JIT Compilation and Dynamically Typed Languages

TJ Barclay

University of Kansas

April 26, 2019

Outline

Dynamic Programming Languages

Dynamic Programming Languages - Introduction

Static

"Stuff" happens at/before compile time

Dynamic

"Stuff" happens at runtime

"Stuff" includes:

- Method binding
- Typing
- Program extension
- Modifying objects/classes

Dynamic Typing vs Static Typing

Static typing: types are checked before runtime, for example Java checks while compiling to bytecode
Dynamic typing: types are checked during runtime, Julia/Actionscript

Optional Typing

Dynamic typing but the programmer can force the type of a variable to be something

JIT Compilation

What is JIT compilation?

Just-In-Time (JIT) compilation is compilation to machine code that happens at runtime.

JIT Tradeoffs

Compilation speed vs. generating performant code

Compiler Optimizations

High-Level Optimization

High-Level Optimization

Optimization that requires knowledge about the language semantics and runtime environment

Examples:

- Type inference
- Method inlining
- Type speculation
- Method specialization
- Object unboxing

Low Level Optimization

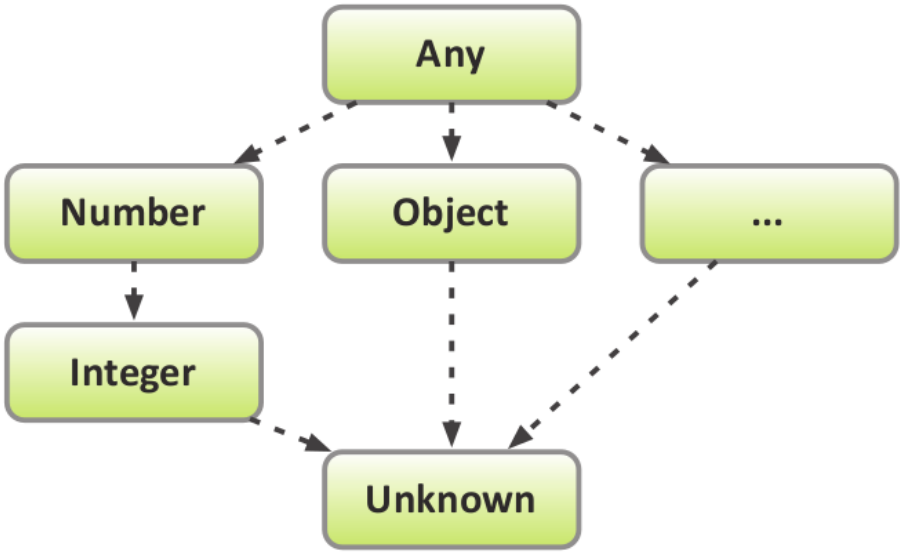
Low-Level Optimization

Optimization that happens in any context, simply by observing the structure of low-level IR

Examples:

- Redundant load/store removal
- Common subexpression elimination
- Dead code elimination
- Register allocation

Type Inference



- Iterative data flow problem
- Start from the most specific type and generalize (contrast with Hindley-Milner)
- *Example in Julia*

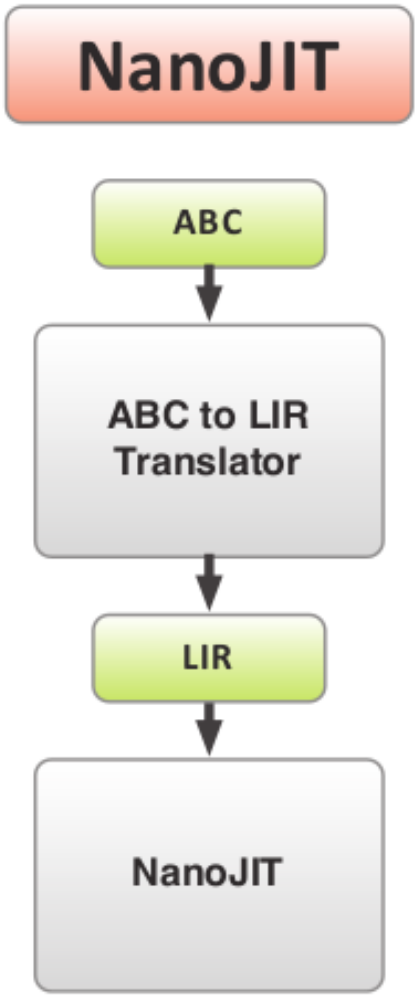
Examples

ActionScript

- Dynamic programming language
- Optionally typed
  - Programmer can specify type of variable or it has `Any` type
- Tamarin VM
  - NanoJIT
  - Type Enriched Static Single Assignment (TESSA)

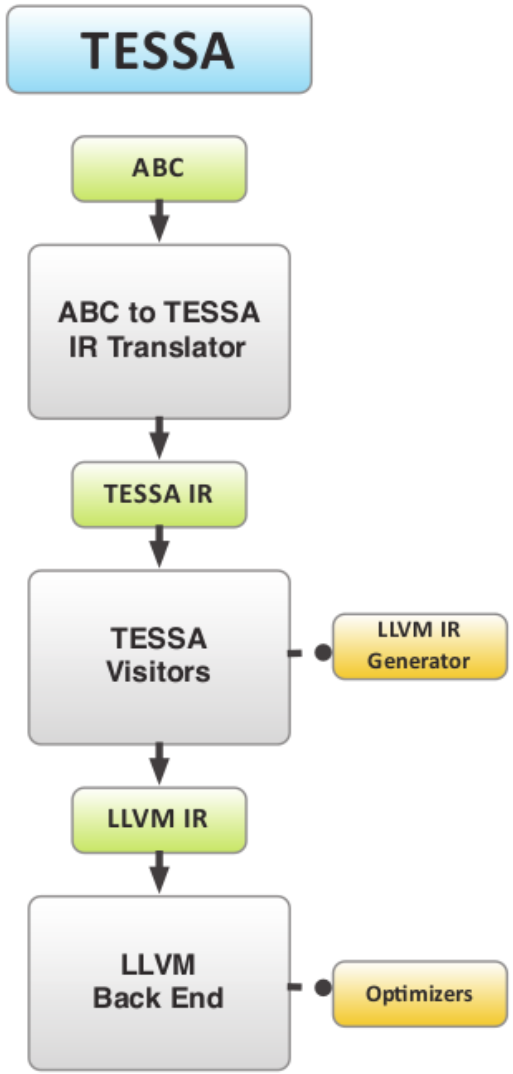
NanoJIT

- Designed for fast compilation
- ActionScript Bytecode (ABC)
- Few optimizations
  - Common subexpression elimination
  - Redundant load/store removal
- Untyped variables given the `Any` Type
  - requires C++ conversion code to be inlined



TESSA

- Designed to produce faster code
- Performs heavier optimizations
  - Type inference
  - Method inlining
  - LLVM low-level optimizations

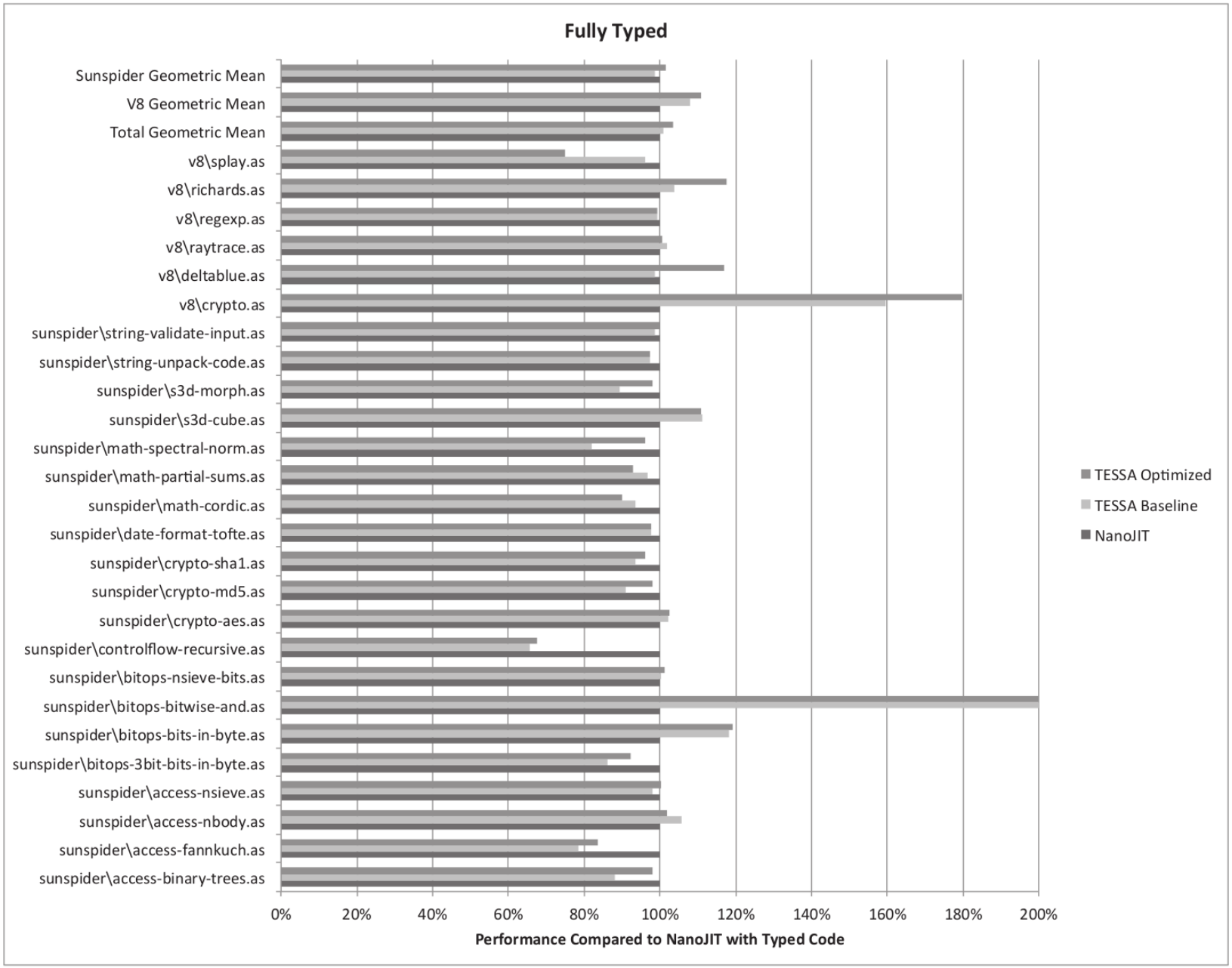


Evaluation

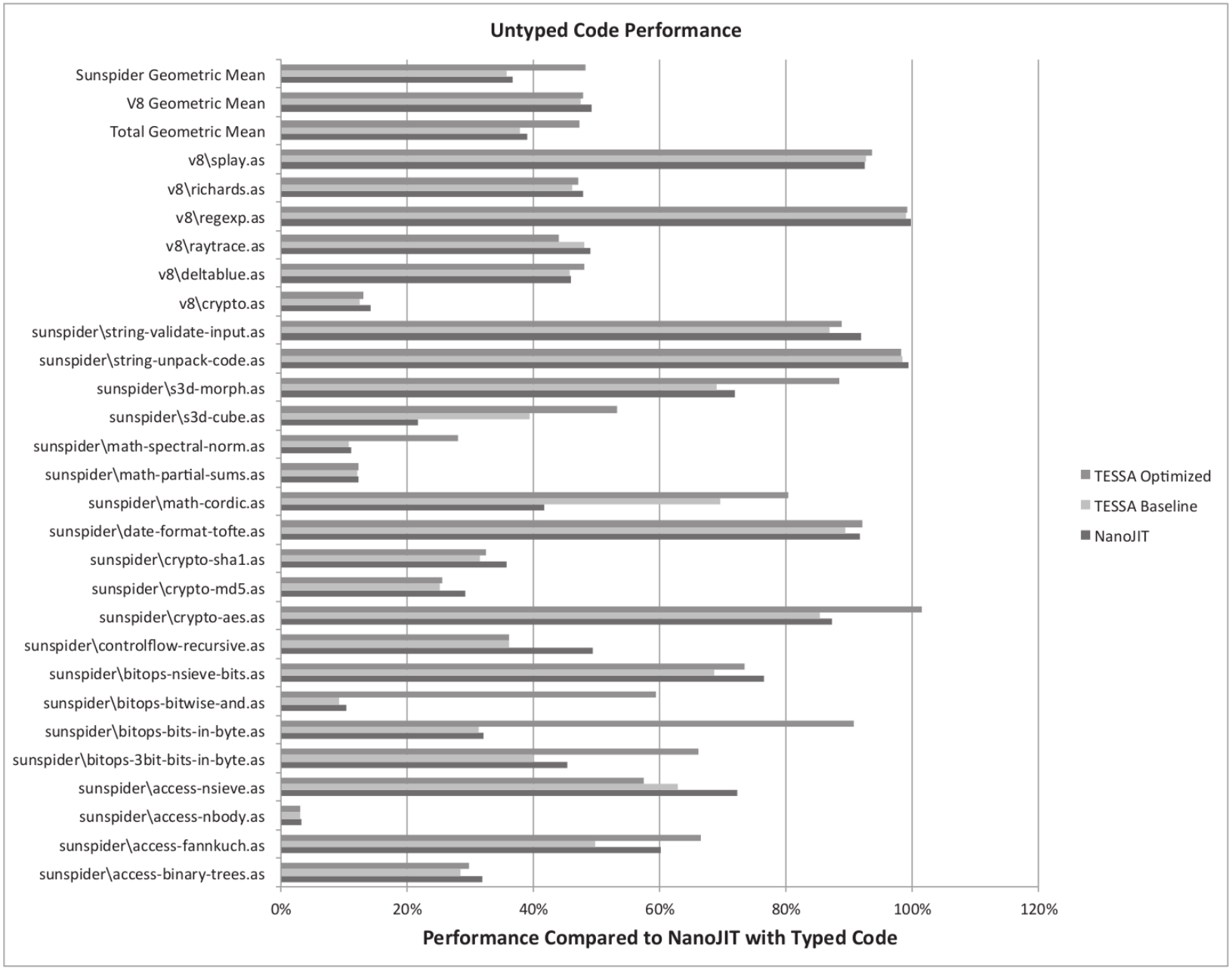
Comparing:

- Generated code performance
  - Differing amounts of type information
  - Different backend optimization levels
- JIT compilation time

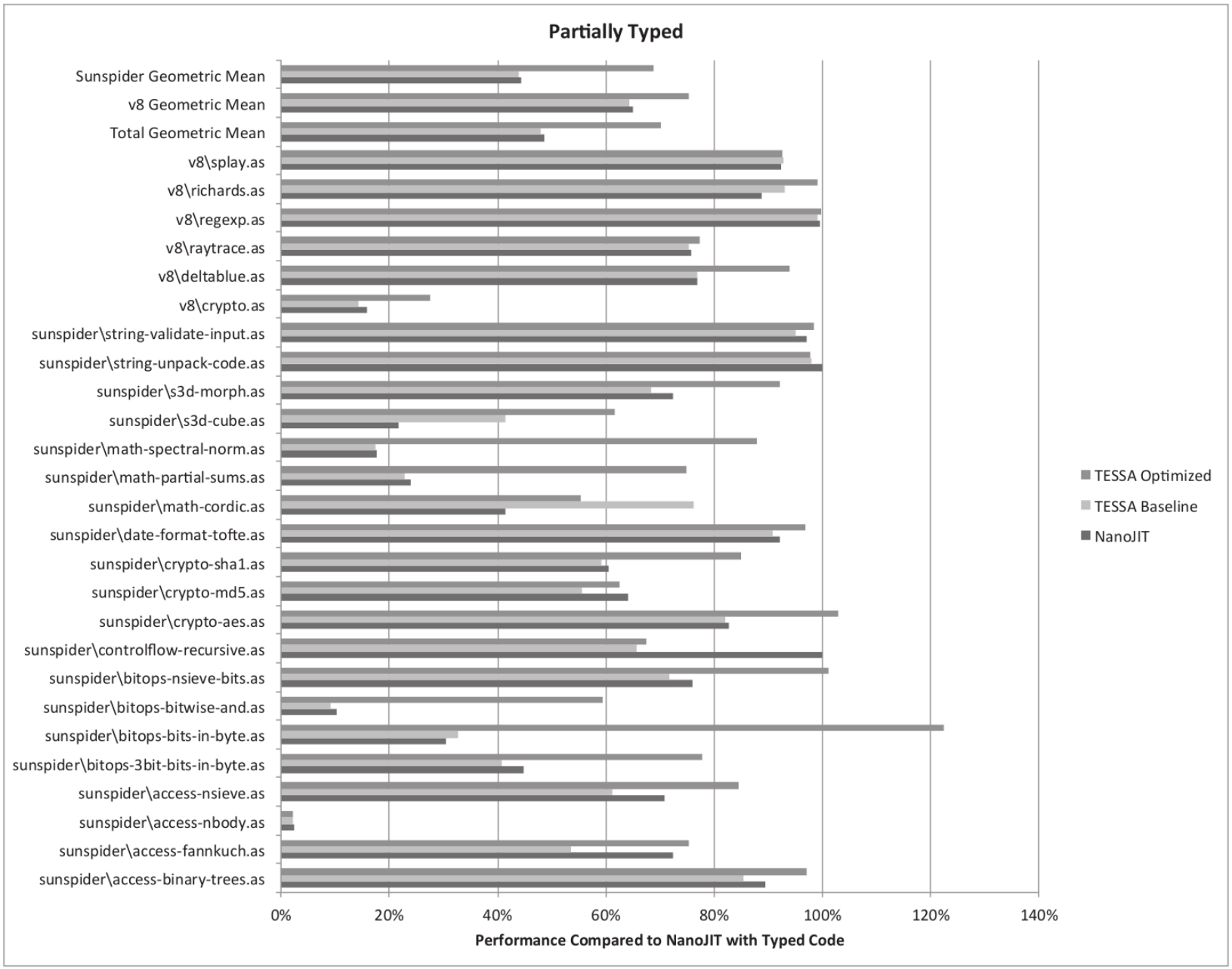
Typed Code


Fully Typed
Sunspider Geometric Mean
V8 Geometric Mean
Total Geometric Mean
v8\splay.as
v8\richards.as
v8\regexp.as
v8\raytrace.as
v8\deltablue.as
v8\crypto.as
sunspider\string-validate-input.as
sunspider\string-unpack-code.as
sunspider\s3d-morph.as
sunspider\s3d-cube.as
sunspider\math-spectral-norm.as
sunspider\math-partial-sums.as
sunspider\math-cordic.as
sunspider\date-format-tofte.as
sunspider\crypto-sha1.as
sunspider\crypto-md5.as
sunspider\crypto-aes.as
sunspider\controlflow-recursive.as
sunspider\bitops-nsieve-bits.as
sunspider\bitops-bitwise-and.as
sunspider\bitops-bits-in-byte.as
sunspider\bitops-3bit-bits-in-byte.as
sunspider\access-nsieve.as
sunspider\access-nbody.as
sunspider\access-fannkuch.as
sunspider\access-binary-trees.as
0%
20%
40%
60%
80%
100%
120%
140%
160%
180%
200%
Performance Compared to NanoJIT with Typed Code
TESSA Optimized
TESSA Baseline
NanoJIT

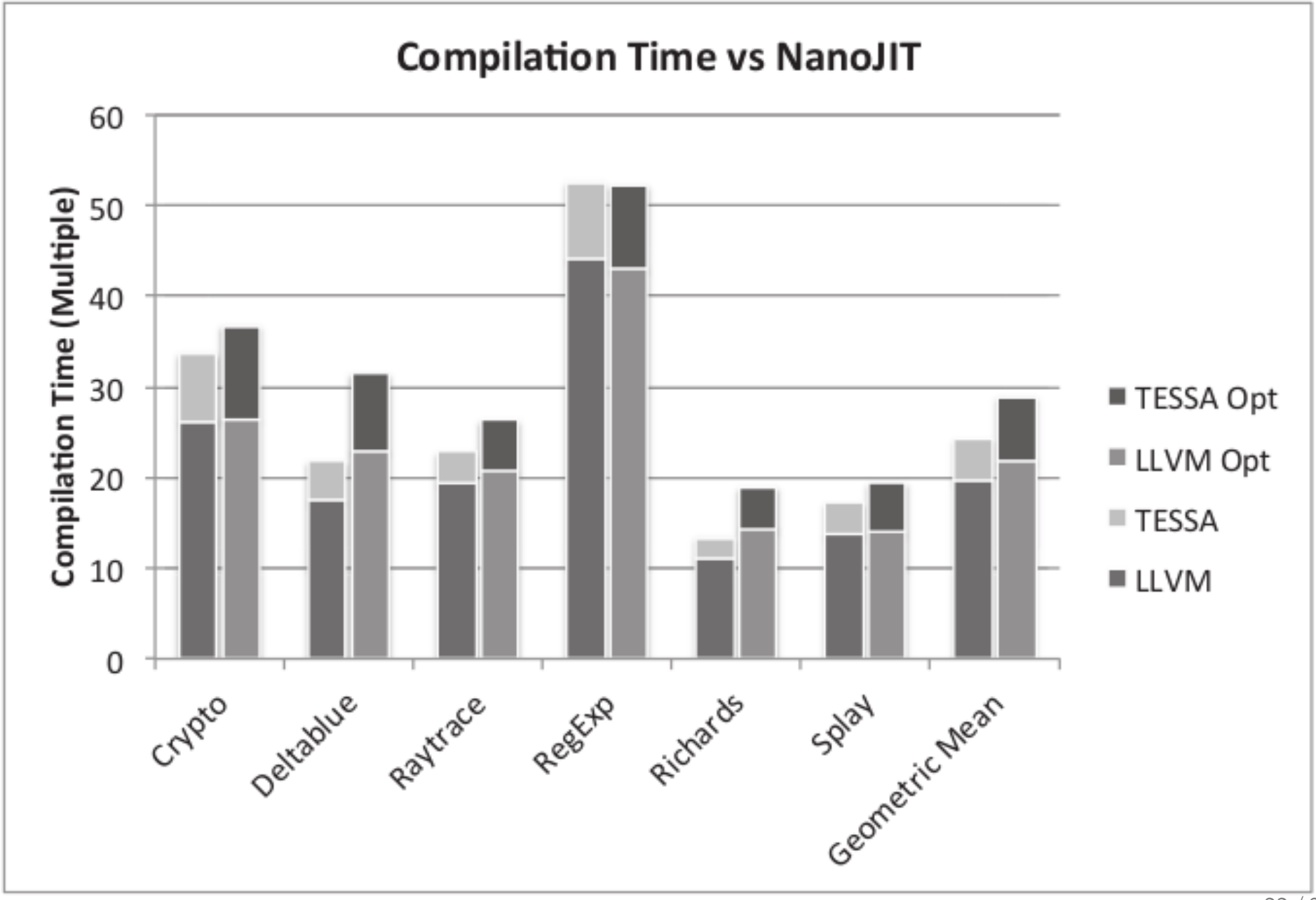
Untyped Code



Partially Typed Code



Compilation Time



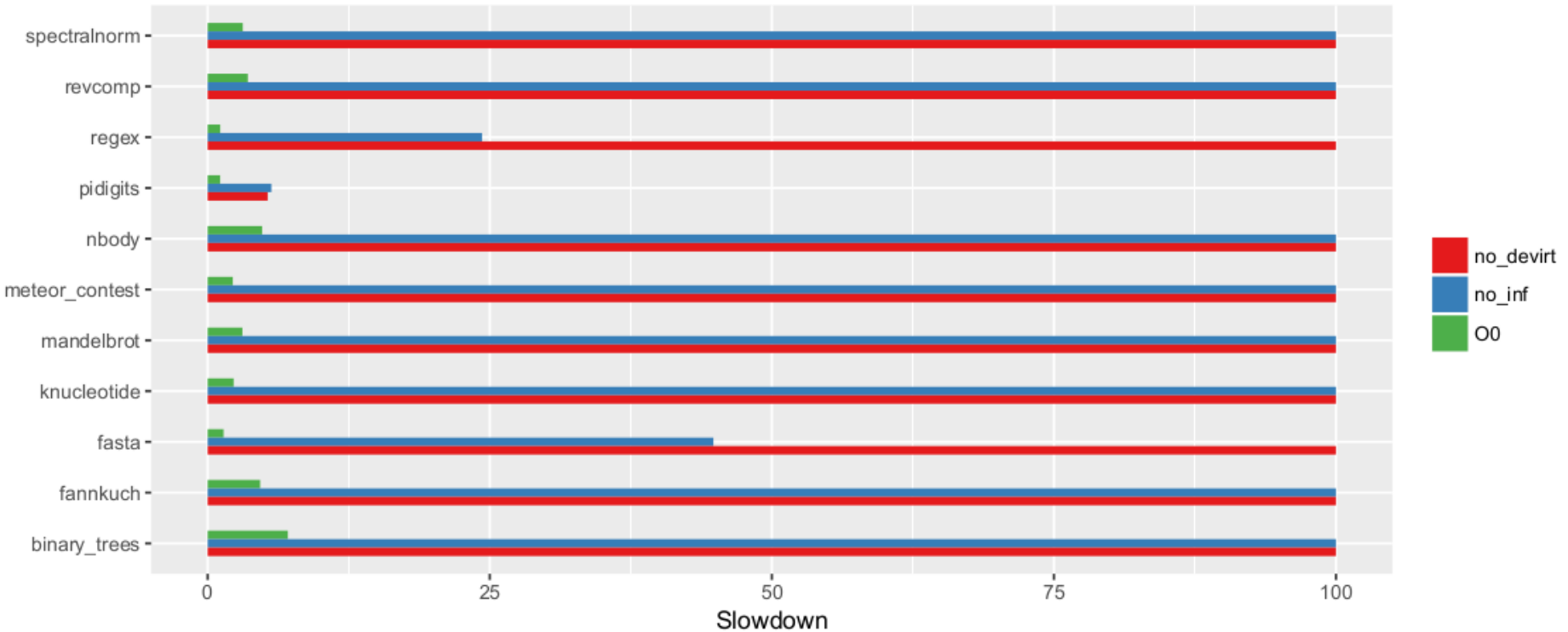
Julia

- Dynamic programming language
- Optionally typed
- Multiple dispatch
- Designed for fast development that can be later sped up
- Can control memory layout of datatypes

Julia's Optimizations

- Method specialization
- Type inference
- Method inlining (In Julia methods are function implementations that are ad hoc polymorphic)
- Object unboxing

Evaluation


spectralnorm
revcomp
regex
pidigits
nbody
meteor_contest
mandelbrot
knucleotide
fasta
fannkuch
binary_trees
0
25
50
75
100
Slowdown
no_devirt
no_inf
O0

Conclusions

Conclusions:

- High-level optimizations are key to performance gains
- Large amounts of low-level optimization often takes too long to justify the speedup

References

Mason Chang, Bernd Mathiske, Edwin Smith, Avik Chaudhuri, Andreas Gal, Michael Bebenita, Christian Wimmer, and Michael Franz.
2011. The impact of optional type information on jit compilation of dynamically typed languages.
SIGPLAN Not. 47, 2 (October 2011), 13-24. DOI: https://doi.org/10.1145/2168696.2047853

Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky.
2018. Julia: dynamism and performance reconciled by design.
Proc. ACM Program. Lang. 2, OOPSLA, Article 120 (October 2018), 23 pages. DOI: https://doi.org/10.1145/3276490

Questions?