# Joule Technical Overview

**A Modern Language for Energy-Efficient Computing**

**David Jean Charlot, PhD** Open Interface Engineering, Inc. (openIE) University of California, Santa Barbara (UCSB) david@openie.dev | dcharlot@ucsb.edu

**February 2026 | Version 1.0**



## Abstract

Joule is a systems programming language designed for energy-efficient, high-performance computing. It combines modern language features—strong static typing, memory safety without garbage collection, and first-class support for concurrent and heterogeneous computing—with an explicit focus on energy awareness. This document provides a technical overview of Joule's design, capabilities, and the research foundations that inform its architecture.

**Keywords**: Joule, systems programming, energy efficiency, type system, ownership, heterogeneous computing, compiler architecture, LLVM, MLIR, Cranelift

## 1. Introduction

### 1.1 Design Goals

Joule addresses a critical gap in the programming language landscape: **energy efficiency as a first-class concern**. Research demonstrates that programming language choice can result in energy consumption differences of up to **75x** for equivalent computations [1]. Joule aims to provide:

- Energy consumption comparable to C/Rust (within 5% overhead)
- Memory safety without garbage collection
- First-class support for heterogeneous computing (CPU/GPU/TPU)

–

- Developer productivity through modern language features

### 1.2 Target Applications

- Systems software requiring predictable energy profiles
- Edge computing and IoT with power constraints
- High-performance computing with sustainability requirements
- AI/ML inference with energy budgets

### 1.3 Why Not Rust?

A fair question: Rust already provides memory safety without garbage collection and achieves energy efficiency comparable to C [1]. Why create a new language?

Joule builds on Rust's foundational insights while addressing different primary goals:

| Aspect | Rust | Joule |
|---|---|---|
| **Primary goal** | Memory safety | Energy efficiency |
| **Energy visibility** | None (external tools only) | First-class ( `@energy_budget` , `energy::Meter` ) |
| **Heterogeneous compute** | Via external libraries (CUDA, etc.) | First-class ( `@kernel` , `@target(GPU)` ) |
| **Hardware telemetry** | Manual integration | Built-in RAPL/thermal awareness (adjusts scheduling as hardware approaches thermal limits) |
| **Compiler backends** | LLVM only | Cranelift + LLVM + MLIR |
| **AI accelerator support** | Limited | Native via MLIR backend |

**The key differentiators:**

1. ***Energy as a language primitive****: Rust optimizes for safety; Joule optimizes for energy. Energy budgets, profiling, and thermal awareness are built into the language, not bolted on.*
2. ***Heterogeneous computing by default****: Writing GPU/TPU code in Rust requires external crates and significant boilerplate. Joule's* `@kernel` *annotation makes cross-hardware code as natural as writing a function.*

–

3. ***Hardware-aware compilation**: Joule's MLIR backend enables targeting emerging AI accelerators (TPUs, NPUs) that LLVM doesn't support well.*

Rust remains an excellent choice for safety-critical systems. Joule is designed for energy-critical systems where sustainability and hardware efficiency are primary concerns.

---

## 2. Core Language Features

### 2.1 Type System

Joule employs a strong, static type system with full type inference based on Hindley-Milner [2]:

```
// Type inference - the compiler determines types
let count = 42              // Inferred as Int
let name = "Joule"          // Inferred as String
let ratio = 3.14            // Inferred as Float64

// Explicit types when needed
let precise: Float128 = 3.141592653589793238

// Algebraic data types
enum Result<T, E> {
    Ok(T),
    Err(E),
}

// Pattern matching with exhaustiveness checking
match result {
    Ok(value) => process(value),
    Err(error) => handle(error),
}
```

**Key characteristics:** - Types checked at compile time, eliminating runtime type errors - Algebraic data types (enums with associated data) - Pattern matching with exhaustiveness checking - Generic types with trait bounds - No null references—Option types enforce explicit handling

### 2.2 Memory Management

Joule uses an ownership and borrowing system for memory management, building on theoretical foundations established by RustBelt [3]:

–

```
// Ownership - each value has exactly one owner
let data = Vector::new([1, 2, 3, 4, 5])
process(data)        // Ownership transferred to process()
// data is no longer accessible here - compile error if used

// Borrowing - temporary access without ownership transfer
let data = Vector::new([1, 2, 3, 4, 5])
analyze(&data)       // Immutable borrow
data.push(6)         // Still accessible - borrow ended

// Mutable borrowing
let mut data = Vector::new([1, 2, 3, 4, 5])
modify(&mut data)    // Mutable borrow - exclusive access
```

**Benefits:** - **No garbage collector**—deterministic memory management - **No dangling pointers**—compiler prevents use-after-free - **No data races**—the type system prevents concurrent mutation [4] - **Predictable energy profile**—no GC pauses or spikes

Research shows that garbage collection can account for 5–25% of program energy consumption, depending on workload characteristics, heap size, and GC algorithm [5]. Memory-intensive applications with frequent allocations see the highest overhead. By eliminating GC, Joule achieves energy efficiency comparable to manually-managed languages while maintaining memory safety.

### 2.3 Energy Annotations

Joule introduces energy budgets as a language feature:

```
// Specify an energy budget for a function
@energy_budget(100.microjoules)
fn process_sensor_reading(data: &SensorData) -> ProcessedData {
    // Compiler estimates energy bounds based on hardware profile
    // Warnings issued if estimated consumption may exceed budget
    ...
}

// Energy-aware profiling in development
@energy_profile
fn main() {
    // Generates detailed energy consumption report
    ...
}

// Query energy consumption at runtime
let meter = energy::Meter::new()
meter.start()
compute_intensive_task()
let consumed = meter.stop()
```

**Capabilities:** - Compile-time energy bound estimation (using hardware profiles and static analysis heuristics) - Runtime energy measurement via Intel RAPL, ARM Energy Probe [6] - Energy-aware optimization hints to the compiler

**Important caveat:** Static energy estimation is inherently approximate due to hardware variability (cache behavior, DVFS, thermal throttling). Joule provides best-effort bounds rather than formal guarantees, complemented by runtime measurement for validation.

---

## 3. Concurrency Model

### 3.1 Structured Concurrency

Joule provides structured concurrency primitives ensuring concurrent operations complete within well-defined scopes [7]:

–

```
// Parallel iteration - automatic work distribution
let results = data
    .par_iter()
    .map(|item| expensive_computation(item))
    .collect()

// Async/await for I/O-bound operations
async fn fetch_all(urls: &[Url]) -> Vec<Response> {
    let futures = urls.iter().map(|url| fetch(url));
    join_all(futures).await
}

// Explicit task spawning with structured lifetimes
scope(|s| {
    s.spawn(|| task_one());
    s.spawn(|| task_two());
    // Both tasks guaranteed complete before scope exits
})
```

### 3.2 Channels and Message Passing

```
// Create a typed channel
let (sender, receiver) = channel::<Message>()

// Send from one task
sender.send(Message::Data(payload))

// Receive in another
match receiver.recv() {
    Message::Data(payload) => process(payload),
    Message::Shutdown => break,
}
```

**Safety guarantees:** - **No data races**—the ownership system prevents them at compile time - **Deadlock avoidance**—structured concurrency eliminates common patterns - **Energy-efficient synchronization**—primitives designed to enable processor sleep states

---

–

# 4. Heterogeneous Computing

## 4.1 First-Class Hardware Targets

Research shows that specialized processors can be **30–80x more energy-efficient** than CPUs for suitable workloads [8]. Joule treats GPUs, TPUs, and accelerators as first-class targets:

```
// Define a computation portable across backends
@kernel
fn matrix_multiply(a: &Matrix, b: &Matrix) -> Matrix {
    // Implementation compiles for CPU, GPU, and TPU
    parallel_for (i, j) in (0..a.rows, 0..b.cols) {
        result[i][j] = dot(a.row(i), b.col(j))
    }
}

// Explicit placement when needed
@target(GPU)
fn gpu_accelerated_render(scene: &Scene) -> Image {
    ...
}

// Automatic placement based on workload and energy constraints
@auto_place(optimize_for: Energy)
fn flexible_computation(data: &LargeDataset) -> Results {
    // Runtime selects most energy-efficient available hardware
    ...
}
```

### 4.2 Hardware Abstraction

```
// Query available hardware
let devices = Hardware::available_devices()
for device in devices {
    println!("{}: {} compute units, {} memory, {:.2} efficiency",
             device.name,
             device.compute_units,
             device.memory,
             device.energy_efficiency_rating)
}

// Select hardware based on energy efficiency
let target = Hardware::most_efficient_for(WorkloadType::MatrixOps)
```

---

## 5. Compiler Architecture

### 5.1 Tri-Backend Design

Joule's compiler supports three code generation backends to balance development speed, production optimization, and emerging hardware support:

| Backend | Primary Use Case | Characteristics |
|---|---|---|
| **Cranelift** | Development | Fast compilation (~10x faster than LLVM), quick iteration |
| **LLVM** | Production | Maximum optimization, broad platform support [9] |
| **MLIR** | Emerging Hardware | AI accelerators, custom silicon, TPUs [10] |

```
# Development build (fast compilation)
joule build --backend cranelift

# Release build (maximum optimization)
joule build --release --backend llvm

# Target specific AI hardware
joule build --backend mlir --target tpu-v4
```

–

### 5.2 Energy-Aware Optimization

The compiler performs optimizations specifically targeting energy consumption [11]:

| Optimization | Energy Impact | Mechanism |
|---|---|---|
| Memory traffic minimization | ~100x per DRAM vs cache access | Loop tiling, data layout |
| SIMD vectorization | 2-8x throughput/watt | Automatic vectorization |
| Power state enablement | Variable | Avoid busy-waits, enable sleep |
| Code density | 10-20% cache efficiency | Instruction selection, inlining |

```
# Standard optimization levels
joule build -O0    # No optimization (debugging)
joule build -O2    # Full optimization
joule build -O3    # Aggressive optimization

# Energy-focused optimization
joule build -Oe    # Optimize for energy efficiency
joule build -Oez   # Energy + code size optimization
```

---

–

# 6. Standard Library

## 6.1 Core Modules

| Module | Purpose |
|---|---|
| `core` | Fundamental types, traits, primitives |
| `collections` | Data structures (Vec, Map, Set, etc.) |
| `io` | Input/output operations |
| `net` | Networking primitives |
| `async` | Asynchronous runtime |
| `compute` | Heterogeneous computing abstractions |
| `energy` | Energy measurement and budgeting |

—

### 6.2 Energy Module

```
use energy::{Budget, Meter, Profile}

// Measure energy consumption
let meter = Meter::new()
meter.start()
perform_computation()
let consumption = meter.stop()
println!("Energy used: {} microjoules", consumption.microjoules())

// Check against budget
let budget = Budget::new(500.microjoules)
if consumption > budget {
    log::warn!("Exceeded energy budget by {}", consumption - budget)
}

// Profile a code section
let profile = Profile::measure(|| {
    complex_algorithm()
})
profile.print_report()  // Shows energy breakdown by function
```

---

## 7. Interoperability

### 7.1 C Foreign Function Interface

Joule provides zero-overhead C interoperability:

—

```
// Declare external C functions
extern "C" {
    fn legacy_algorithm(data: *const u8, len: usize) -> i32
}

// Export Joule functions for C consumption
@export
fn joule_entry_point(input: *const u8, len: usize) -> i32 {
    // Safe Joule code wrapping unsafe FFI boundary
    let slice = unsafe { slice::from_raw_parts(input, len) }
    process(slice)
}
```

### 7.2 WebAssembly Support

```
# Compile to WebAssembly
joule build --target wasm32-unknown-unknown

# With WASI support for system interfaces
joule build --target wasm32-wasi
```

---

## 8. Platform Support

### Tier 1 (Full support, CI-tested)

- Linux x86_64, aarch64
- macOS x86_64, aarch64 (Apple Silicon)
- Windows x86_64

### Tier 2 (Builds, community-tested)

- FreeBSD x86_64
- WebAssembly (wasm32)

### Tier 3 (Experimental)

- RISC-V (rv64gc)

—

- Embedded ARM (Cortex-M)

---

## 9. Getting Started

### Installation

```
# Unix-like systems
curl -sSf https://joule-lang.org/install.sh | sh

# Verify installation
joule --version
```

### Hello World

```
// main.joule
fn main() {
    println!("Hello, sustainable computing!")
}
```

```
joule run main.joule
```

---

## References

[1] Pereira, R., Couto, M., Ribeiro, F., et al. “Energy Efficiency across Programming Languages.” *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, 2017. https://joule.openie.dev/research/energy-efficiency-languages

[2] Milner, R. “A Theory of Type Polymorphism in Programming.” *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348-375, 1978. https://doi.org/10.1016/0022-0000(78)90014-4

[3] Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D. “RustBelt: Securing the Foundations of the Rust Programming Language.” *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, 2018. https://doi.org/10.1145/3158154

[4] Boyapati, C., Lee, R., Rinard, M. “Ownership Types for Safe Programming: Preventing Data Races and Deadlocks.” *Proceedings of OOPSLA*, 2002. https://doi.org/10.1145/582419.582440

[5] Pinto, G., Castor, F., Liu, Y.D. "Mining Questions About Software Energy Consumption." *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, 2014. https://doi.org/10.1145/2597073.2597110

[6] Khan, K.N., Hirki, M., Niemi, T., et al. "RAPL in Action: Experiences in Using RAPL for Power Measurements." *ACM TOMPECS*, vol. 3, no. 2, 2018. https://doi.org/10.1145/3177754

[7] Sutter, H. "Structured Concurrency." *ISO/IEC JTC1/SC22/WG21*, 2021. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2300r0.html

[8] Jouppi, N.P., et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit." *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017. https://doi.org/10.1145/3079856.3080246

[9] Lattner, C., Adve, V. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." *Proceedings of CGO*, 2004. https://doi.org/10.1109/CGO.2004.1281665

[10] Lattner, C., et al. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation." *Proceedings of CGO*, 2021. https://doi.org/10.1109/CGO51591.2021.9370308

[11] Schulte, E., Dorn, J., Harding, S., et al. "Post-compiler Software Optimization for Reducing Energy." *Proceedings of ASPLOS*, 2014. https://doi.org/10.1145/2541940.2541980

[12] Charlot, D.J. "Bounded Entropy in Formal Languages: A Mathematical Foundation for Deterministic Code Generation." OpenIE Technical Report, December 2025.

[13] Charlot, D.J. "Cortex: Neural-Symbolic Programming for Energy-Efficient Code Execution." OpenIE Technical Report, January 2026.

[14] Charlot, D.J. "Metabolic Cascade Inference: Hardware-Aware Adaptive Routing for Energy-Efficient AI." OpenIE Technical Report, January 2026.

*[15] Charlot, D.J. "Cortex: The AI-First Programming Language Based on Selective State Spaces." OpenIE Technical Report, February 2026.*

---



***Contact**: david@openie.dev | dcharlot@ucsb.edu **Latest Updates**: https://openie.dev/projects/joule*

—