlike C, which uses a weaker type system where some conversions are performed implicitly. Also every declared register (`%0`–`%7`) is assigned exactly once. The 'add' instruction is appropriate for both signed and unsigned integers, because LLVM integer types use a two's complement representation.

```
define i64 @fib(i32 %n) #0 {
  %1 = icmp ult i32 %n, 2
  br i1 %1, label %2, label %4

; <label>:2                                       ; preds = %0
  %3 = zext i32 %n to i64
  ret i64 %3

; <label>:4                                       ; preds = %0
  %5 = add i32 %n, -1
  %6 = tail call i64 @fib(i32 %5)
  %7 = add i32 %n, -2
  %8 = tail call i64 @fib(i32 %7)
  %9 = add i64 %8, %6
  ret i64 %9
}
```

Listing 2.5 – IR of the serial elision of Listing 2.1 compiled with `-O3`

2.4 OctoPOS

OctoPOS, a product family of parallel operating systems (POS), is a core component on which the work of this thesis is based. It was created and designed as an operating system for *Invasive Computing*, a new paradigm to address the hardware- and software challenges of future massively parallel systems like a Multi-Processor System on a Chip (MPSoC).

This section will provide the necessary information in order to understand how OctoPOS was used in this thesis. I start with an introduction to Invasive Computing in Section 2.4.1, continue with a detailed explanation of OctoPOS's design and goals in Section 2.4.2 and conclude with a discussion of the relevant parts of OctoPOS's API in Section 2.4.3.

2.4.1 Invasive Computing

The parallel operating system (POS) *OctoPOS* emerged from the Transregional Collaborative Research Centre "Invasive Computing". The Centre explores hard- and software of future massively parallel systems and focuses on the idea of resource-aware programming, in order to achieve high utilization and efficiency. The concept of Invasive Computing was described by Teich et al. [Tei+11]. The essential and novel idea of Invasive Computing is that a program should be able to discover and claim the available hardware resources, like processing units, memory and interconnect structures. Programs in invasive computing should be able to adopt their behavior to the ulitmatlively given

resources, as the operating system may not assign all requested resources. Hence they need to be programmed to be aware of their resources.

This is achieved through three basic primitives:

invade A request to associate resources with the issuing entity. If successful the run-time support system will start an *invasion* of the requested resources and assign them to a *claim* (which represents successfully invaded hardware resources).

infect Issued to use the granted resources, by infecting them like a virus with the program. After successful infection, the parallel execution may start on all infected resources.

retreat A request to retreat from the associated resources. If successful the run-time support system will free the claimed resources.

After the program has been successfully granted resources by invading them, it owns those resources *exclusively*, i.e. there is no preemption or other kind of displacement. The underlying idea is that the system consists of so many resources that one can assume that there are enough resources for everyone. Furthermore there is an superior instance which decides how the resources are distributed. As result, it is not required to provide a mechanism for resource-sharing.

The basic unit of execution, i.e. a piece of program subject to invasive-parallel execution, is referred to as an *i*-letwhich is short for “invasive-let”. An *i*-let represents a section of a program which is potentially going to be executed in parallel. It may not be executed in parallel because only one processing unit was invaded or because of scheduling decisions.

i-lets are usually grouped into *teams* which may carry a hint about the run-time behavior, such as the desired scheduling technique. After its creation, a team is used to infect a given *claim*.

2.4.2 OctoPOS Design and Goals

OctoPOS provides the required OS-level primitives and execution environment for invasive-parallel applications and was designed as Invasic Runtime Support System (IRTSS). In particular, OctoPOS is specifically designed for a Partitioned Global Address Space (PGAS) programming model like it is used with tiled many-core architectures [Moh+15]. The cores of such architectures see a global address space, but cache coherence is only guranteed for a group of cores which form a tile. Inter-tile communication is done over a simple scalable interconnection in a message-based fashion. Not relying on global cache coherency allows OctoPOS to efficiently support large systems.

Furthermore, compared to contemporary operating systems, the parallel execution model implemented by OctoPOS is very lightweight: the overhead to switch to another *i*-let after the last run *i*-let finished is comparable to an indirect function call [Oec+11].

This is achieved by designing *i*-lets as structure containing a void-function pointer and a pointer to some opaque piece of data. Upon execution of the *i*-let, OctoPOS will invoke the function pointer using the data pointer as its only argument.

The OctoPOS version used runs on computing cores accessing the same piece of memory with uniform access costs. OctoPOS supports neither spatial nor temporal preemption in order to support a slimmed down operation system [Oec+11]. As result an *i*-let in execution will run to completion, with the notable exception when it invokes a blocking system call. If such a system call occurs, OctoPOS will stop the currently running, but now blocking, *i*-let and remove it from the processing unit. Since the now blocked *i*-let's execution will be resumed in the future, the *i*-let's execution context needs to be preserved too. Thus OctoPOS has the concept of a *context*, akin to processes of traditional operating systems, which typically represents a stack on which *i*-lets are executed. Hence the current implementation of a context consists of a stack frame, a field for a saved stack pointer, in order to be able to resume the context later on, and a field indicating if the context is used or free. Now if an *i*-let is going to get blocked, the instruction pointer and the non-scratch registeres are pushed on the current context's stack, the stack pointer is aved and this context is removed from the processing unit. Then a new context is fetched from the pool of unused contexts and used to start executing, on the now unused processing unit, a new *i*-let. Once the condition to unblock the *i*-let is resolved, another *i*-let is scheduled. When executed, this new *i*-let will switch the current processing elements context to the one of the now no longer blocked *i*-let, after moving the current context into the pool of free contexts.

Three targets are supported by OctoPOS. Those are

SPARC Leon The initial architecture used for invasive computing. It features a hardware based Core *i*-let Controller (C*i*C), which is responsible for dispatching ready *i*-lets to the available cores of the claim.

x86 Guest Compiling OctoPOS for this target produces a x86 binary which can be run under Linux. If executed, OctoPOS runs as "guest" program under the Linux operating system.

x64 Native Targets standard x64 (x86-64, AMD64) hardware. On platforms with multiple Non-uniform Memory Access (NUMA) domains, OctoPOS creates a tile for every NUMA domain.

The fast and scalable execution model, together with the provided ligthweight primitives for executing *i*-lets and efficient synchronization mechansism make OctoPOS an ideal platform for large and highly parallel programs which exploit micro-parallelism.

2.4.3 System Calls

2.4.3.1 Invade and Retreat

The C API exported by OctoPOS provides functions required to apply the invasive programming paradigm. In order to invade cores on a tile `invade_simple(claim_t, uint32_t)` is used. Similar `retreat(claim_t, uint32_t)` will retreat from the given number of cores.

2.4.3.2 Infect

A claim can be infected with new *i*-lets by calling `infect(claim_t, simple_ilet[], uint32_t)`. But first the team of *i*-lets, given as second argument of `infect`, needs to be allocated and initilized. The memory for the team can be allocated on the stack, since OctoPOS will copy the *i*-lets. A single *i*-let is initialized by calling `simple_ilet_init(simple_ilet*, ilet_func, void*)`.

2.4.3.3 Simple Signal

Since OctoPOS strieves to provide the base for highly parallel programs, it also provides an efficient synchronization mechanism called *simple signal*. A signal shares many similiaraties with semaphores, but unlike those, only a single thread of execution can wait on a signal. As result, it is not allowed for multiple entities to simultaneously call `simple_signal_wait()`, which is the analogous to a semaphore's `P()` operation, on the same signal. This concept is also known as *private Semaphores*[Dij67].

This restriction allows for low overhead context switches when a thread needs to wait for a signal, a low memory footprint of the synchronization primitive and the signal and efficient synchronization. This is mostly owed due the `simple_signal` using a non-blocking atomic Compare-And-Swap (CAS) operation as basic synchronization primitive on which it is build upon. Which makes it the ideal construct to synchronize organize micro-parallelism.

A typical usage pattern of simple signals is as follows: After memory for the signal has been allocated, they are initialized with a counter using `simple_signal_init(simple_signal*, uint32_t)`. After initialization, the counter can be increased by calling `simple_signal_add_signalers(simple_signal*, uint32_t)`. If an *i*-let wants to signal an event, e.g. the completion of a particular task, it can decrease the counter of the signal by one using `simple_signal_signal(simple_signal*)`. Any other *i*-let previously waiting for the counter to decrease to zero by calling `simple_signal_wait(simple_signal*)` will become unblocked and re-considered for scheduling. An *i*-let calling `simple_signal_wait` on a signal which counter is still greater then zero will become blocked and its context will be saved. They blocked *i*-let will be suspended until the signal's counter reaches zero because another *i*-let called `simple_signal_signal`, which will re-schedule the blocked *i*-let and thus unblocking it.

2.5 Related Work

An approach similar to what would be using CilkO on OctoPOS over multiple distributed tiles is described in "Adaptive and Reliable Parallel Computing on Networks of Workstations" [BL97]. The thesis introduces "Cilk NOW (Network of Workstations)" and evaluates how Cilk can be as concurrency platform for distributed systems.

The Plan9 operating system traditionally used symetric rendezvous as primary synchronization primitve, but switched to asynchronous semaphores to wake up another process without blocking the own [MC08]. This approach is similar to CilkO's usage of `simple_signal` (Section 3.1).

Concurrent Cilk: Lazy Promotion from Tasks to Threads in C/C++ [Zak+15] presents an extension to the Intel Cilk Plus runtime, where task are lazily promoted to threads in case they end up calling a blocking system call.

3 ARCHITECTURE

The primary goals of this thesis can be described as mapping the Cilk language extensions to OctoPOS primitives, while moving as many tasks as possible performed by the Cilk runtime into the operation system, eventually replacing the runtime layer completely in the end. The outcome should be the capability to compile C source code including Cilk keywords to OctoPOS binaries which utilize the relevant OctoPOS syscalls to realize Cilk's features for parallelism. For this purpose we present **CilkO**, an enhanced LLVM/Clang compiler able to transform C code including Cilk's linguistic constructs into machine code tailored for OctoPOS. The following chapter describes how `cilk_spawn` and `cilk_sync` are transformed by the compiler and which changes where made to OctoPOS in order to schedule parallelism more efficiently. Section 3.2 describes how the C compiler was modified to generate the desired code for `cilk_spawn` and `cilk_sync`, before Section 3.4.1 provides an insight into the performed modifications to OctoPOS's scheduler in order to achieve efficency.

3.1 Mapping Cilk's Linguistic Constructs onto OctoPOS

As explained in Section 2.2 every `cilk_spawn` introduces two new strands of execution: One is the spawned function and one the continuation after the function call. Now the question arises how those parallel strands are created and executed. Since OctoPOS's unit of execution is an *i*-let, it is evident that at least one of those strands must be mapped to such and made available for concurrent execution. The other stand can be executed using the current thread of execution. Therefore at least one *i*-let must be created and infect the current claim for every `cilk_spawn`.

Given that the signature of an *i*-let function has the fixed type `void (*)(void*)`, it will not match the spawned functions signature in all cases. The CilkO compiler solves this by creating an *i*-let-helper function, which wraps the spawned function, and an accompanied *i*-let-helper context struct for every spawned function. The context struct holds the arguments of the spawned function (if any) and a pointer to the memory location where its return value should be stored (if any). The helper function is invoked with the context struct as argument, unwraps the arguments from it and invokes the spawned function with it, while also ensuring that the return value is stored at the right place.

After the *i*-let has been initialized with the helper function and context struct, the current claim is "infected" with it, which means that it will be executed by OctoPOS once an available core of the current claim is found.

What's left is how to join the created parallelism. OctoPOS already provides a mechanism to synchronize on concurrent events: The API around the `simple_signal`. Hence we use a `simple_signal` to synchronize parallel strands at the end of a `cilk_sync`. First the CilkO compiler needs to generate code which places exactly one `simple_signal` on the stack of every spawning function. Furthermore the compiler inserts code before every spawn, i.e. before every infect, which increments the signal's counter. The completion of spawned functions is signalled using this `simple_signal` by the helper function as its very last action. In order to make it avaialble to the helper function, a pointer to the `simple_signal` is placed in the context struct. Finally every implicit or explicit `cilk_sync` within the function is simply transformed to a `simple_signal_wait` system call on the current spawning functions `simple_signal`. After this call returns, which indicates that all previously created parallelism has finsihed, the signal's counter is reset to zero. This allows the signal to be re-used for potentially following `cilk_spawn` and `cilk_sync` operations.

3.2 Compiler support for CilkO

The previous section already explained that the C compiler needs to produce some additional code in order to enable the parallelism introduced by Cilk's keywords on OctoPOS. This section describes how the Cilk concepts are transformed to (machine) code using the OctoPOS API to enable parallelism.

3.2.1 Transforming Spawning Functions

Once the compiler identifies a function to be a spawning function, i.e. the function body contains at least once the `cilk_spawn` keyword, it generates the code to allocate and initialize a function local `simple_signal` and places the code at the beginning of the function. This `simple_signal` is used later to synchronize and join the parallelism created by Cilk as explained in Section 3.2.2 and Section 3.2.3.

3.2.2 Transforming `cilk_spawn`

A `cilk_spawn` annotated function call is essentially made concurrent by wrapping the invoked function into an *i*-let and infecting the current claim, i.e. the claim the spawning function runs on, with that *i*-let. Since every spawning function has at least one implicit `cilk_sync` before it returns, the function needs to keep track of its spawned functions and their completion. As explained in Section 3.2.1, the compiler generates a `simple_signal` for that purpose. Before a function is spawned, the counter of the `simple_signal` needs to be incremented by one, because there is now one more *i*-let representing a spawned function.

```
1 void bar() {
2   ...
3   int i = 10;
4   long l = 7;
5   char c = cilk_spawn foo(i, l);
6   ...
7 }
```

Listing 3.6 – Example spawning function `bar()`

Listing 3.6 shows the spawning function `bar()` which somehwere within its body spawns the function `foo(int, long)`. The compiler will annotate `bar()` as spawning function, and generate the code to declare and initialize a `simple_signal` at its beginning. It further generates code which declares the context struct (Listing 3.7) and the *i*-let-helper function (Listing 3.8) for `foo`.

```
1 struct __ilet_ctx_foo {
2   void* simple_signal_ptr;
3   int i;
4   long l;
5   char* result_ptr;
6 }
```

Listing 3.7 – *i*-let context struct for `foo()`

The helper struct always contains as its first field a pointer to the (invoking) spawning functions's `simple_signal`, followed by *n* fields representing the arguments of the function to be called. The last field of the struct will be a pointer to the memory location where the result should be stored. This field is only created by the compiler and thus existent if the spawned function's return type is non-void.

```
1 void __ilet_helper_foo(__ilet_ctx_foo* ctx) {
2   *(ctx->result_ptr) = foo(ctx->i, ctx->l);
3   simple_signal_signal(ctx->simple_signal_ptr);
4 }
```

Listing 3.8 – *i*-let helper function for `foo()`

The *i*-let-helper function for `foo()` shown in Listing 3.8 receives a pointer to a matching context struct and simply invokes the wrapped function with the arguments found in the context struct, and saves its result to the designated address which is also to be found in the context struct. As (second

and) last action, the completion of the spawned function is signaled by invoking `simple_signal_signal` on the signal found in the struct.

```
1  simple_signal_add_signaler(__cilko_simple_signal, 1);
2  __ilet_ctx_foo ctx;
3  ctx.simple_signal_ptr = &__cilko_simple_signal;
4  ctx.i = i;
5  ctx.l = l;
6  ctx.result_ptr = &c;
7  simple_ilet cilko_simple_ilet;
8  simple_ilet_init(&cilko_simple_ilet, &__ilet_helper_foo, &__ilet_ctx_foo);
9  claim_t claim = get_claim();
10 infect(claim, &cilko_simple_ilet, 1);
```

Listing 3.9 – Compiler generated code for `cilk_spawn foo()`

Listing 3.9 shows the resulting code the compiler transformed the `cilk_spawn` found in Listing 3.6 into. The context struct is declared and initialized in lines 1-5, which is then used together with the *i*-let helper function to initialize a single `simple_ilet` in line 8. Finally the *i*-let is used to infect the current claim.

3.2.3 Transforming `cilk_sync`

The required transformations for a `cilk_sync` are minimal compared to what was described in the previous sections for the `cilk_spawn`. It is directly transformed into a `simple_signal_wait()` call, using the current spawning function's signal as parameter, in order to ensure that all spawned functions are completed before continuing with the execution. And to be able to re-use the same `simple_signal`, the compiler generates a call to `simple_signal_init` after the `simple_signal_wait`, which initializes the same signal and resets its counter back to zero. This initialization is done uncondtitionally whether or not the `simple_signal` will be used afterwards, since its an inexpensive operation.

3.3 Joining Parallel Strands

Efficiently joining parallel strands is similarly crucial to the performance of a concurreny platform as is distributing the strands over the available cores (Section 3.4.1). In case of Cilk, there are two combinations where parallel strands are potentially joined: When the last spawned function signals its completion while another strand is already waiting for this event in a `cilk_sync`, or when a `cilk_sync` is encountered that is ready to proceed, i.e. all previously spawned functions have already completed.

In terms of transformed Cilk code using OctoPOS system calls, the joining can either happen at the call to `simple_signal_wait` or `simple_signal_signal`. What happens at the time of executing those system calls depends on the value of the signal's counter: The strands will only be joined if the counter's value is zero. The following paragraphs explain the different cases.

The `simple_signal_wait`, to which the `cilk_sync` statements are transformed, behaves as follows: If the signal's counter is already zero when the call is invoked, because all previously "spawned" *i*-lets have already completed, then it returns immediately and the current *i*-let simply continues executing. If the counter is positive, meaning that there are still unfinished *i*-lets, then the `simple_signal_wait` call blocks (from the perspective of the caller). OctoPOS will also move the context executing the `simple_signal_wait` into the signal and schedule a new context on the current core.

The counterpart of moving the context into the signal will eventually happen in the *i*-let-helper function. At its end, `simple_signal_signal` will be invoked, which again behaves differently depending on the counter state. If the counter's value is zero, it means that the blocked context which is "parked" in the signal is ready to continue. And in order to continue the context, the function schedules a special *i*-let which will resume the context once executed. Otherwise the function will simply return, and since it is the *i*-let's last statement, the control flow will reach the dispatch loop which schedules the next runnable *i*-let.

3.4 The Criminal OctoPOS: Work and Context Stealing

3.4.1 Work Stealing Scheduling

Usually the hardware-based C*i*C found on the SPARC Leon platform is responsible for scheduling and dispatching *i*-lets to the system's available cores. But since the hardware C*i*C only exists when OctoPOS is built for the SPARC target, all other targets come with a C*i*C emulator class, which implements the C*i*C's task in software. The software emulator performs a simple scheduling of newly infected and thus ready to run *i*-lets: New *i*-lets are distributed round-robin to the ready lists, implemented as FIFO queues, of the cores. This scheduling approach may cause underutilization of cores because a queue could become empty while other queues stil contain plenty of work. It was therefore decided to extend OctoPOS by a work stealing scheduler, which guarantees that all cores are utilized if there is work available.

At its core, any work stealing scheduler consists of a semi-concurrent deque implementation: the *work-stealing queue*, which is created for every worker. The worker is able to push new work and to pop existing work from the top of the queue, essentially using the deque in a stack like (FIFO) fashion. These two operations must not be executed concurrently, but this is never the case since there is only a single worker per queue. The only operation that can be performed concurrently with all other operations on the queue is popping elements from its bottom. This operation is performed by a worker trying to steal work from a different worker's queue. Section 4.2 describes and discusses in detail the different implementations of work-stealing queues which where added to OctoPOS.

And Section 3.5.1 explains the advantages of using FIFO semantics for the work-stealing queues compared to the LIFO queues the hardware C*i*C employs.

Algorithm 3.1 shows the *work stealing scheduling* loop every core (the "worker") executes once OctoPOS has been fully booted. First the core tries to fetch work from its own queue. If the queue is empty a random other core is selected to steal work from, eventually iterating over other cores until work in form of an *i*-let is found. If no *i*-let could be obtained, then the core sets itself to sleep.

It is of crucial importance that the scheduling alrogithm selects the core for work stealing uniformly at random. Blumofe and Leiserson showed in [BL99] that only by randomly selecting the victim to steal work from can the bounds of Equation (2.8) be achieved.

```
while true do
  i-let ← popTop()
  if ilet == nil then
    for all Cores of the same claim do
      i-let ← popBottom(core)
      if ilet != null then
        break
      end if
    end for
  end if
  if i-let != nil then
    execute(ilet)
  else
    sleep()
  end if
end while
```

Algorithm 3.1 – The scheduler loop

The enqueue function shown in Algorithm 3.2 is invoked every time the core infects its own claim with new *i*-lets. It first pushes the new *i*-let on top of its own queue and then tries to find a sleeping core of the same claim in order to wake it up. It is sufficient to wake up exactly one sleeping core, since an *i*-let an only be executed by one core. The function intentionally uses `isSleeping(core)` as first condition for the short-circuit evaluation `isSleeping(core) and sameClaim(core)`, because we assume that the common case is that all cores are powered up. As a result it is more likely that a core is not powered down compared to the core not belonging to the same claim.

The careful observer may already noticed that the work stealing algorithm suffers from the lost wakeup problem. As there is no synchronization providing atomicity, a core may decide to sleep (`sleep()` statement in Algorithm 3.1 while another is enqueuing new work. But since CilkO is meant for highly parallel programs utilizing micro-parallelism, even if the core decides to enter sleep mode, it is very likely to be woken up shortly after because a new *i*-let gets enqueued. Hence the

design of the scheduling algorithm accepts the existence of the lost wakeup problem, as it appears to be a favorable trade-off compared to introducing additional overhead imposed by introducing synchronization in order to solve the problem.

```
pushTop(i-let)
for all cores do
  if isSleeping(core) and sameClaim(core) then
    wakeUp(core)
    break
  end if
end for
```

Algorithm 3.2 – The `enqueue(`i`-let)` function

3.4.2 Context Stealing

As we will see in Section 3.5.1, in order to expose the continuation for stealing, OctoPOS needs to look for a free context every time a function is spawned.

Existing OctoPOS implementations use a CAS synchronized bitmap to keep track of the free contexts, which are marked as free by flipping their respective bit to one in the bitmap. A lookup of a free context therefore means linearly processing the bitmap until a word is found which is not zero (as otherwhise all contexts "managed" by the word are in use). If such a word is found, the lookup method will set the first non-zero bit of the word to zero and store the new value of the word using a CAS operation. As is typical for CAS usage, the whole process is repeated if the operation fails.

When a context is put back into the pool of free contexts, then the bit representing the context is first set to one and then the according word is saved using CAS.

Hence the bitmap is a single datastructure modified using CAS operations by all cores. And since every `cilk_spawn` typically involves two modifications to the bitmap, one when retrieving a free context and the other when putting the free context back, it becomes a performance bottleneck caused by contention around the words of the map. This contention also increases the communication work required by the cache coherence protocol.

After this situation was realized, the existing bitmap was replaced by the introduced work-stealing queues. Similar to Section 3.4.1, a work-stealing queue for every core is created which holds free contexts. Those queues are then used by cores to steal contexts, as apposed to work, when required: When in need of a free context a core first tries to pop a context from its own queue. If the core's own queue is empty it resorts to *context stealing*, i.e. trying to retrieve a free context from other cores queues. If a core does not require a context any longer, i.e. the context became unused, putting the context back to the pool of free contexts is a inexpensive synchronization free push operation on the queue.

Contrasting to the previous bitmap based context management, retrieving a context from the core's queue is in $\mathcal{O}(1)$ and does not require synchronization in terms of CAS, which is an improvement over linearly scanning the bitmap for a free entry, which is in $\mathcal{O}(n)$, and using CAS. The context stealing algorithm will traverse over the queues of the other cores, only if the core's queue does not hold any contexts

Putting a context back using context stealing has the same bound as the previously used bitmap based algorithm: $\mathcal{O}(1)$. But it does not require a CAS operation, which avoids any overhead imposed by the cache coherence protocol.

Table 3.1 summarizes the differences between the bitmap based and continuation stealing based approach for conext management.

3.5 Optimizations

3.5.1 Accomplishing Continuation Stealing

The transformation deployed by the compiler so far results in the child being offered to thieves, i.e. employs child stealing, which comes with the drawback mentioned in Section 2.2.5. It is therefore desireable to implement continuation stealing.

The basic concept to deployed by continuation stealing can be described as follows: Instead of spawning child functions until a `cilk_sync` is reached, the continuation is first pushed onto the work-stealing queue, then the spawned function, and then the current thread yields its execution. At the yield the queue will have the spawned function as top most item, with the continuation below, which essentially exposes the continuation to (stealing) thieves.

Thus the OctoPOS API was extended to allow for continuation stealing by introducing two new system calls:

`cilk_create_continuation` Prepares a new continuation, which begins right after the next call to `cilk_yield`, and infects the current claim with the continuation, effectivly placing the continuation on top of the current worker's queue.

Context Management	**Retrieve**		**Put Back**	
	Complexity	Wait-free	Complexity	Wait-free
Bitmap based	$\mathcal{O}(n)$	no	$\mathcal{O}(1)$	no
Context stealing	own queue: $\mathcal{O}(1)$ otherwise: $\mathcal{O}(n)$	if from own queue	$\mathcal{O}(1)$	always

Table 3.1 – Context Management: Bitmap based versus context stealing

`cilk_yield` Causes the execution of the current context to yield until the continuation, which was created with a preceeding `cilk_create_continuation` call is executed. Yielding a context results in a new context being executed on the current core.

Those system calls are invoked around the infection of the current claim with the spawned function's *i*-let: `cilk_create_continuation` is placed before the `infect()` invokation, and `cilk_yield` is placed after it. Listing 3.10 shows the relevant parts of Listing 3.9 extended with those calls (Lines 3 and 5).

```
1 ...
2 claim_t claim = get_claim();
3 cilk_create_continuation();
4 infect(claim, &cilko_simple_ilet, 1);
5 cilk_yield();
```

Listing 3.10 – Compiler generated code for `cilk_spawn foo()`

To illustrate how the addition of those two calls leads to continuation stealing we observe the contents of the worker's queue during each step of Listing 3.10, assuming an empty queue at the beginning. First the continuation is pushed on to the queue because of `cilk_create_continuation` in line 3. Then the *i*-let of the spawned function is pushed, which makes it the current item on the bottom end of the queue, on which the local worker operates. "Underneath" this item lies the *i*-let representing the continuation. Assuming that the queue was previously empty, it is also the last item at the top of the queue, from which thieves steal work, thus allowing thieves to steal the continuation. Next the `cilk_yield` in line 5 is executed. This causes the actual context to yield and the current core to switch to a new context, which will begin executing tasks from the bottom of the worker's queue. Depending on how much work has been stolen from the queue, it will either execute the *i*-let of the spawned function or find an empty queue (meaning the worker will become a thief).

It should be noted that the continuation, which is realized using a special kind of *i*-let called *wake up* i-*let*, will be busily waiting until `cilk_yield` has been called. This is because the wake up *i*-let will only resume the context after it has been marked as unused, which is what `cilk_yield` does, besides using a new context on the core to continue the schedule loop. Thus a thief trying to execute the continuation may end up in a busy loop for a brief period of time, i.e. the time span between `cilk_create_continuation` and the corresponding `cilk_yield` to be precise.

This approach of implementing continuation stealing is entirely different from how it is done in traditional Cilk implementations. Using OctoPOS's existing context handling infrastructure removes the burden from implementing (and manageing) a cactus stack (Section 2.2.4). Instead CilkO uses a "new" stack, represented by a different context, for every spawned function. This may appear to introduce unneccsary overhead caused by copying the spawned function's arguments into the

new stack in case no work is stolen. But the inpact of the overhead is likely comparable to the one required to manage the cactus stack.

3.5.2 Improving `simple_signal_signal`

The `simple_signal_signal` at the end of the *i*-let-helper function provides possibilities for optimizations. This is because it is the very last statement the helper function executes, and thus it is ok if the method never returns (from the perspective of the helper function).

This observation allows for optimizations in all cases the signal may end up in.

If the counter's value is zero, it means that the blocked context which is "parked" in the signal is ready to continue. Thus we can simply unwrap the context and continue the unwrapped context on the current core right away. Of course we first have to put the current context into the pool of free contexts, before continuing the unwrapped one. And we also need to ensure that the unwrapped context belongs to the same claim, otherwise we simply schedule the context on the correct claim.

If the counter is non-zero the current context can be restarted, continuing in the dispatch loop of the scheduler. It is possible to continue with the dispatch loop right away since we know that this is what would happen anyway if the `simple_signal_signal` returned, as it is the last statement of an *i*-let function. Although not as many as the "counter has reached zero" optimization, this does also save a few CPU cycles (namely the `ret` instructions which would otherwise lead to the dispatch loop).

Table 3.2 summarizes how parallelism is joined as described in Section 3.3 and in this section.

`simple_signal_wait`		`simple_signal_signal_and_exit`	
Counter Value			
$= 0$	> 0	$= 0$	> 0
Returns immediately	Wraps current execution context into the simple signal. Then fetches a new context, starts it and dispatches *i*-lets from the "ready" queue.	Unwraps execution context from signal and resumes it on the current core	Restarts the context, which continues in the dispatch loop, working on the queues *i*-lets
Does join	Does not join	Does join	Does not join

Table 3.2 – Join cases

3.6 Architectural differences between Cilk Plus and CilkO

This section compares the architecture of CilkO with Intel's Cilk Plus implementation, which is the latest commercially available Cilk-like concurrency platform.

The most noteworthy difference from Cilk Plus is the complete removal of the runtime library layer. Work scheduling and management is done directly within the operating system, and all auxiliary code enabling the parallelism is generated by the compiler.

Since CilkO does not have a runtime layer between the parallel program and the operating system, and because OctoPOS executes *i*-lets in a run-to-completion manner, a Cilk worker is a physcial core of the system running OctoPOS, and not a designed as thread.

Traditional Cilk platforms (MIT Cilk, Cilk Plus) implement continuation stealing using non-local jumps, i.e. `setjmp()` and `longjmp()`, creating a cactus stack leaf if the continuation is spawned. They also determine if the continuation has been stolen by exploting the LIFO semantics of the work-stealing queue: the top item is pushed from the queue after executing the spawned function. If it is the continuation pushed just before executing the spawned function, the runtime continues a "fast clone", continuing the execution just the serial elision were executed. Otherwise, when a thief has stolen the continuation, a "slow clone" of the continuation is executed.

CilkO does not (yet) distinguish these cases, that is, it does not determine if the continuation has been stolen. It always performs the context switch. This appears to be a key enabler for efficient scheduling of Cilk programs, as we find in Section 5.3.4. On the other hand the `simple_signal_signal_and_exit` of the spawned function's helper will pop the continuation and resume it in an efficient manner (but unlike the "fast clone" case, still requiring a stack frame switch).

The worker threads used by Cilk Plus busily poll for work, executing an occasional `pthread_yield()` if there has been no new work in a while. This results in the worker threads fully utilizing the processor, even if there is no work. CilkO with OctoPOS by comparison will put processors to sleep if there is no work. Sleeping processors will be woken up by an interrupt as soon as new work becomes available.

Also Cilk Plus worker threads executing a (long) blocking system call will be unable to process further work as long as the call is blocking their execution. Using blocking system calls in parallel code could thus lead to deadlocks. Thus most concurrency platforms view the usage of blocking system calls as a violation of the contract between the programmer and the platform. OctoPOS on the other hand is fully aware of blocking system calls and will simply schedule another *i*-let in this case.

Table 3.3 summarizes the differences. Please note that "Fiber" is used as term in the Cilk Plus to describe a specific datastructure in the runtime. Usually the term describes something more lightweight then a thread.

Intel's Cilk Plus	**CilkO**
Strand encapsulation	
`__cilkrts_stack_frame`	*i*-let *i*-let-helper function *i*-let-helper context struct
Spawn closure	
Spawn helper	*i*-let-helper function
Coroutine state holder	
Fiber	Context
Work managed by	
Runtime library	Operating system
Continuation exposed for stealing via	
non-local jump's `jmp_buf`	Wake Up *i*-let
Sync approach	
Mutex implemented as spinlock	`simple_signal` using CAS
Abstract worker type	
Thread	Core
Worker state	
`__cilkrts_worker`	Context executing on a core and the queue of the core

Table 3.3 – Comparison of Cilk Plus and CilkO

3.7 Summary

This chapter showed how Cilk's linguistic extensions are transformed to code utilizing the invasive API provided by OctoPOS, and how OctoPOS was improved to process parallelism, in form of *i*-lets, more efficiently.

The next chapter discusses the implementation in more detail.

4 IMPLEMENTATION

This chapter describes the implementation in detail. It begins with explaining the modifications applied to the LLVM suite, in order to obtain a "CilkO-enabled" LLVM compiler suite, and finishes with the changes made in OctoPOS.

4.1 Adding Support for CilkO to LLVM

Since OctoPOS's execution model is similar to the execution model provided by the Cilk runtime, it is possible to completely replace the usually required Cilk runtime with syscalls provided by OctoPOS. A Cilk aware compiler targeting OctoPOS would thus produce extra code which invokes syscalls instead of calls into a runtime (like it is usually done by Cilk aware compilers). The following section explains in detail how LLVM was extended and modified in order to transform C source code including Cilk language extensions into specialized machine code for OctoPOS. The modifications where based on the Cilk Plus/LLVM[6] project by Intel.

4.1.1 The Clang Driver

When someone compiles a program with Clang he or she is actually using the Clang driver, which provides access to the Clang compiler and tools over a command line interface. Its job is to orchestrate the Clang compiler and the various tools to produce the desired result.

For the CilkO purpose, the Clang driver was extended by the following two options.

`-fcilkplus` Enables support for the `cilk_spawn` and `cilk_sync` keywords. The reason that it says "cilkplus" instead of "cilko" stems from the fact that the LLVM modification for CilkO is based on the existing modifications of LLVM by Cilk Plus, and should probably be changed to "cilko" in the future. Disabled by default.

`-fno_octo_cilk_support` Enables the generation of code using OctoPOS's Cilk support API to allow continuation stealing. Enabled by default.

[6] `https://cilkplus.github.io/`

These driver flags translate to language options defined in a TableGen defintion and are available to every Clang component through the `LangOpts` object.

4.1.2 The Cilk Headers

In order to easily produce the serial elision of a Cilk program, the actual Cilk keywords of the language extension are aliased using preprocessor marcros. For example

```
#define cilk_spawn _Cilk_spawn
```

is used to assign the alias `cilk_spawn` to `_Cilk_spawn`. The definition of those aliases is guarded by an `#ifndef cilk_spawn` statement. As result the serial elision of a Cilk program can be produced by defining `cilk_spawn` and `cilk_sync` to the empty string before the inclusion of the Cilk header as shown in Listing 4.11.

```
1 #define cilk_spawn ""
2 #define cilk_sync ""
3 #include <cilk/cilk.h>
```

Listing 4.11 – Preprocessor `define` statements to produce the *serial ellision*

Since these headers must be available on the host building a CilkO program, they should be distributed by our CilkO-enabled LLVM. Hence, the *compiler-rt* LLVM project was modified so that it installs the Cilk header into the configured `include/` path used by the LLVM distribution.

4.1.3 Modifications to Clang's AST

CilkO introduces three new AST nodes into Clang's AST:

CilkSpawnExpr An expression annotated with a `cilk_spawn`, e.g. `cilk_spawn foo(42);`

CilkSpawnDecl A declaration which is initialized by the result of a `cilk_spawn` expression, e.g.
`int i = cilk_spawn bar(42);`

CilkSyncStmt A statement representing a `cilk_sync`.

The `CilkSpawnDecl` is designed as simple mixer class, i.e. a class wrapping multiple other classes: `CilkSpawnDecl` consists of a `DeclContext` and a `CapturedStmt`. `CilkSpawnExpr`, on the other hand, does nothing more than simply wrapping a `CilkSpawnDecl`. A not so powerful class wrapping a more poerful one may not appear like the typical OOP approach, but this comes in handy in the CodeGen step when creating IR out of the AST (as will be explained in Section 4.1.6).

Introducing new AST node classes to Clang also requires changes to existing utility classes which must be aware of every AST node type. For example the `DataRecursiveASTVisitor`, commonly used to recursively visit AST nodes, needs to become aware of the new Cilk nodes.

Besides introducing new AST nodes, it was also necessary to modify existing ones. The `FunctionDecl` class was extended by an `IsSpawning` boolean, which when set to true, indicates that this FunctionDecl's body contains at least one `cilk_spawn` keyword, i.e. this function is a spawning function.

As explained above, a `CilkSpawnDecl` consists of a declaration context and a captured statement. The latter is expressed as `CapturedStmt` AST node. This node caputures a statement into a function, by acting as holder for a `Stmt` instruction, which can be multiple statements in case `CompoundStmt` is used, and the captured variables, including the expression used to initialize such, used in the caputured statements. Although CilkO uses only a subset of the features `CapturedStmt` provides, e.g. there will only ever be one statement captured and this will always be a function call (i.e. `CallExpr`), it is nevertheless the perfect match to wrap our `CilkSpawnDecl` around, because of the LLVM infrastructure already provided around it.

For convienience `Decl` was also extended by a simple `isSpawning()` method, in order to easily determine if a `Decl` instance is of type `FunctionDecl` which is spawning.

4.1.4 Clang's Lexer

Clang's Lexer is responsible for transforming the fed source code into a stream of tokens. Its core datastructure is a TableGen database containting the information about every known token's structure and its kind, like 'keyword', 'identifier' or 'punctuator'.[7] TableGen databases are used in in various places in LLVM. The database usually holds domain specific information which is then transformed by a TableGen backend into a concrete representation (usually source code which can be included).

```
1 KEYWORD(_Cilk_spawn , KEYCILKPLUS)
2 KEYWORD(_Cilk_sync  , KEYCILKPLUS)
```

Listing 4.12 – TokenKinds.def TableGen entries for the Cilk keywords

This database was extended to include the information about the added Cilk keywords `_Cilk_spawn` and `_Cilk_sync` (Listing 4.12. Those tokens will then be consumed by the parser.

4.1.5 Clang's Parser

The added language keywords `cilk_spawn` and `cilk_sync` are valid to appear in three distinct locations within a C program: `cilk_spawn` may appear prefixing an expression spawning a function, or within a declaration where a variable is declared and defined using the result of a spawned function. `cilk_sync` may appear everywhere a statement would be valid. Thus Clang's parsing library was extended to expect them at those locations.

[7] For all token kinds see [JTC11, Section 6.4].

The existence of the `cilk_spawn` keyword before a spawned function is checked within the parser function used to parse cast expressions, since they are both found at the same place (with the exception that `cilk_spawn` must be placed before a function call): Right before a postfix expression[8]. For `CilkSpawnDecl` AST nodes, the parser also needs to determine now whether or not a variable declaration is initialized using a full expression prefixed with `cilk_spawn`. Finally parsing support for `cilk_sync` simply means adding a case for `tok::kw__Cilk_sync` in the switch-case statement used to parse statements.

Essentially all the parser does is to call Clang's semantic analysis for the current source code location once a particular keyword has been parsed.

4.1.5.1 Semantic Analysis

Semantic analysis with regard to CilkO means checking for certain illegal constructs. For example, it is not possible to spawn certain function types (like CUDA kernel calls).

Besides performing context sensitive validation of the code, it also marks existing function declaration AST nodes as spawning if they contain a `cilk_spawn` call. We will later use this information when generating IR as explained in Section 4.1.6.

After the parsed code has successfully passed the semantic analysis, the according AST nodes will be created and inserted into the AST.

4.1.6 Generating IR Out of the AST: Clang's CodeGen Step

The next step after creating an AST representing the input source is to transform this AST into IR for further processing by LLVM. This is done in Clang's CodeGen step using the equally named Clang library.

In the light of CilkO, this CodeGen step means generating the calls to the OctoPOS API and declaring variables at the right place as described in 3.1. Special care has to be taken when declaring variables and automatically allocating them on the stack as this requires to know the size of the type to allocate. CilkO retrieves the correct size, of e.g. `simple_signal`, by traversing the AST of the current compilation unit for the declaration of the particular type and using the information to calcuate the size. The declaration is part of the compilation unit, because every OctoPOS program should have the OctoPOS header included. If this is not the case, CilkO emits a compiler error.

The same happens for the required OctoPOS API functions: the AST node corresponding to their function declaration is retrieved from the AST and used to generate the IR invoking those functions.

What is left is emitting the correct IR code for `cilk_spawn` and `cilk_sync`. As already explained in 3.1 the according `simple_signal` function calls need to be emited for a `cilk_sync`. In case of `cilk_spawn` more IR code is required to create the *i*-let, the *i*-let-helper function and glue code around it. Here it comes in handy that the `CilkSpawnExpr` AST node simply wrapps the `CilkSpawnDecl`, as the function emitting the IR code for those two can be re-used by simply

[8] [JTC11, Section 6.5.2]

unwrapping the node and leaving out a few parts in case not variable is initialized with the `cilk_spawn`.

4.2 Work-Stealing Queues

After the adjustments to the LLVM compiler have been discussed, this section continues with the applied improvements to OctoPOS, namely the added work-stealing queues.

Work stealing queues are double-ended queues (deques), but provide asymmetric operations on their ends: It is only possible to append on one end. Traditionally the end where it is possible to append and remove is called the bottom. This 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 and poped from the *bottom*. Thus the operations the worker uses are called `pushBottom`, to add a new item to the queue, and `popBottom` to remove the "bottom-most" item, i.e. the one added with the last `pushBottom` invocation, from the queue. Work stealing attempts are solely performed on the other end of the queue, which is called the *top*. Thieves use the `popTop` operation to steal work from queues of other workers.

All work-stealing queues share the property that they are *partially* concurrent. That is, only `popTop` can be used concurrently with any of the other three operations of the queue. It is not permited to use any of the bottom operations concurrently. But since those are only used by a single worker on the same queue, this is not really a limitation, instead it is a helpful constraint which allows for efficient partially concurrent implementations of work-stealing queues.

OctoPOS was extended with two additional abstract data types that fulfill all requirements which where mentioned above, to act as work-stealing queues. Both queues implement `popBottom` and `popTop` using a lock-free non-blocking algorithm leveraging an atomic compare-and-swap (CAS) instruction (alternatively load-linked/store-conditional could be used on architectures not supporting CAS). The `pushBottom` operation is performed in a simple, but efficient, wait-free manner. Also both queues have two fields to keep track of the current head and bottom element. The main difference between them is *how* those fields are used to identify the head and bottom element, as we will see in 4.2.1 and 4.2.2.

While not a direct requirement of work-stealing queues, both queue implementations are fixed-size array to hold the tasks[9]. This makes the queues bounded; the `pushButtom` operation will signal the fact that the queue is fully occupied by returning 'false'. Fortunately this did not require any modifcations of OctoPOS, since the existing implementation already behaves the same way and their is an optional overflow handling.

The following two sections explain the design and implementation of the two work-stealing queues added to OctoPOS. It will also discuss their respective advantages and disadvantages.

[9] See [CL05] for a work-stealing deque algorithm using a dynamic array.

4.2.1 ABP Queue

The first work-stealing queue which was added to OctoPOS is based on the algorithm presented by Arora, Blumofe, and Plaxton in [ABP98] and henceforth referred to as ABP queue.

The ABP queue's top and bottom fields directly hold an integer index which directly references work tasks in the underlying array, i.e. the integer acts as array subscript. The top field points to the oldest work task in the queue and the bottom field always points to the next free entry in the array. If a worker calls `popBottom` to retrieve a task, it is first checked if the bottom field is non-zero. If it is zero, then the queue is empty and the method returns 'false'. Otherwise the bottom field will be decremented and the result is used to address the task witin the array. Since the bottom field is never modified concurrently, i.e. it is only touched by the same worker, no synchronization is necessary.

The same is true for the `pushBottom` method, which also only modifies the bottom field. Its current value is used to determine the free slot in the array, where the given task is placed. Then bottom is incremented.

The `popTop` method also only ever modifies the top field, but this operation is potentially invoked concurrently by multiple workers which are acting as thieves. First the method checks if `bottom > top`, to determine if the queue holds any elements, returning 'false' if it is empty. Next the value of the top field is used to locate the task to steal from the array, copying the task into the result field. Then CAS is used on the top value, with the old value of the field and the old value incremented by one, to synchronize multiple callers trying to steal from the queue. If this CAS operation fails, then the caller lost a race against another caller and retries the steal attempt.

The main disadvantage of the ABP queue is caused because its top value is increased every time `popTop` is successfully invoked. The distance between the top and bottom value thus decreases by one with every steal successful attempt. But since the ABP queue is only able to use the free array slots from the top value to the end of the array, the number of free slots is efficively decreased by one by every steal. As result it is possible that the queue overflows although there is plenty space left in the array.

The ABP queues tries to mitigate this problem by resetting top and bottom to zero if `popBottom` finds both fields to yield the same value, which means that the queue got empty after `popBottom` decremented bottom. But it is conceivable that this mitigation never takes place and thus the number of effictively free slots is only a fraction of the real array size. This could happen to queues which were frequently stoolen from, and thus the top field points to a slot near the bottom end of the queue, but the queue never got completely empty because the worker produces work in the same speed as it consumes it. If the worker now enters a phase where it produces additional work, it may end up in a situation where the queue is full, although the majority of the fields in the backing array are empty, leading to OctoPOS's overflow strategy to get employed, which may cause an aching performance penalty.

4.2.2 CL Queue

Chase and Lev present in [CL05] an improved version of the ABP queue which does have the overflow problem described above. Their algorithm is based on a circular-array which is indexed using a modulo operation: The queue elements stored in the circular array are indexed modulo the array size. Using this modulo operation renders the ABP queue's reset-on-empty heuristic unnecessary. Since the only operation that modifies `top` is `popTop`, and unlike the ABP queue, `top` is never decremented. This yields the risk of `top` overflowing, which would cause the algorithms of the queue to determine the current number of stored items and to find the top item to produce invalid results. Potential issues caused this are bypassed by using an unsigned 64-bit integer, which is large enough to accomodate 64 years of queue operations at a rate of 4 billion operations per second. This requires the CL queue's to use the double-wide CAS instruction, instead of a single-word CAS instruction.

The different approach of indexing queue items requires a new approach to determine if the queue is full. Compared to the ABP queue, which can simply compare `bottom` with the array size, the CL queue needs to compute the number of items it is currently holding by subtracting `top` from `bottom` and then compare the value with the maximum possible size, i.e. the size of the backing array. This method is slightly more expensive then the one of the ABP queue.

Table 4.1 summarizes the differences of the two queue implementations.

4.3 Future Improvements

The current implementation still has room for improvements, which will be discussed in this section.

The static knowledge by the compiler could be used to aggregate multiple `simple_signal_add_signaler` invocations into a single invocation. This would also save an unnecessary `simple_signal_init` unconditionally created after a `cilk_sync` in case the `simple_signal` is never used afterwards. While this may appear as trivial task at first, this optimization must take care if the control flow contains `cilk_spawn` statements in conditional blocks.

The invocation of the spawned function within the *i*-let-helper function could be uncondtionally inlined (if its definition is available at compile time). The LLVM infrastructure already provides a suiteable method to do so on IR level. This requires the IR of both functions, the one with the call site and the one to inline, to be available at the time the inlining is attempted, and thus needs to be done as LLVM core pass.

Queue	Synchronization	Index	`bottom`/`head` types	`isFull()`
ABP	`cas`	direct	`uintptr_t`	`bottom == SIZE`
CL	`dwcas`	modulo	`uint64_t`	`bottom - top == SIZE`

Table 4.1 – Comparison between ABP and CL queue

5 ANALYSIS

An obvious candiate to compare CilkO on OctoPOS with, in order to evaluate its performance, is Cilk Plus on Linux. Section 5.3 does provide this comparison, together with some insights of current bottlenecks and their impact on the general performance.

Another important metric is the speedup CilkO is able to provide. The amount of speedup a concurreny platform is able to provide is especially relevant when a high number of cores is utilized, as there is usually a tail off in speedup when the number of cores increases. Section 5.4 provides the speedup numbers with different sizes of parallelism. It compares micro- to macro-parallelism speedup when executed on one up to 20 cores.

A comparison of the work-staling queues and different stealing approaches is Concliding this chapter. The comparison focus on comparing the implementations in the light of their effect on performance and stability in Section 5.5 and Section 5.6.

5.1 Analysis Setup

The following evaluations were performed on two hardware platforms. The first we use is a typical workstation system and is dubbed “Fastbox”. The other one is a multi-core server system (HP ProLiant DL560 Gen8), which we will refer to as “Bigbox” from now on. The hardware details of those two are found in Table 5.1.

Besides the `fib()` benchmark used in Section 5.3, the main benchmark program used to evaluate CilkO consists at its heart of a recursive function as shown in Listing 5.13.

This benchmark has the following three configurable parameters:

MAX_REC_DEPTH The maximum recursion depth.

BREADTH The number of `cilk_spawn`s spawning `recFun(depth + 1)` recursively in every pass.

WORKLOAD The number of iterations the work loop will perform.

This is a synthetical bechmark which reassembles a typical parallel recursive function, similar to the `fib()` function in Listing 2.1. The parameters act as knobs allowing fine tuning and adjusting of

	Fastbox	Bigbox
Processor	Intel Xeon E3-1275	Intel Xeon E5-4640 v2
Processor Speed	3.50GHz	2.20GHz
Physcial CPUs	1	4
Cores per CPU	4	10
Logical Cores per CPU (threads)	8	20
Total Logical Cores	8	80
Memory	32 GiB	128 GiB

Table 5.1 – Hardware

```
1  static void recFun(uint32_t depth) {
2    if (depth > MAX_REC_DEPTH) return;
3    volatile uint64_t tmp;
4    volatile uint64_t workres[256];
5    for (uint64_t i = 0; i < WORKLOAD; ++i) {
6      tmp *= i * 2;
7      tmp /= (i + 3) * 4;
8      workres[i % 256] = tmp;
9    }
10   for (unsigned int i = 0; i < BREADTH; ++i) {
11     cilk_spawn recFun(depth + 1);
12   }
13 }
```

Listing 5.13 – The recursive function of the master benchmark.

the benchmark behavior. `MAX_REC_DEPTH` and `BREADTH` allow to specifiy the number of executued `cilk_spawn` function calls of `recFun()`. This number can be calculated using Equation (5.1).

$$\text{SpawnCount} = \sum_{i=0}^{\text{MAX\_REC\_DEPTH}} \text{BREADTH}^i - 1 \tag{5.1}$$

Every execution of `recFun()` can be viewed as a new work package, which executes work depening on the value of `WORKLOAD`. Thus increasing `WORKLOAD` incrases the work size every work package has to process.

In the following `bench(MAX_REC_DEPTH, BREADTH, WORKLOAD)` will be used to describe the used benchmark parameters.

5.2 Code Length

Using Cilk's linguistic extensions increases the expressivnes and readiblity of a parallel program, compared to using OctoPOS's native API to achive the parallelism. Listing 5.14 shows an example program computing the 11th Fibonacci number which does not use Cilk. Listing 5.15 shows the same program but now using the Cilk keywords to achieve paralleism.

The number of source lines of code (SLOC) measured by `sloccount 2.26` was 45 for Listing 5.14 and 15 for Listing 5.15. Thus using Cilk keywords increases the SLOC down to 33% of the original size.

5.3 Performance Analysis using Micro-Parallelism

The following evaluations use the parallel `fib()` function from Listing 2.1 in order to evaluate the overhead introduced by the concurrency platform as closely as possible. The `fib()` function is a highly parallel computation where the actual work which gets parallelized is minimal, thus differences of the execution duration (using the same input) can be credited almost exclusively to the concurrency platform and its performance.

For this analysis, the `fib()` function was invoked using 30 as argument which should yield 832040 as result, i.e. the 30th Fibonacci number. Every initial `fib()` invokation, in order to measure its execution time, was run 20 times in order to compensate jitter while performing the measurement. The clock values used as start and stop times where retrieved out of the High Precision Event Timer (HPET) found in modern x86-64 hardware.

5.3.1 An Early Performance Comparison

The first evaluation of CilkO performed in the early stages of development yielded surprising but disappointing results. CilkO on OctoPOS running on all 8 cores of the Fastbox turned out to be an order of magnitude slower compared to executing the same program on the same hardware but now using Linux with the Intel Cilk Plus runtime. The results are shown in Figure 5.1: the average time it takes to compute `fib(30)` using CilkO is *262.9ms*, whereas it is *15ms* using Cilk Plus on Linux. CilkO/OctoPOS was 17.5 times slower than Cilk Plus on Linux. This is arguably not a good start, but very motivational to analyse and mitigate the reasons.

5.3.2 Context Stealing

One bottleneck found as result of reviewing the OctoPOS implementation because of the early evaluation results seen in Section 5.3.1 was the implementation of the unused context pool. After implementing *context stealing* (see Section 3.4.2) the performance increased: the average runtime of `fib(30)` went down from 262.9ms to 240.5ms as can be seen in Figure 5.2. In other words, context stealing reduces the runtime to 91% of the original runtime.

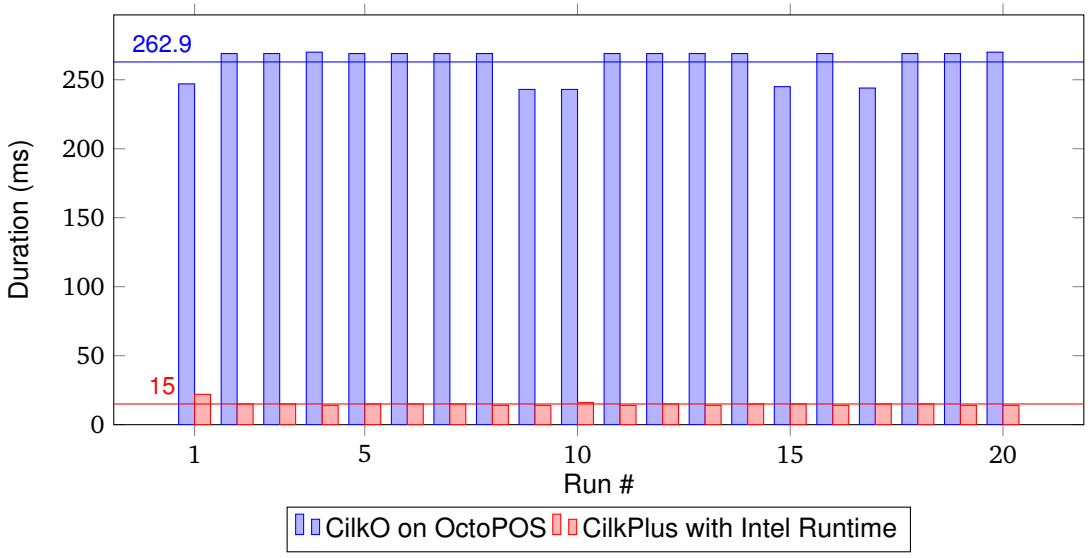

Figure 5.1 – Execution time of `fib(30)` using CilkO and Cilk Plus.

This is likely credited to the common case, where a core will be able to retrieve a context out of its local context queue without any atomic primitives like CAS being involved, being more efficient as the previously used approach in OctoPOS to manage the free context pool, where lookup of a free context always required at least one CAS operation. Only if the local context queue is found empty, the core resorts to steal attempts from remote queues which involves CAS.

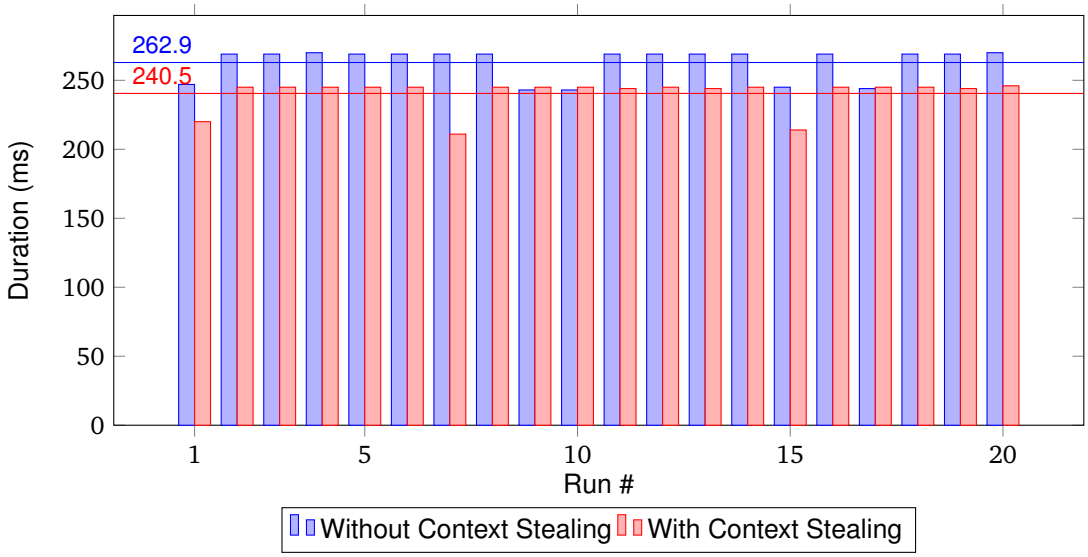

Figure 5.2 – Execution time of `fib(30)` with and without context stealing.

5.3.3 Core-Local Storage

The next optimization applied to OctoPOS was the addition of *Core-Local Storage (CLS)*. It can be seen as OctoPOS's equivalent to Thread Local Storage (TLS) and is in fact implemented in a way similar to how contemporary operating systems like Linux implement TLS: Using the `%fs` segment register on x86-64 to point to the core local storage. But since OctoPOS has no concept of threads, therefore we speak of Core-Local Storage (and in fact the value of the `%fs` register never changes once set for a core).

The addition of core local datastructures, that is, the queues for work (Section 3.4.1) and context (Section 3.4.2) stealing, was the main motivation for adding CLS support to OctoPOS. Without CLS, a core needs to first look up its own ID, which was done via a read from a memory mapped Local Advanced Programmable Interrupt Controller (LAPIC) register, and then used this ID as index for the array of queues to retrieve its own queue.

Right now only the core ID is stored in CLS, avoiding the read from the memory mapped LAPIC register. Per core datastructures like the work-stealing queues still reside consecutively in an array. But as Figure 5.3 shows, avoiding reading the core ID from the LAPIC and using CLS instead already further decreases the runtime down to 88% of the original runtime, from 234.5ms to 206.9ms.

5.3.4 Disabling Continuation Stealing

A major improvment in performance was achieved when compiling the benchmark with the `fno_-octo_cilk_support` CilkO Clang switch. This disables continuation stealing (Section 3.5.1) and results in child stealing being used instead. Figure 5.4 shows that this reduced the runtime from

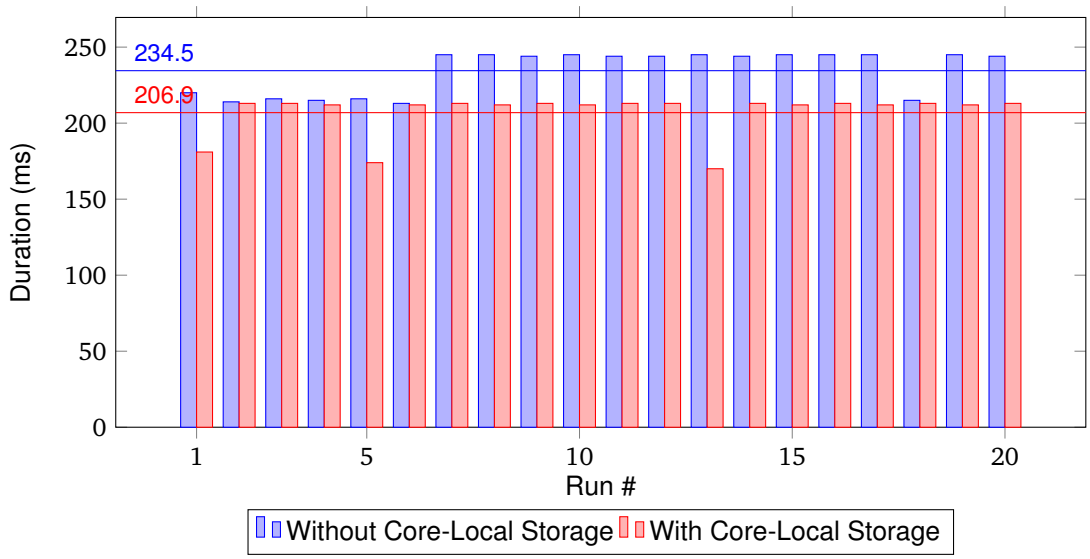

Figure 5.3 – Execution time of `fib(30)` with and without core-local storage.

206.9ms down to 117ms, an reduction to 56.5% of the orignal runtime. This is, however, still *7.8* times slower then Cilk Plus under Linux.

But it shows that a major drain of performance is caused by the context switches performed on *every* `cilk_spawn` when continuation stealing is active. The act of storing the registers and restoring them later on, which happens on a context switch, appears to be a major hurdle. And even if CilkO is used with child stealing the overhead of creating and spawning the *i*-let-helper function appears to be to big to compete with Cilk Plus. As explained in 2.2.6 Cilk Plus solves this in a more efficient way. Namely

- the registers are only restored if the continuation was actually spawned.
- the spawned function call is invoked without a detour through an helper function.

5.4 CilkO's Speedup

An important factor is the speedup a concurrency platform is able to provide. The following shows the speedup using a benchmark program consisting of 5.13, which spawns a certain number of work packages with a configurable workload factor. This factor determines the amount of work every work package performs, by acting as upper bound of the iterators of a `for` loop whose body performs some arithmetic operations and writes the result to a fixed size array in memory. The evaluation was performed using the fastbox system and with a total of 9330 spawned workload packages. As can be seen in Figure 5.5 CilkO and OctoPOS achieve nearly linear speedup for a workload of 10000 or more. But with increasing workload the scheduling overhead starts to eat up the advantage of the available parallelism.

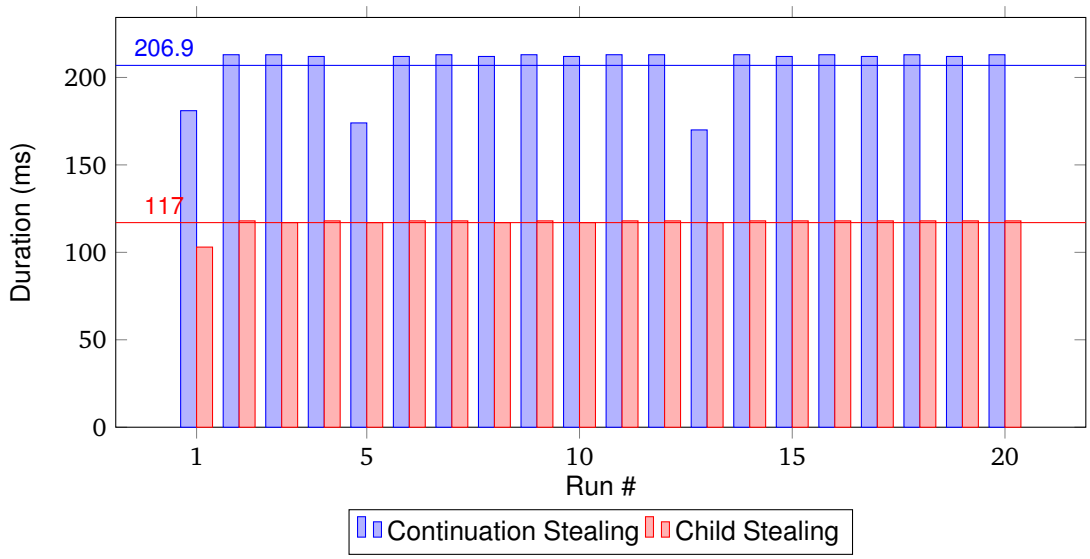

Figure 5.4 – Execution time of `fib(30)` with and without continuation stealing.

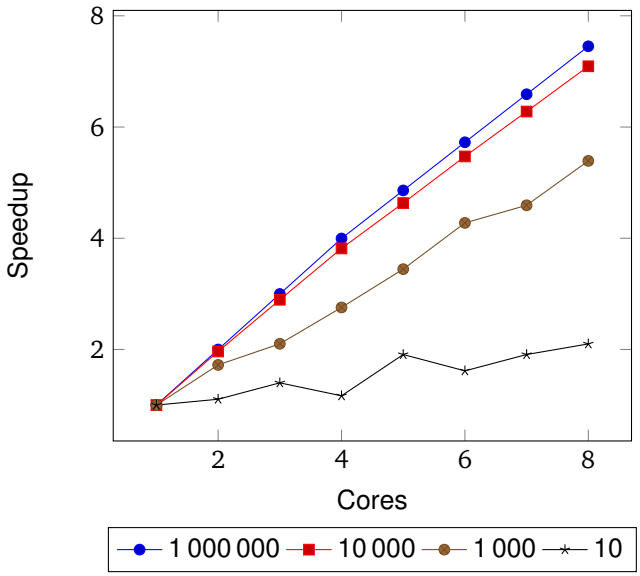

Figure 5.5 – Speedup of CilkO running on Fastbox and different workload levels.

The speedup results, as shown in Figure 5.6, on the Bigbox yield similar results. When running the benchmark on this system, OctoPOS was configured to use at most 20 cores of the same NUMA domain, since this is number of cores per NUMA domain in this system. The graphs show that CilkO on the Bigbox is not as much affected by a speedup tail-off using a workload of 1,000 compared to the resulsts of the Fastbox. This is likely due the slower clock rate of its processor when compared to the Fastbox. Also the slope of the speedup graphs is slightly decreasing after 10 or more cores, this is likely due to hyper threading, i.e. the because the 20 cores of the used NUMA domain consists of 10 real cores.

5.5 Work-Stealing Queues

Benchmarks of the two work-stealing queue types, ABP (Section 4.2.1) and CL (Section 4.2.2), yielded nearly identical results. Obviously the overhead introduced by the additional modulo operation in the CL queue is negligible for the overall performance. On the other hand, the ABP queue's underutilization defect likely rarely happens in this benchmark scenario. The pathological case which would trigger the defect causing an underutilized ABP queue from rejecting further work while its backing array still has free elements could be constructed if we consider a long-running *i*-let, a full or nearly full queue of the core executing the long-running *i*-let together with thieves stealing from the queue just up to the point where the queue becomes nearly empty so that the reset on empty heuristic is not triggered. If the long running *i*-let then tries to add additional work, the queue may reject the work, because the bottom pointer has reached its limit, with the top pointer just a few items behind, while there are still empty slots, i.e. the slots behind the top pointer, in the

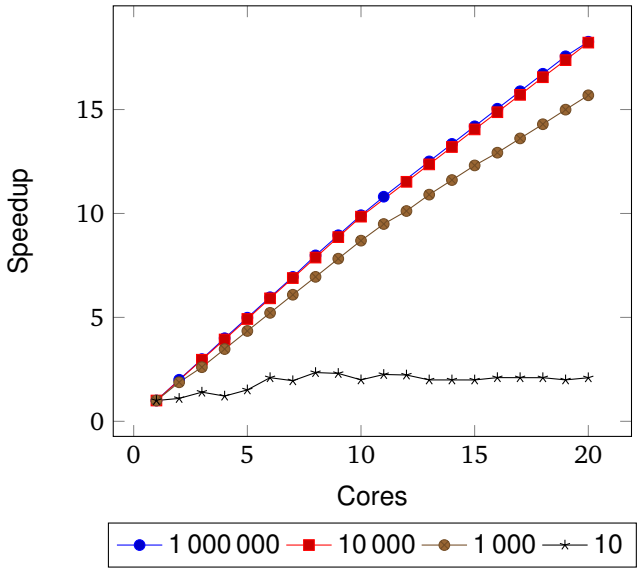

Figure 5.6 – Speedup of CilkO running on Bigbox with different workload levels.

backing array. While not impossible, this seems to be a very uncommon scenario, especially in the used benchmark, where the work load for every package is identical.

But since the CL queue yields the same performance and does not suffer from this defect, it is a good candidate for the default work-stealing queue implementation.

5.6 Continuation Stealing

While Section 5.3.4 shows that disabling continuation stealing in favor of child stealing with the current implementation yields a major performance advantage, it is trivial to produce a test case where child stealing fails while continuation stealing succeeds. To show this effect, the benchmark was configured so that it uses a low maximum recursion depth of 2, but a breadth of 1000. As result every work package will spawn 1000 further work packages in a `for` loop. A benchmark configured this way and compiled with the `-fno_octo_cilk_support` CilkO flag results in an aboration of the execution because OctoPOS is unable to enqueue the created *i*-lets for the work packages, whereas the benchmark terminates successfully if the flag is not used. Of course one could increase the default queue size of 256 in OctoPOS, but this would only protract the effect.

And while c5.3.4 showed that disabling continuation stealing yields an performance advantage when very small work packages are used (like it is the case in the used `fib()` function), this advantage decreases significantly when we increase the size of the work per work package. Table 5.2 shows the benchmark performed on the Bigbox with a work load of 1,000,000 and creating 9331 work packages. The numbers of continuation stealing and child stealing are nearly identical. In cases where 10 or more CPUs are used, child stealing performs even a little worse than continuation

stealing. This is likely caused by the increased locality since only a small amount of the queue's backing array is touched if continuation stealing is used.

Cores	1	5	10	15	20
Continuation Stealing	160348	32162	16089	11256	8664
Child Stealing	160295	32132	16117	11352	8796

Table 5.2 – Runtime in milliseconds with continuation and child stealing

```c
#include <stdio.h>
#include <octopos.h>

typedef struct {
  int n;
  int *result;
  simple_signal *signal;
} fibParams;

static void fib(fibParams *params) {
  int n = params->n;
  int *result = params->result;
  simple_signal *signal = params->signal;

  if (n < 2) {
    *result = n;
  } else {
    simple_signal newSignal;
    simple_signal_init(&newSignal, 2);

    int a, b;
    fibParams newParams1;
    newParams1.n = n - 1;
    newParams1.result = &a;
    newParams1.signal = &newSignal;
    fibParams newParams2;
    newParams2.n = n - 2;
    newParams2.result = &b;
    newParams2.signal = &newSignal;

    simple_ilet ilets[2];
    simple_ilet_init(&ilets[0], (void(*)(void*)) fib, &newParams1);
    simple_ilet_init(&ilets[1], (void(*)(void*)) fib, &newParams2);
    infect(get_claim(), ilets, 2);

    simple_signal_wait(&newSignal);
    *result = a + b;
  }
  simple_signal_signal(signal);
}

void main_ilet(claim_t claim) {
  simple_signal signal;
  simple_signal_init(&signal, 1);

  int result;
  fibParams params;
  params.n = 11;
```

```c
#include <cilk/cilk.h>
#include <stdio.h>

#include <octopos.h>

static int fib(int n) {
  if (n < 2)
    return n;

  int a = cilk_spawn fib(n-1);
  int b = cilk_spawn fib(n-2);

  cilk_sync;
  return a + b;
}

void main_ilet(claim_t claim) {
  int result = fib(11);
  printf("Result: %d\n", result);
}
```

Listing 5.15 – Fibonacci on OctoPOS with Cilk

6 CONCLUSION

> I may not have gone where I intended to go, but I think I have ended up where I needed to be.
>
> Douglas Adams
> *The Long Dark Tea-Time of the Soul*

One of the goals of the work this thesis deals with was to replace the Cilk runtime completely by an operating system (OctoPOS), hence bringing the execution model of the Cilk runtime down to the operating system. This goal was achieved in two places: Firstly, by creating a specialized version of the LLVM C compiler Clang which transforms Cilk's linguistic extensions to system calls provided by OctoPOS, we are able to transform Cilk code into binaries tailored for OctoPOS. Secondly, by modfying OctoPOS's scheduling algorithm to use work stealing.

One obvious benefit of using the Cilk keywords `cilk_spawn` and `cilk_sync` instead of OctoPOS's native API to achieve parallelism is the reduced source code size. This relieves the developer from resorting to complex, error-prone constructs, instead he or she can pick the simple-to-use Cilk keywords to denote parllelism.

A new concurreny platform was created with CilkO on OctoPOS, and like all new products it will get compared to existing solutions. When the work on this thesis started, it seemed trivial to outperform a concurrency platform using a full-fledged operating system with a solution that tailors `cilk_spawn` calls onto an operating system designed for future many-core architectures.

But as it turned out, CilkO's speedup when used with micro-parallelism is far from ideal when compared with Cilk Plus. The analysis shows that the current approach of spawning the `cilk_spawn` annotated function, i.e. creating an *i*-let for it and using OctoPOS primitives to allow parallel execution of this *i*-let, was not the best approach, as it is not suitable to schedule massive micro-parallelism in an efficient manner. Instead it is the continuation of the spawned function which must be made available for stealing by other workers, while the actual spawned function is invoked more or less as usual. Finally, what happens after the spawned function returns must depend on whether the continuation was stolen or not. If it was not, which is the common case, then execution

continues as usual. Otherwise execution of the current function must stop, as its continuation is already executed elsewhere.

This is Cilk Plus's excellent implementation of the work-first principle. And this is what future work on CilkO should focus on. Implementing this approach should be feasible, but likely requires more effort by the CilkO compiler and hence more work on the compiler.

On the other hand, the analysis has shown that the context switching overhead becomes a decreasing factor in the overall performance of CilkO when macro-parallelism is scheduled. CilkO performs reasonably well using sufficiently big work tasks.

The work of this thesis also motivated, caused and brought along many improvements within OctoPOS. Context stealing, implemented as part of working on this thesis, is a novel approach to efficently share resources like stack frames between consumers. Also, the insight arose that core-local storage is required in order to allow an efficient way to look up datastructures exclusivly owned by the core or the current state of the core.

Bringing the execution model down to the operation system also relieves developers from working around potential deadlock scenarious by avoiding potentially blocking calls. OctoPOS is fully aware when a system call will block and is able to simply execute another ready-to-run *i*-let in an efficient manner. It therefore does not require any kind of lazy task promotion as described in [Zak+15].

CilkO on OctoPOS allows the developer to write efficient parallel code which is able to scale on future many-core architectures. The free lunch may be over, but with CilkO developers get a tool to grow their own food.

LIST OF ACRONYMS

AST	Abstract Syntax Tree
CAS	Compare-And-Swap
dag	Directed Acylic Graph
IR	Intermediate Representation
PGAS	Partitioned Global Address Space
LAPIC	Local Advanced Programmable Interrupt Controller
MPSoC	Multi-Processor System on a Chip
NUMA	Non-uniform Memory Access
SSA	Static Single Assignement
TLS	Thread Local Storage

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

REFERENCES

[ABP98] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. "Thread Scheduling for Multiprogrammed Multiprocessors." In: *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '98. Puerto Vallarta, Mexico: ACM, 1998, pp. 119–129. ISBN: 0-89791-989-0. DOI: 10.1145/277651.277678. URL: http://doi.acm.org/10.1145/277651.277678.

[Acm] *ACM SIGPLAN - Most Influential PLDI Paper Award*. URL: http://www.sigplan.org/Awards/PLDI/ (visited on 06/26/2015).

[BL93] Robert D. Blumofe and Charles E. Leiserson. "Space-efficient Scheduling of Multithreaded Computations." In: *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*. STOC '93. San Diego, California, USA: ACM, 1993, pp. 362–371. ISBN: 0-89791-591-7. DOI: 10.1145/167088.167196. URL: http://doi.acm.org/10.1145/167088.167196.

[BL97] Robert D. Blumofe and Philip A. Lisiecki. "Adaptive and Reliable Parallel Computing on Networks of Workstations." In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '97. Anaheim, California: USENIX Association, 1997, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1268680.1268690.

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

[Bre74] Richard P. Brent. "The Parallel Evaluation of General Arithmetic Expressions." In: *J. ACM* 21.2 (Apr. 1974), pp. 201–206. ISSN: 0004-5411. DOI: 10.1145/321812.321815. URL: http://doi.acm.org/10.1145/321812.321815.

[Cha+05] Philippe Charles et al. "X10: An Object-oriented Approach to Non-uniform Cluster Computing." In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: ACM, 2005, pp. 519–538. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094852. URL: http://doi.acm.org/10.1145/1094811.1094852.

[Cha13] Bradford L. Chamberlain. “A Brief Overview of Chapel.” pre-print of a chapter that is to appear in an upcoming programming models book. Jan. 2013. URL: http://chapel.cray.com/papers/BriefOverviewChapel.pdf.

[CL05] David Chase and Yossi Lev. “Dynamic Circular Work-stealing Deque.” In: *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’05. Las Vegas, Nevada, USA: ACM, 2005, pp. 21–28. ISBN: 1-58113-986-1. DOI: 10.1145/1073970.1073974. URL: http://doi.acm.org/10.1145/1073970.1073974.

[CM13] The Association for Computing Machinery. *ACM HONORS COMPUTING INNOVATORS*. http://awards.acm.org/press_releases/tech_awards_2012.pdf. Apr. 2013. URL: http://awards.acm.org/press_releases/tech_awards_2012.pdf.

[Cor+09] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[Cor14] Intel Corporation. *Cilk Plus/LLVM. An implementation of the Intel® Cilk™ Plus C/C++ language extensions in LLVM*. https://cilkplus.github.io/. 2014. URL: https://cilkplus.github.io/.

[Dij65] E. W. Dijkstra. “Solution of a Problem in Concurrent Programming Control.” In: *Commun. ACM* 8.9 (Sept. 1965), pp. 569–. ISSN: 0001-0782. DOI: 10.1145/365559.365617. URL: http://doi.acm.org/10.1145/365559.365617.

[Dij67] Edsger W. Dijkstra. “The Structure of the &Ldquo;the&Rdquo;-multiprogramming System.” In: *Proceedings of the First ACM Symposium on Operating System Principles*. SOSP ’67. New York, NY, USA: ACM, 1967, pp. 10.1–10.6. DOI: 10.1145/800001.811672. URL: http://doi.acm.org/10.1145/800001.811672.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. “The Implementation of the Cilk-5 Multithreaded Language.” In: *SIGPLAN Not.* 33.5 (May 1998), pp. 212–223. ISSN: 0362-1340. DOI: 10.1145/277652.277725. URL: http://doi.acm.org/10.1145/277652.277725.

[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: ACM, 2009, pp. 79–90. ISBN: 978-1-60558-606-9. DOI: 10.1145/1583991.1584017. URL: http://doi.acm.org/10.1145/1583991.1584017.

[Goh11] Dan Gohman. *LLVM IR is a compiler IR*. Oct. 2011. URL: http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-October/043719.html.

[Gra69] R. L. Graham. “Bounds on Multiprocessing Timing Anomalies.” In: *SIAM Journal on Applied Mathematics* 17.2 (1969), pp. 416–429.

[HD68] 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*. AFIPS '68 (Spring). Atlantic City, New Jersey: ACM, 1968, pp. 245–251. DOI: 10.1145/1468075.1468111. URL: http://doi.acm.org/10.1145/1468075.1468111.

[Ima11] Shams Mahmood Imam. "Habanero-Scala: A Hybrid Programming model integrating Fork/Join and Actor models." PhD thesis. Rice University, 2011.

[IS12] Shams M. Imam and Vivek Sarkar. "Integrating Task Parallelism with Actors." In: *SIGPLAN Not.* 47.10 (Oct. 2012), pp. 753–772. ISSN: 0362-1340. DOI: 10.1145/2398857.2384671. URL: http://doi.acm.org/10.1145/2398857.2384671.

[JTC11] ISO/IEC JTC1/SC22/WG14. *Programming languages — C*. Standard. Geneva, CH: International Organization for Standardization, 2011.

[Kan11] Jin-Gu Kang. "More Target Independent LLVM Bitcode." http://llvm.org/devmtg/2011-09-16/EuroLLVM2011-MoreTargetIndependentLLVMBitcode.pdf. Sept. 2011. URL: http://llvm.org/devmtg/2011-09-16/.

[KHM89] D. A. Kranz, R. H. Halstead Jr., and E. Mohr. "Mul-T: A High-performance Parallel Lisp." In: *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*. PLDI '89. Portland, Oregon, USA: ACM, 1989, pp. 81–90. ISBN: 0-89791-306-X. DOI: 10.1145/73141.74825. URL: http://doi.acm.org/10.1145/73141.74825.

[Kli] Manuel Klimek. *The Clang AST - a Tutorial*. URL: https://youtu.be/VqCkCDFLSsc?t=349.

[LA04] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–. ISBN: 0-7695-2102-9. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

[Lat02] Chris Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization." MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, 2002. URL: http://llvm.org/pubs/2002-12-LattnerMSThesis.html.

[Lat11] Chris Lattner. "LLVM." In: *The Architecture Of Open Source Applications*. 2011. Chap. 11.

[Lea00] Doug Lea. "A Java Fork/Join Framework." In: *Proceedings of the ACM 2000 Conference on Java Grande*. JAVA '00. San Francisco, California, USA: ACM, 2000, pp. 36–43. ISBN: 1-58113-288-3. DOI: 10.1145/337449.337465. URL: http://doi.acm.org/10.1145/337449.337465.

[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: ACM, 2010, pp. 411–420. ISBN: 978-1-4503-0178-7. DOI: 10.1145/1854273.1854324. URL: http://doi.acm.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: ACM, 2009, pp. 522–527. ISBN: 978-1-60558-497-3. DOI: 10.1145/1629911.1630048. URL: http://doi.acm.org/10.1145/1629911.1630048.

[MC08] Sape Mullender and Russ Cox. “Semaphores in Plan 9.” In: *Proceedings of 3rd International Workshop on Plan 9*. IWP9. Volos, Greece, 2008, pp. 53–62. URL: http://doc.cat-v.org/plan_9/IWP9/2008/iwp9_proceedings08.pdf.

[Moh+15] Manuel Mohr et al. “Cutting out the Middleman: OS-level Support for x10 Activities.” In: *Proceedings of the ACM SIGPLAN Workshop on X10*. X10 2015. Portland, OR, USA: ACM, 2015, pp. 13–18. ISBN: 978-1-4503-3586-7. DOI: 10.1145/2771774.2771775. URL: http://doi.acm.org/10.1145/2771774.2771775.

[Oec+11] Benjamin Oechslein et al. “OctoPOS: A Parallel Operating System for Invasive Computing.” In: *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA’11)*. Ed. by Ross McIlroy et al. Vol. USB Proceedings. Salzburg, 2011, pp. 9–14. URL: http://www4.cs.fau.de/~benjamin/documents/octopos_sfma2011.pdf.

[Rob14] Arch Robison. *A Primer on Scheduling Fork–Join Parallelism with Work Stealing*. Tech. rep. N3872. ISO/IEC JTC 1/SC 22/WG 21—The C++ Standards Committee, Jan. 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3872.pdf.

[Sut05] Herb Sutter. “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.” In: *Dr. Dobb’s Journal* 30.3 (2005), pp. 202–210. URL: http://www.gotw.ca/publications/concurrency-ddj.htm.

[Tei+11] Jürgen Teich et al. “Invasive Computing: An Overview.” In: *Multiprocessor System-on-Chip – Hardware Design and Tool Integration*. Ed. by M. Hübner and J. Becker. Springer, Berlin, Heidelberg, 2011, pp. 241–268.

[Zak+15] Christopher Zakian et al. *Concurrent Cilk: Lazy Promotion from Tasks to Threads in C/C++*. 2015.

CURRICULUM VITAE

Florian Schmaus

Incarnation date (Unix time) 446822022

	Akademische Bildung
2012 – 2015	MSc, Informatik, Nebenfach: Psychologie, FAU Erlangen
2009 – 2012	BSc, Informatik, Nebenfach: Astronomie, FAU Erlangen
	Schulbildung
2008 – 2009	fachgebundene Hochschulreife, Berufsoberschule Nürnberg
2007 – 2008	Fachhochschulreife, Berofsoberschule Erlangen
	Berufserfahrung
2012 – 2014	Studentische Hilfskraft, Tutor Systemprogrammierung 2, FAU Erlangen
2011 – 2012	Werkstudent, Siemens AG, Healthcare, Erlangen
2007 – 2008	Werkstudent, Siemens AG, IT-Operations (ITO), Erlangen
2003 – 2007	Angestellter, Siemens AG, IT-Operations (ITO), Erlangen
2000 – 2003	Ausbildung, IT-Systemelektroniker, Siemens AG