Adrian Sutherland
Rexx Language Symposium, Vienna

Introduction: Reinvigorating Rexx with High Performance

Rexx: An Enduring Legacy

- Designed for usability, scripting, and text processing.
- Strong presence in mainframe (VM, MVS, TSO/E), OS/2, and as a glue language.
- ANSI Standard X3.274-1996 provides stability.

The Performance Imperative: Bridging the Gap

- Traditional interpreters face limits due to Rexx's dynamism (typing, strings, runtime checks).
- Can be a bottleneck for demanding tasks.

CREXX: A Modern High-Performance Approach

- Ground-up implementation using contemporary VM techniques.
- Aims for significantly higher performance while preserving Rexx's strengths.

The CREXX Architecture: A Modern Foundation

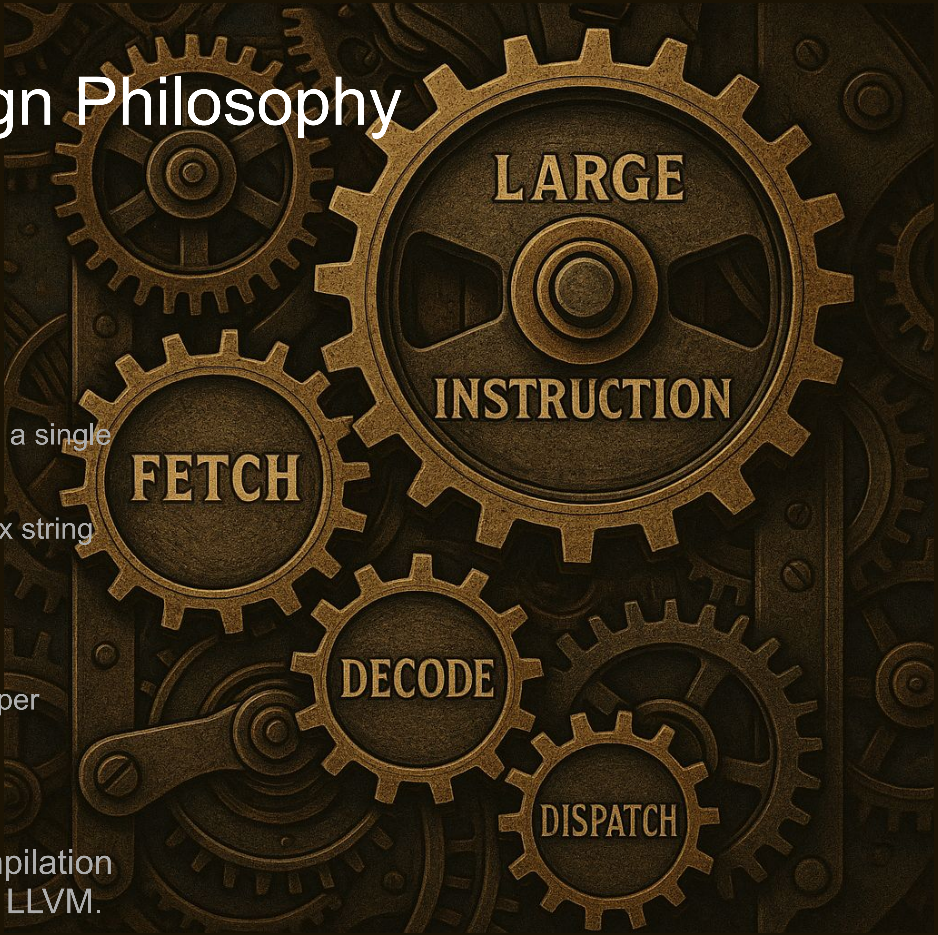
- Integrated Components
  - CREXX Assembler (RXAS): Defines the low-level input format
  - CREXX Virtual Machine (VM): The core execution engine (interpreter)
  - Plugin System: Enables extensibility and customization
  - Planned LLVM Converter: For future native code generation
- Leveraging Modern C/C++ Development
- Anticipated sophisticated compiler technologies like LLVM from the outset

The CREXX Assembler: Design Philosophy

Interface between a Rexx frontend and the CREXX VM

- Instruction set tailored for efficient VM execution
- Defining Characteristic: "Large Instructions"
  - Encapsulate complex, higher-level Rexx semantics in a single instruction
  - Example: A single instruction might perform a complex string function or arithmetic with implicit type handling
- Rationale: Optimizing Interpreter Performance
  - Fewer instruction fetch, decode, and dispatch cycles per high-level operation.
  - Directly reduces instruction dispatch overhead.

Trade-off: Can increase complexity for future JIT/AOT compilation due to the "Semantic Gap" when targeting simpler IRs like LLVM.



The CREXX Virtual Machine: Interpreter Core

The heart of the execution environment

- Execution model based on a core loop: fetch, decode, execute.
  - Key Performance Focus: Instruction Dispatch
  - Employing "Threading" techniques (e.g., Direct or Indirect Threaded Code).
- Implemented using compiler extension, computed gotos (GCC/Clang).

Goal: Minimize the overhead associated with dispatching to the correct instruction handler.

- Reduces branching and improves branch predictor efficiency compared to a traditional switch statement.
- Aims to make the interpreter loop itself as fast as possible.

Object Model and Data Representation

Supporting Rexx's dynamic typing and its single fundamental data type: the character string.

Includes the requirement for arbitrary-precision decimal arithmetic.

Unique Register Model: "Large Registers"

- Designed to hold complex data structures directly.
- Can contain "Child Registers," effectively acting as objects or structured data containers within the VM state.

Direct Representation of Rexx Stems:

- Mapping stem elements or substructures to child registers within a parent large register.
- Potential to avoid overhead of traditional implementations using external hash tables.

Numeric operations (e.g., using decNumber for arbitrary precision) can be integrated seamlessly via the plugin system.

The current object model is intrinsically linked to this large register/child register structure.

The Plugin Architecture: Extensibility

Provides extensibility and customization without requiring modifications to the core VM source code.

Allows external modules to enhance or modify the VM's behavior dynamically.

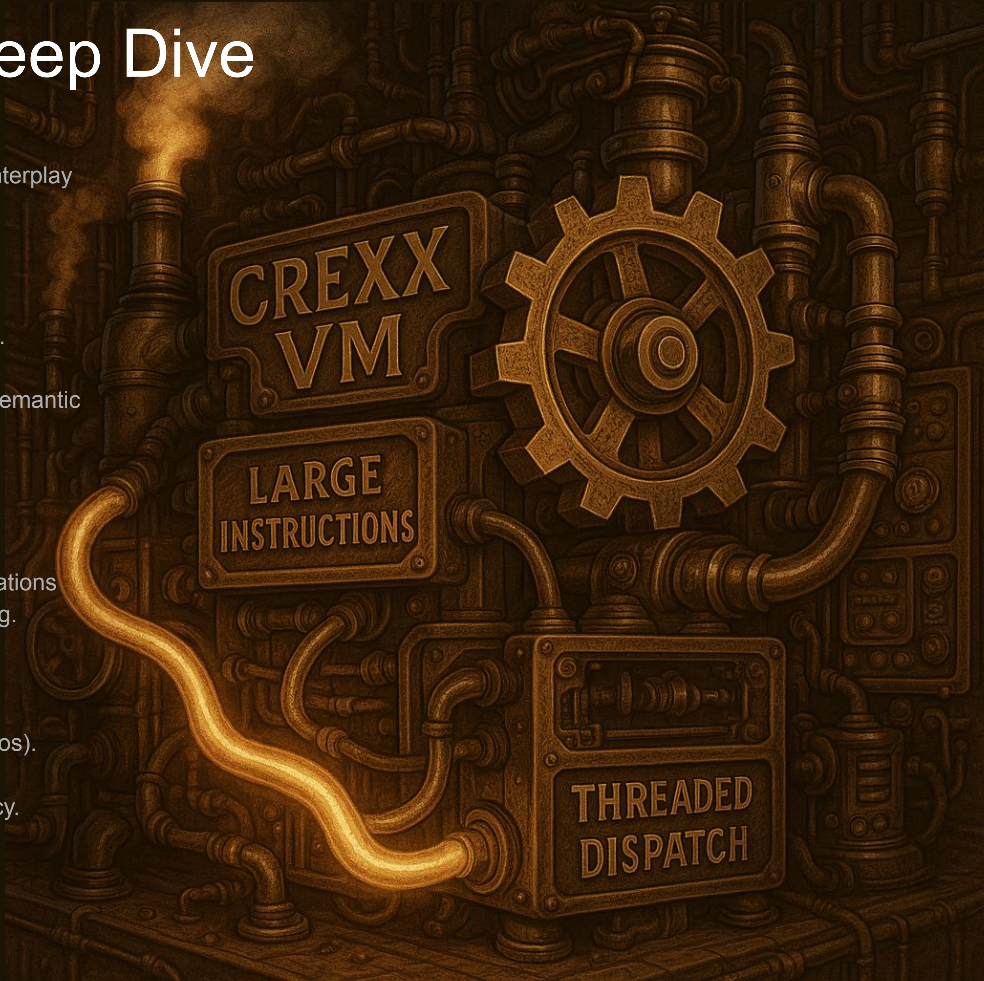
Two Primary Types

- Native Function Plugins: Seamlessly integrate functions written in native languages (C/C++). Leverage existing high-performance libraries.
- Instruction Implementation Plugins: Allow plugins to override the default implementation of specific CREXX assembler instructions. Example: Plugging in Mike Cowlishaw's decNumber library for arbitrary-precision arithmetic.

Offers remarkable flexibility, allowing core VM semantics (like arithmetic precision) to be customized or replaced.

Trade-off: Requires an indirection mechanism in the dispatch path, potentially introducing a small performance overhead compared to statically fixed instruction semantics. Signifies a design choice favoring flexibility.

Performance Optimisation Deep Dive



The pursuit of high performance is a key factor in design choices, particularly the interplay between large instructions, large registers, and threaded dispatch.

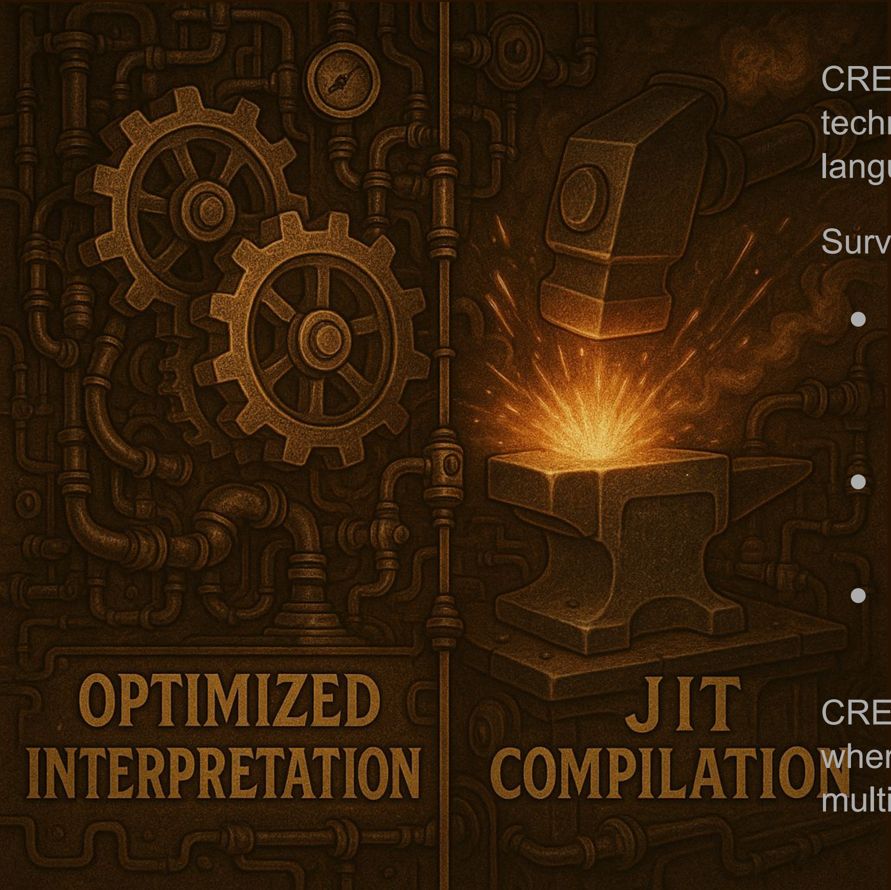
Large Instructions and Registers:

- Represents a deliberate departure from common VM design philosophies.
- Primary Advantage: Reduction of interpreter overhead. Fewer steps per semantic task, operations potentially work directly on complex data in registers.
- Aims to maximize the performance of the interpreter execution mode.
- Trade-offs: Increase pressure on instruction cache, require more complex decoding. Present a "semantic gap" when targeting lower-level representations like LLVM IR. Prioritizes optimizing interpretation over simpler JIT mapping.

Instruction Dispatch ("Threading"):

- Optimizes the interpreter's main execution loop (e.g., using computed gotos).
- Eliminates large, multi-way branches, improving branch predictor efficiency.
- A well-established technique for maximizing pure interpreter speed.

Comparative Analysis: CREXX in the VM Landscape



CREXX's design comparing with general VM performance techniques and high-performance implementations for other languages.

Survey of VM Performance Techniques

- Just-In-Time (JIT) Compilation: Translating bytecode/IR to native code at runtime. (Method-based, Tracing, Meta-tracing).
- Garbage Collection (GC): Automatic memory management to minimize pause times and maximize throughput.
- Optimizations: Compiler optimizations like inlining, dead code elimination, type specialization, inline caching

CREXX's initial focus was on optimizing the interpreter phase, whereas many modern high-performance VMs rely heavily on multi-tiered JIT compilation

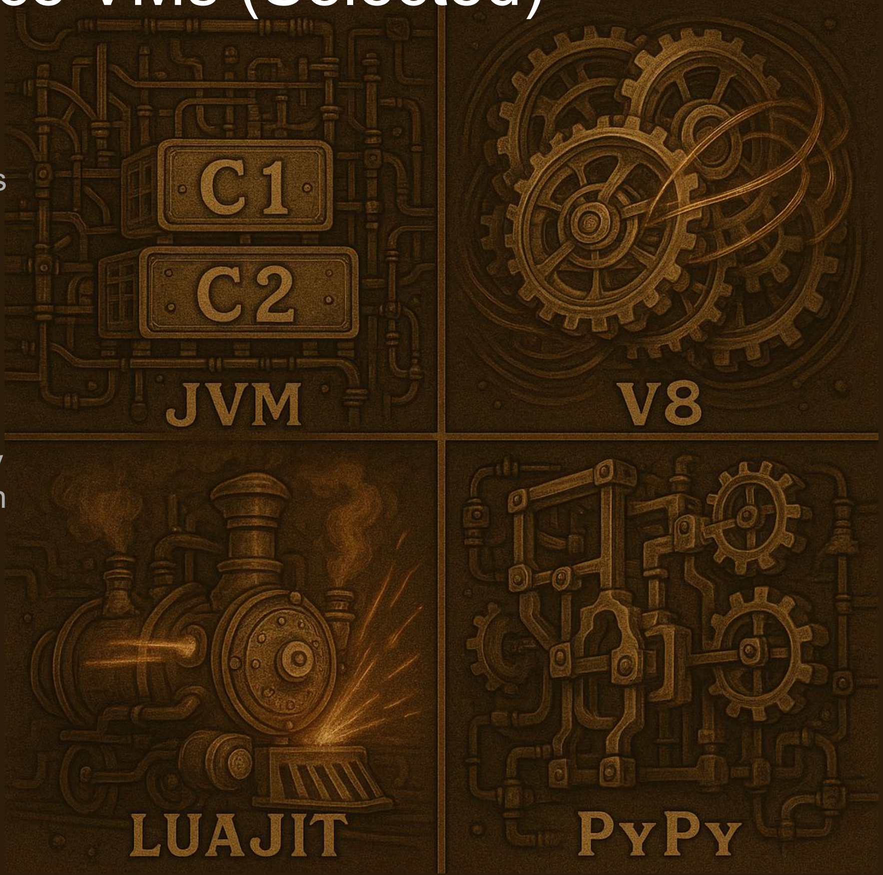
Case Studies: High-Performance VMs (Selected)

HotSpot JVM (Java): Mature, robust. Features a sophisticated multi-tiered JIT (C1, C2), adaptive optimization, advanced GC. Excellent peak performance for long-running applications, but has warm-up time.

V8 (JavaScript/WebAssembly): Developed by Google. Multi-tiered compilation pipeline (Ignition, Sparkplug, Maglev, TurboFan). Generational GC, aggressive inline caching, hidden classes. Optimized for dynamic web workloads, fast startup.

LuaJIT (Lua): Renowned for exceptional performance, especially numerical. Combines a highly optimized assembly interpreter with a tracing JIT. Very low FFI overhead. Can be sensitive to coding style, potential trace explosion.

PyPy (Python via RPython): Implementation framework with a meta-tracing JIT generator. Often significant speedups over CPython. Optimizes through layers of abstraction. Issues are warm-up time, and complexity of the meta-compilation framework.



Positioning CREXX

CREXX's target language, Rexx, is relatively niche but has unique features like powerful string manipulation and stem variables.

Design choices are strongly tailored to optimize these specific Rexx features within an interpreted context.

The large instruction/register philosophy is a departure from the trend towards simpler intermediate representations seen in many JIT-focused VMs (e.g., V8 Ignition bytecode, LLVM IR itself).

While most modern high-performance VMs rely heavily on multi-tiered JIT compilation, CREXX's initial focus is on maximizing interpreter performance through techniques like threaded dispatch and high-level instructions/registers.

The plugin system, particularly the ability to override instruction implementations, offers a unique and deep level of extensibility compared to typical FFI or embedding APIs.

The planned LLVM backend represents a significant future direction, aiming to leverage AOT/JIT compilation.

The success of the LLVM backend hinges on overcoming the significant challenge of mapping CREXX's high-level VM architecture onto LLVM's low-level IR without sacrificing performance.

Future Directions: Enhancing the VM Core - Fast Multi-branching

A planned enhancement involves integrating a mechanism termed "ACPH" to accelerate multi-way branching.

Purpose is to optimize constructs like Rexx's SELECT statement and potentially internal VM dispatch scenarios involving multiple choices.

Addresses the inefficiency of standard implementations (sequential comparisons, simple jump tables) for complex conditions or a large number of branches.

It involve heuristics or pathfinding concepts to determine an optimal evaluation order or structure for complex multi-branch scenarios.

Optimizing these control flow structures directly within the VM could provide significant performance improvements for certain classes of Rexx programs (including advanced parsing), complementing instruction-level optimizations.

Future Directions: Evolving the Object Model

The initial object model, based on large registers containing child registers, is slated for significant evolution to provide a more complete and flexible object foundation.

Key Planned Enhancements:

- Text-indexed Child Registers: Allowing child registers (representing object properties or stem variable elements) to be looked up using arbitrary text strings as keys. Necessitates implementing an efficient underlying data structure (binary tree or hash table) integrated with the register system. Brings the model closer to the dynamic nature of Rexx stems and typical dictionary/map implementations.

  Trade-off: Introduces lookup overhead compared to direct indexing.

- Function Lookups for Registers: Including the ability to associate functions directly with registers/objects. Implementing method dispatch capabilities within the VM's core object model. Allows behavior (code) to be tied directly to the VM's primary data structures.

These enhancements support a trajectory towards a more conventional, albeit highly integrated, object system.

Future Directions: The CREXX Assembler to LLVM Converter - Motivation

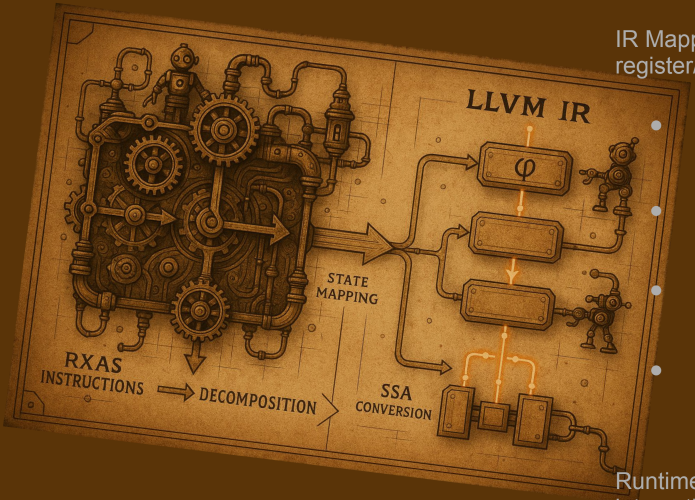
The most ambitious future plan is the development of a converter to translate CREXX Assembler code into LLVM Intermediate Representation (IR).

This holds the promise of unlocking significant performance gains and portability benefits.

Primary Motivations:

- Leveraging LLVM's Optimization Passes: Accessing LLVM's mature and powerful suite of optimization passes (e.g., opt).
- Highly Optimized Native Code Generation: Generating efficient machine code for a wide array of target architectures (x86, ARM, z/Architecture).
- Enabling Ahead-of-Time (AOT) Compilation: Compiling CREXX programs into standalone executables.
- Enabling a Just-In-Time (JIT) Compilation Backend: Moving towards a tiered execution model, combining the optimized interpreter for startup/cold code with LLVM-generated native code for hot spots. This mirrors strategies used by VMs like JVM and V8.

Future Directions: The CREXX Assembler to LLVM Converter - Technical Approach Outline



Parsing and Representation: The converter must parse the RXAS code and build an internal representation for translation.

IR Mapping (The Core Challenge): Translating CREXX's high-level "large instructions" and the complex "large register/child register" state onto LLVM's low-level, stateless, RISC-like, SSA-based IR.

- Instruction Decomposition: Breaking down single large RXAS instructions into sequences of multiple simpler LLVM IR instructions (loads, stores, arithmetic ops, etc.).
- State Representation: Representing large and child registers using LLVM constructs like struct types, alloca, getelementptr (GEP), and load/store instructions.
- Dynamic Typing: Handling Rexx's dynamic typing within the statically typed LLVM IR (e.g., using tagged pointers or boxing values).
- SSA Conversion: Translating the potentially mutable RXAS register model into LLVM's Static Single Assignment (SSA) form, potentially leveraging LLVM's mem2reg pass or explicitly inserting phi nodes at control flow merges.

Runtime Library: Implementing complex Rexx semantics (arbitrary-precision arithmetic, intricate string ops, plugin interactions) in a C/C++ runtime library that the generated LLVM code would call into.

LLVM Integration: Using LLVM's APIs (C++ API or C bindings) to programmatically construct the LLVM IR module.

Optimisation and Code Generation: Running LLVM's standard optimisation passes (opt) followed by the backend (llc) to generate native code.

Future Directions: The CREXX Assembler to LLVM Converter - Anticipated Challenges

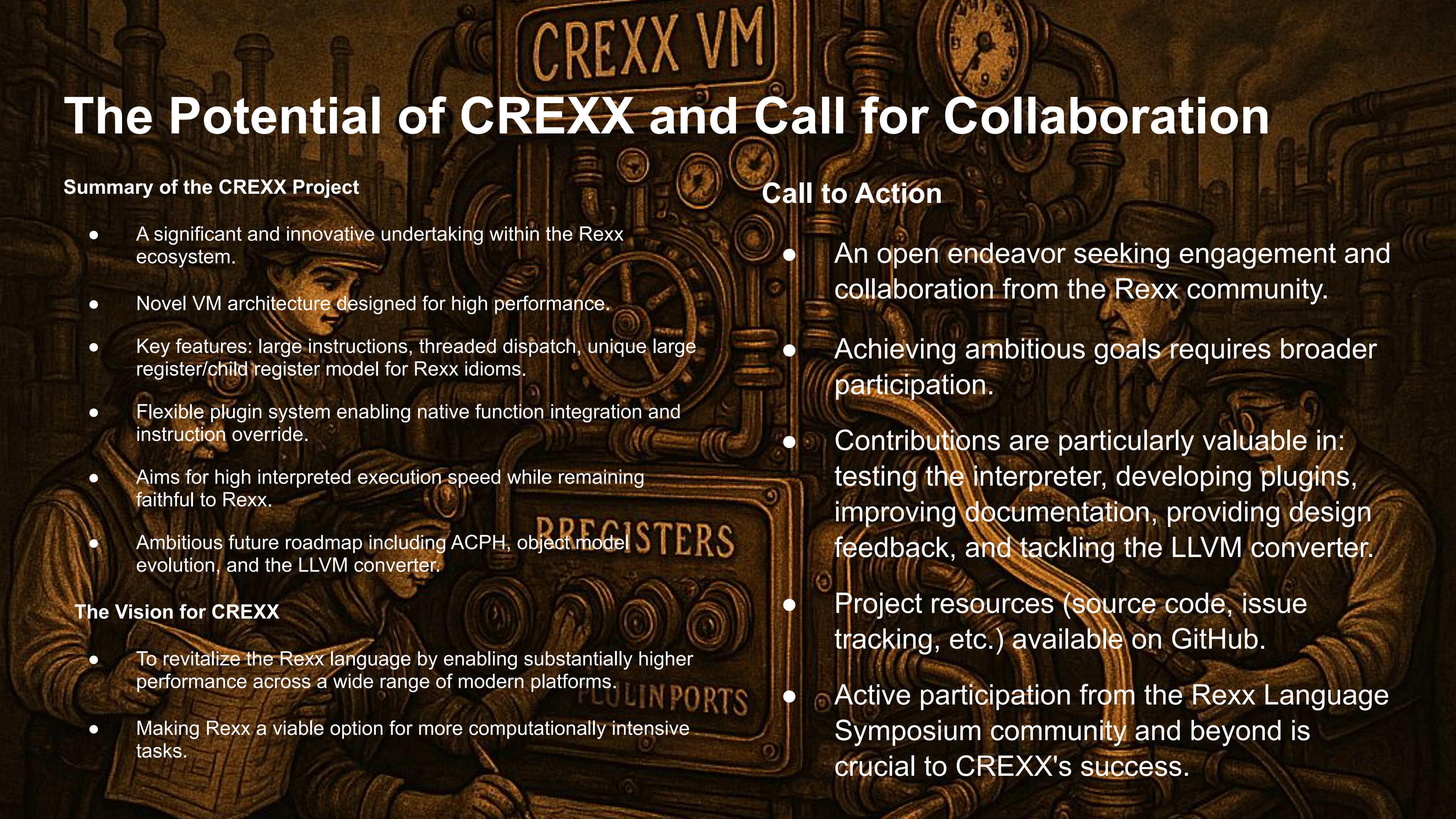
Semantic Gap: The fundamental mismatch between the high-level abstraction of CREXX assembler/VM and the low-level nature of LLVM IR is the primary challenge. Bridging this gap efficiently is non-trivial.

State Mapping Complexity: Translating the potentially complex state held in large/child registers into LLVM's memory model using structs, pointers, and GEP requires careful design to ensure correctness and performance.

Potential Performance Overheads: The decomposition of large instructions, the overhead of handling dynamic typing, and the necessity of calls to a runtime library could potentially negate the benefits of LLVM's optimizations, possibly resulting in code slower than the highly optimized CREXX interpreter. LLVM's own JIT process can also introduce latency and memory overhead.

LLVM Complexity: The LLVM framework itself is vast and complex, with a steep learning curve. Debugging issues within LLVM or in the generated code can be challenging.

The design choices made to optimize the interpreter (large instructions/registers) inherently make this translation task more complex than for VMs with simpler instruction sets.



The Potential of CREXX and Call for Collaboration

Summary of the CREXX Project

- A significant and innovative undertaking within the Rexx ecosystem.
- Novel VM architecture designed for high performance.
- Key features: large instructions, threaded dispatch, unique large register/child register model for Rexx idioms.
- Flexible plugin system enabling native function integration and instruction override.
- Aims for high interpreted execution speed while remaining faithful to Rexx.
- Ambitious future roadmap including ACPH, object model evolution, and the LLVM converter.

The Vision for CREXX

- To revitalize the Rexx language by enabling substantially higher performance across a wide range of modern platforms.
- Making Rexx a viable option for more computationally intensive tasks.

Call to Action

- An open endeavor seeking engagement and collaboration from the Rexx community.
- Achieving ambitious goals requires broader participation.
- Contributions are particularly valuable in: testing the interpreter, developing plugins, improving documentation, providing design feedback, and tackling the LLVM converter.
- Project resources (source code, issue tracking, etc.) available on GitHub.
- Active participation from the Rexx Language Symposium community and beyond is crucial to CREXX's success.