**Main Idea:**

dsl to scala ir (not part of project), ir to rust

front end: IR which can represent rust

back-end: take ast convert to rust binaries

**Proposal 1:**

Construct a DSL which can describe a stream processor and be converted into a RustAST

**Proposal 2:**

Take RustAST and compose to rust code. Use rust compiler (optionally Cargo) to get executables for specific architectures.

**Spark:**

*Source: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*

Spark is an expressive distributed computing framework running on the Java virtual-machine (VM) **[need source]**. Spark aims to improved performance by avoiding moving data eagerly to external stable storages. Other popular distributed computing frameworks prior to Spark moved intermediate result to an external stable storage, as a distributed database, to guarantee fault tolerance. This impedes their execution and decrease performance significantly. Spark introduced Resilient Distributed Datasets (RDDs) as an alternative. RDDs are abstract execution plans on data stored in stable storage. RDDs are lazily evaluated. No intermediate results are calculated before a result value from the expression the RDD represents is requested. The only exception to this is if the user explicitly instructs the intermediate result to be persisted to stable storage. At the time when the result value is requested, Spark will optimize the execution plan and distribute the workload over the cluster. The effects Spark had on performance was an improvement of great magnitude, up to 20x compared to Hadoop.

**Spark SQL:**

*Source: Spark SQL: Relational Data Processing in Spark*

Application analyzing Big Data tend to utilize a mixture of procedural algorithms and relational queries. Using Sparks RDDs or other distributed computing frameworks, as Hadoop, for relational queries can be tedious and complex. It requires a lot of manual optimization to match the performance of frameworks specialize for relational queries. To tackle this problem, Spark introduced the DataFrame concept along with an optimizer specifically developed for query optimization called Catalyst. DataFrames represents a distributed set of rows with a uniform schema. These rows can be manipulated using the DataFrame API, which consists of a set of lazily evaluated relational operators, similar to RDD transformations. A DataFrame can be transformed to an RDD of row objects. This bridge the gap between relational queries and procedural algorithms.

**Flare:**

*Source: Flare: Native Compilation for Heterogeneous Workloads in Apache Spark*

Sparks performance can suffer radically due to the intent to be expressive whilst keeping the distribution “under the hood”. Flare is an attempt to fix a part of the performance issue, focusing on DataFrames. Operations on a DataFrame will originally be executed on the Java VM. Running operations in a VM is inherently slower than native-code. Therefore, Flare transforms SQL queries on DataFrames to native code and improves performance significantly. Sparks query performance with Flare matched the performance of the best SQL engines without affecting Sparks expansive expressiveness. **(CDA is an attempt to make a complete system compile to native code, or at least as much as possible. Performance can probably be significantly improved doing this.)**

**Flink:**

*Sources: State Management in Apache Flink*

*Apache flink: Stream and batch processing in a single engine*

*https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/runtime.html*

Flink is an open source system focused on continuous distributed computing. Flink’s system will be scheduled once and long running. Therefore, it needs to be adaptive and fault tolerant. To achieve this, Flink employs a dynamic cluster architecture which is composed of three processes: the client, the Job Manager and the Task Managers.

The client will compile and optimize the logical pipeline before sending it to the Job Manager. The Job Manager is responsible for coordination, physical deployment and fault tolerance. The system can even handle Job Manger failures using leader election in ZooKeeper. The Task Managers are workers, using a single Java VM to execute tasks assigned to it.

**Weld:**

*Source: Weld Rethinking the Interface Between Data-Intensive Libraries*

Performance of data-intensive computations suffer heavily from data movement due to usage of different libraries within the same program. This is caused by the fact that compilers cannot optimize across library boundaries. Instead all intermediate results are forced to be calculated and stored in memory before passing control to an external library. Weld has shown that huge speedups in execution time can be achieved by addressing this problem. Weld implement a cross-library optimizer to enhance performance of data-intensive computations on bounded data. To enable the cross-library optimizations, each library must supply an IR representation of their program using Welds’ IR representation. This IR can be generated after the source code has been optimized to achieve domain specific optimizations, such as rearrangements of linear algebra operations. Weld then constructs a complete IR of the program containing all function calls across libraries. Then Weld optimizes the whole IR tree. The final resulting IR tree is then used to compile to hardware specific code.

Weld’s compile time, including optimization of the IR, ranged from 62-256ms. The mean compile time was 126ms and median 117ms. Since the target users will apply weld to programs analyzing huge datasets this compile time seems acceptable. The compilation time would be amortized constant time if Weld were applied to a long running stream processor.

Weld does not support asynchronous computations **(interesting in CDA?)**.

**Deep Embedding of DSLs:**

Most compilers for DSLs are deep embedded which means they work as a program generator. The compiler and the host language, the language the compiler is implemented in, only interprets the DSL as a data structure. The data structure will be optimized using domain specific optimizations and then transformed to a target language.

**Language Virtualization:**

*Source: Language Virtualization for Heterogeneous Parallel Computing*

General-purpose Languages are a great platform for general software development. But experts in specific problem domains may suffer from using General-purpose Languages. This is due to the requirement of General-purpose Languages to create correct code for a wide set of problem domains. They cannot optimize based on domain specific knowledge. Therefore, developing a DSL for the specific problem domain with a unique optimizer, built to take advantage of the optimizations availible in the problem domain, is usually beneficial. Additionally, it eases the development proecess for the domain expert by introducing a syntax and semantics more suitable to the domain.

In later years the development of new hardware components for parallel computing have proven to be a problem for parallel DSLs. When a new hardware platform is introduced, as an example CUDA or OpenMP, the step from DSL to native code has to be reimplemented in the compiler. This can require substatial effort based on the size of the DSL.

Language Virtualization is a technique to make this step easier.

Language Virtualization is similar to hardware virtualization in the sense that it is not required by developer to have specific knowledge of the underlying hardware.

**Ziria:**

The implementation of wireless protocol is none trivial due to the high throughput demands. Data in the rate of Gigabits/second have to be processed by a wireless router. Most protocols have therefore been implemented without much hardware abstractions. This enhances performance, since the programs are tailormade for the underlying hardware, but it makes the software development process more complex and error prone. Ziria is an DSL which has been implemented to enhance development and maintenance of wireless systems. It consists of two main components: stream transformer and stream computer.

The transformer takes an input stream, transform the data, and passes the transformed data to an output stream. The computer is similar, but besides consuming the input and emitting output, it produces another value called control value. The workflow of a stream computer is as follows, it continuously consumes input and produces output which is passed downstream. After a while the computer halts and emits the control value. The workflow of the pipeline is not delayed by the computers since they continuously emit output when calculating the control value. The control value can be metadata about the input stream. Ziria can use this to reconfigure the whole pipeline downstream. An example from the paper is a payload header decoder, where the computer emits a control value corresponding to which decoder should be used for the payload body. The rest of the downstream pipeline can then dynamically be initialized as the required decoder for the current input in the stream. **This might be interesting for CDA, where the stream processing pipeline may change depending on the input data.**

**Refernces:**

Language virtualization for heterogeneous parallel computing (2010)

Modular Domain Specific Languages and Tools (1998)