ASTro による JIT コンパイラの試作

STORES 株式会社
笹田 耕一

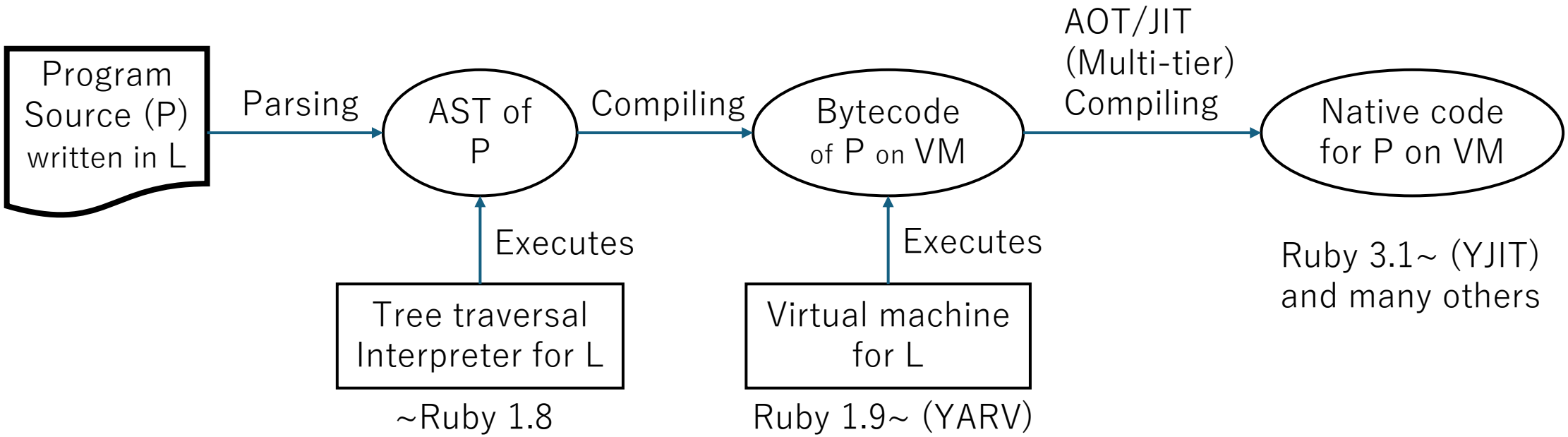
この発表は？

AST: Abstract Syntax Tree
JIT: Just in time compilation
AOT: Ahead of time compilation
PGO: Profile guided optimization

- 先行研究：ASTro: AST reusable optimization framework [10]
  - 1) 部分評価によって、与えられたASTから**高性能コードを生成**
    - PE(Tree walking インタプリタ定義, AST）→ C コード生成
  - 2) AST（部分木）にハッシュ値をつけることで、生成コードを共有
  - 課題: AOT/PGO のみでいい評価 → JIT できるか不明
- 本研究：JITしてみた
  - 実行時にC compiler (CC) を動かす仕組み
  - コンパイル済みコードを再利用する仕組み

[10] Koichi Sasada: ASTro: An AST-Based Reusable Optimization Framework, VMIL ‘25, pp. 22-32 (2025).

インタプリタ開発のつらさ

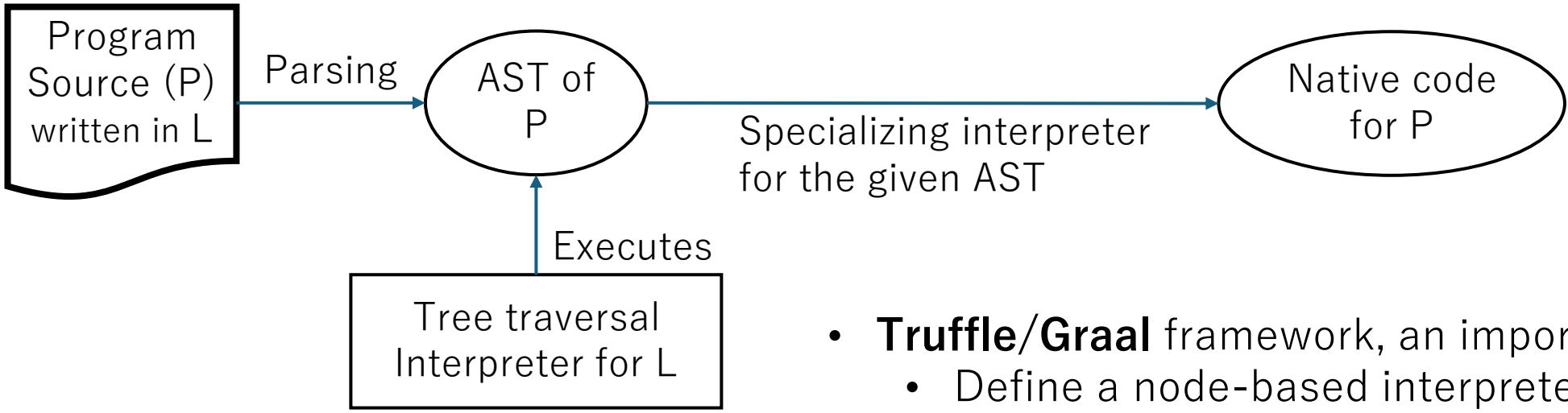


✅ Simple, Easy
❌ Slow

✅ Fast
❌ Complicated

分業が進むと、前段を変更しにくい

部分評価による高速なインタプリタ



✅ Simple, Easy and Fast!!
❌ Heavyweight Framework
Hard to make and modify

- **Truffle/Graal** framework, an important system:
  - Define a node-based interpreter in **Truffle SDL**
  - Profile given node execution
  - Specialize (PE) the given AST and interpreter using **Truffle framework** and generate native code with **Graal compiler**

もっと簡単に作れないか/使えないか？

ASTroの提案
Cコンパイラを利用する部分評価器

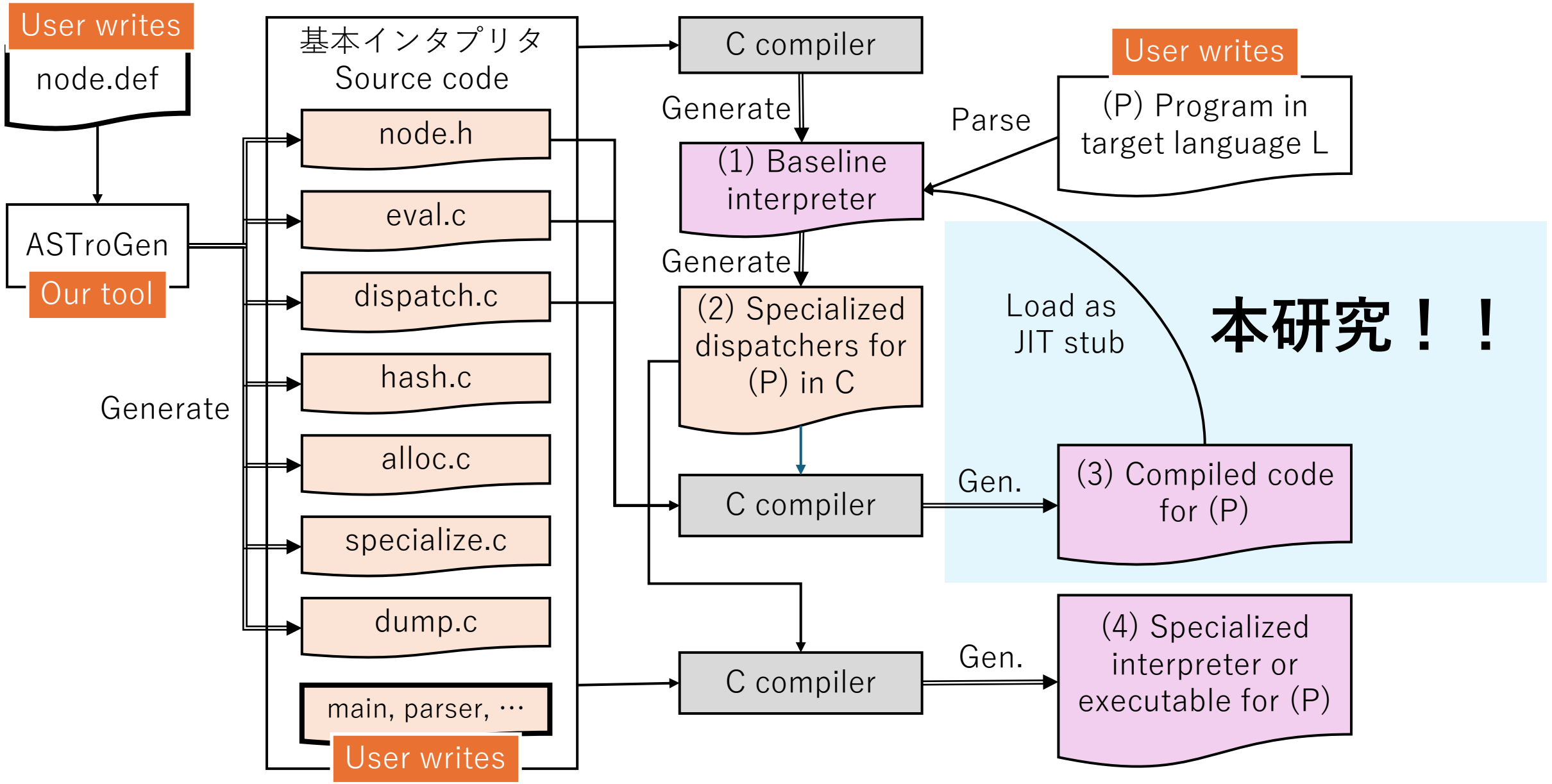
- ASTro: AST-based reusable optimization framework
  - **ASTroGen** (~500 lines in Ruby) がベースラインインタプリタ生成
    - ユーザーは各ノードの挙動を定義
    - ASTのノードを辿るインタプリタ（簡単）を生成
  - ASTを受け取り、特化したものを生成する部分評価器も生成

    アイディア：特化した起動関数（小さいCコード片）のみ生成

    部分評価のために定義を見る必要なしで**作成が容易**
  - どこにでもあるCコンパイラが再利用可能なネイティブコードを生成

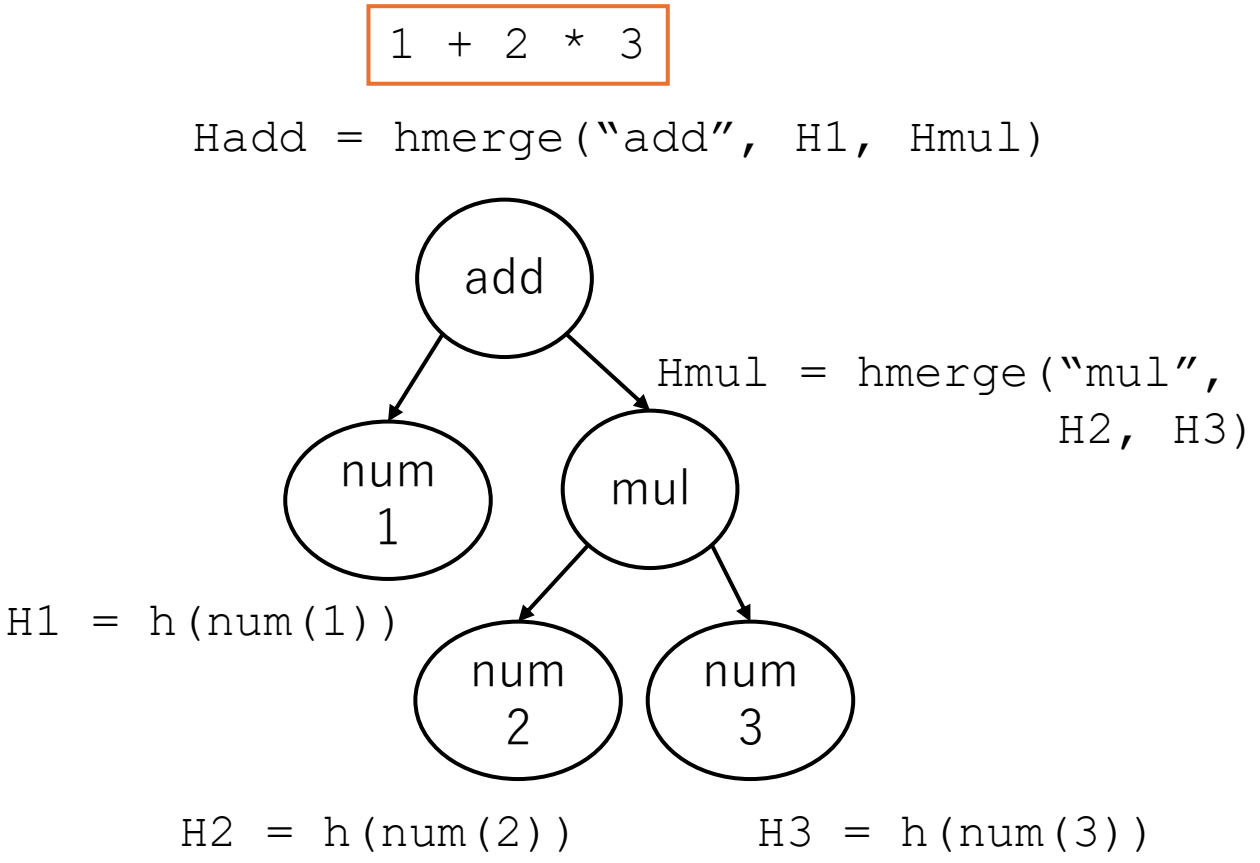
    アイディア：AST（部分木）にマークル木で名前付け

About this talk



アイディア1: AST（部分木）にハッシュ値で名づけ

- マークル木
  - 子ノードの値でつけるハッシュ値　分散計算でよくやる
- マークル木として AST（部分木）に名づけることで、いろんな最適化をインタプリタプロセスを超えて共有可能



未解決：
　衝突は？
　適したアルゴリズムは？

H… is something like "fb75fdabb0d5ef6" in practice

アイディア2
ディスパッチ関数のみを生成する部分評価

- 部分評価は、AST部分木に対するディスパッチ関数のみ生成
  - ノード評価部を一切触らないため部分評価器開発が容易
  - C コンパイラによるインライン展開で最適コード生成

```
// specialized dispatcher for num(2)
SD_<H2>(CTX *c, NODE *n) {
  return EVAL_node_num(c, n, 2);
}
```

<H2> はノードのハッシュ値

Generate specialized code

num
2

H2 = h(num(2))

ASTroGen generates

```
// specialize.c

static void
SPECIALIZE_node_num(FILE *fp, NODE *n)
{
  /* name is SD_<hash> */
  const char *name = alloc_dispatcher_name(n);
  n->head.dispatcher_name = name;
  fprintf(fp, "static VALUE¥n");
  fprintf(fp, "%s(CTX *c, NODE *n)¥n{¥n", dname);
  fprintf(fp, " return EVAL_node_num(c, n, %d);¥n",
          n->u.node_num.num);
  fprintf(fp, "}¥n");
}
```

具体例
Tree Traversal Interpreter w/ switch/case

```
int eval1(CTX *c, NODE *n){ // c は実行コンテキスト
  switch(n->type) {
    case NODE_NUM:
      return n->value;
    case NODE_ADD:
      return eval(n->lv) + eval(n->rv);
    case NODE_MUL:
      return eval(n->lv) + eval(n->rv);
    case NODE_LSET:
      return lset(c, n->name, eval(n->rhs));
```

太字の部分が各ノードの挙動。
ボディ部というようにします。

eval2
Tree Traversal Interpreter w/ func ptr

```
int eval2(CTX *c, NODE *n){
    return (*n->eval)(c, n);
}
```

各ノードは eval 関数へのポインタをもつ。
ちょっと見やすくなった？
速度はむしろ遅くなりそう

```
int eval_num(CTX *c, NODE *n) {
    // さっきの case の中身（ボディ部）を
    // 関数にくくりだしただけ
    return n->value;
}
```

…

eval3
Tree Traversal Interpreter w/ dispatch func

```
int eval3(CTX *c, NODE *n){
  return (*n->dispatcher)(c, n);
}

int dispatch_num(CTX *c, NODE *n) {
  // さっきの eval_num を、ディスパッチ関数と分離
  return eval_num(c, n, n->value);
}

int eval_num(CTX *c, NODE *n, int value) {
  return value;
}
```

…

**ASTroGen が作るベースライン
インタプリタ。
各ノードがディスパッチ関数への
関数ポインタをもち、それを実行**

ユーザーはこの部分を書く

起動関数のみを生成する簡易部分評価 実例

例: `1 + 2 * 3 → add(num(1), mul(num(2), num(3)))`

```
// 特殊化されたディスパッチ関数だけ生成
SD_H<num(1)>(CTX *c, NODE *n) {
  return eval_num(c, n, 1); // 1 に特殊化
}

// 2, 3 も同様に生成
```

eval_num() の実装は `return value` だけなので、
つまりこの関数は `return 1` だけにインライン展開される

起動関数のみを生成する簡易部分評価 実例

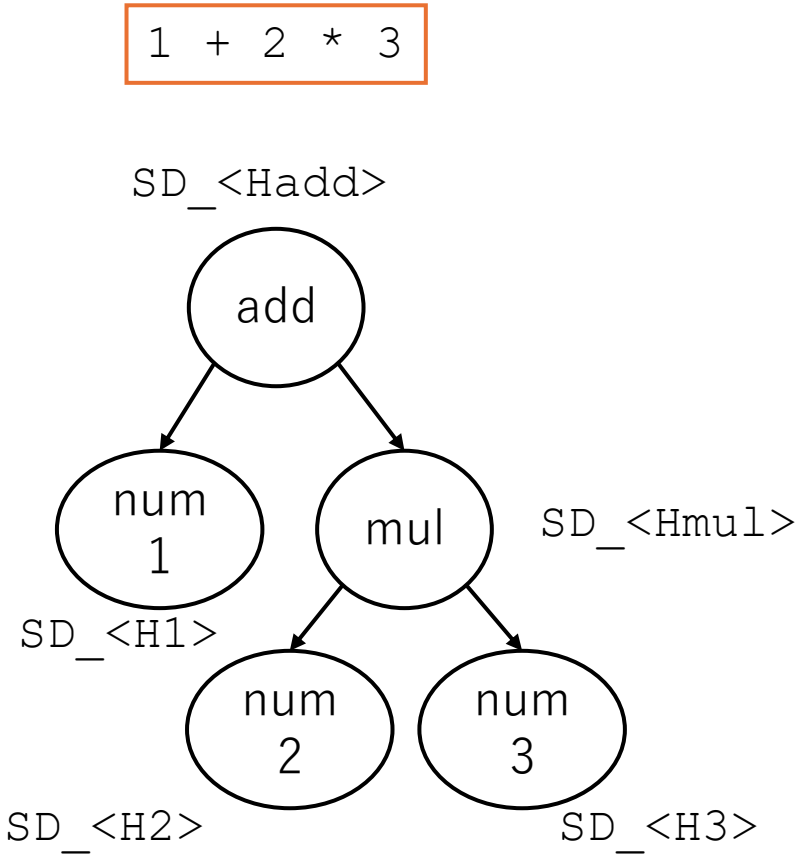
例: `1 + 2 * 3 → add(num(1), mul(num(2), num(3)))`

```
// 子ノードの呼び出し先を埋め込む
SD_H<mul_2_3>(CTX *c, NODE *n) {
  return eval_mul(c, n, n->lv,
                        dispatch_H<num_2>,
                        n->rv,
                        dispatch_H<num_3>);
}
```

起動関数のみを生成する簡易部分評価実例

- **ディスパッチ関数だけ**を作ることで、めでたく最適化されたネイティブコードを生成可能
  - num/add の定義を処理系は一切見ないので簡単
  - インライン化してくれるから高速なコード
  - ハッシュ値で名前付け→再利用可能
    - AOT, PGO, JIT（まだ）

```
0000000000001a20 <SD_<Hadd>>:
1a20: endbr64
1a24: mov $0x7,%eax ; literal 7
1a29: ret
```



この研究に必要な情報のまとめ

- プログラム（AST）を入力すると、
  それに対する**ユニークな名前（ハッシュ値）**のついた
  プログラムを高速に実行する **C プログラム**を生成

- 先行研究は、これを使って AOT, PGO 処理系を開発

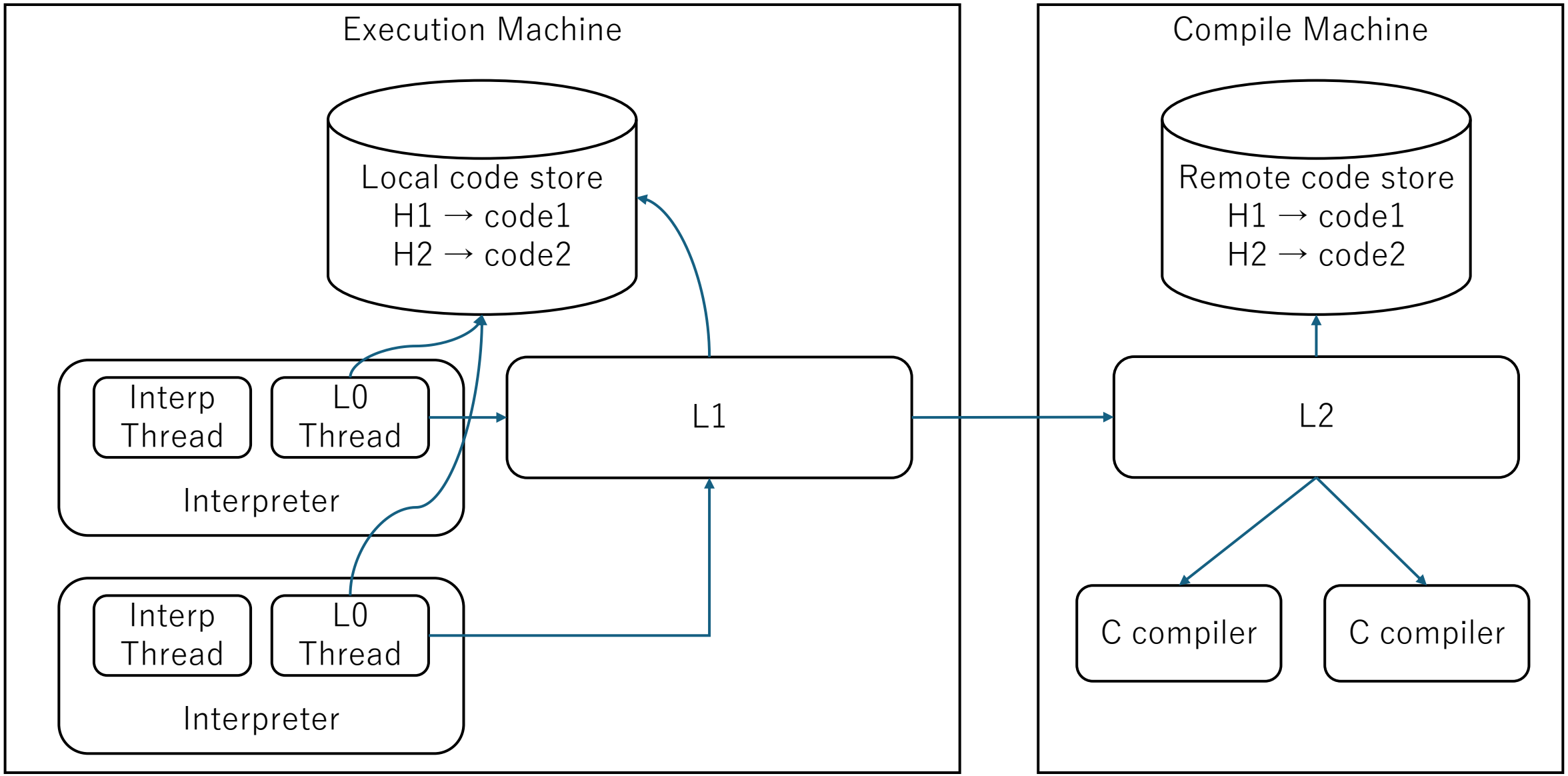
本研究：ASTro の仕組みでJIT？

- 素直な疑問
  - インタプリタ生成系といいながらJITもできないなんて
  - C のコードを出すなら、C compiler を実行時に呼べば JIT できるんでは？
    → **でも、C コンパイラ呼ぶと遅いよなあ？**
  - コンパイル済みコードはハッシュ値で名前付いてるので再利用可能
    → **でも、具体的にどうやって？**

提案: ASTro JIT

- 実行ノードとコンパイルノードをわけた構成
  - L0: インタプリタにつく、JIT用ネイティブスレッド L0
  - L1: ローカルノード
  - L2: 共有コンパイルノード
- ローカル・グローバルなコンパイル済みコードストア
- 試作
  - L0 は C、L1, L2 は Ruby で実装
  - コンパイル済みノードは Shared Object (.so)

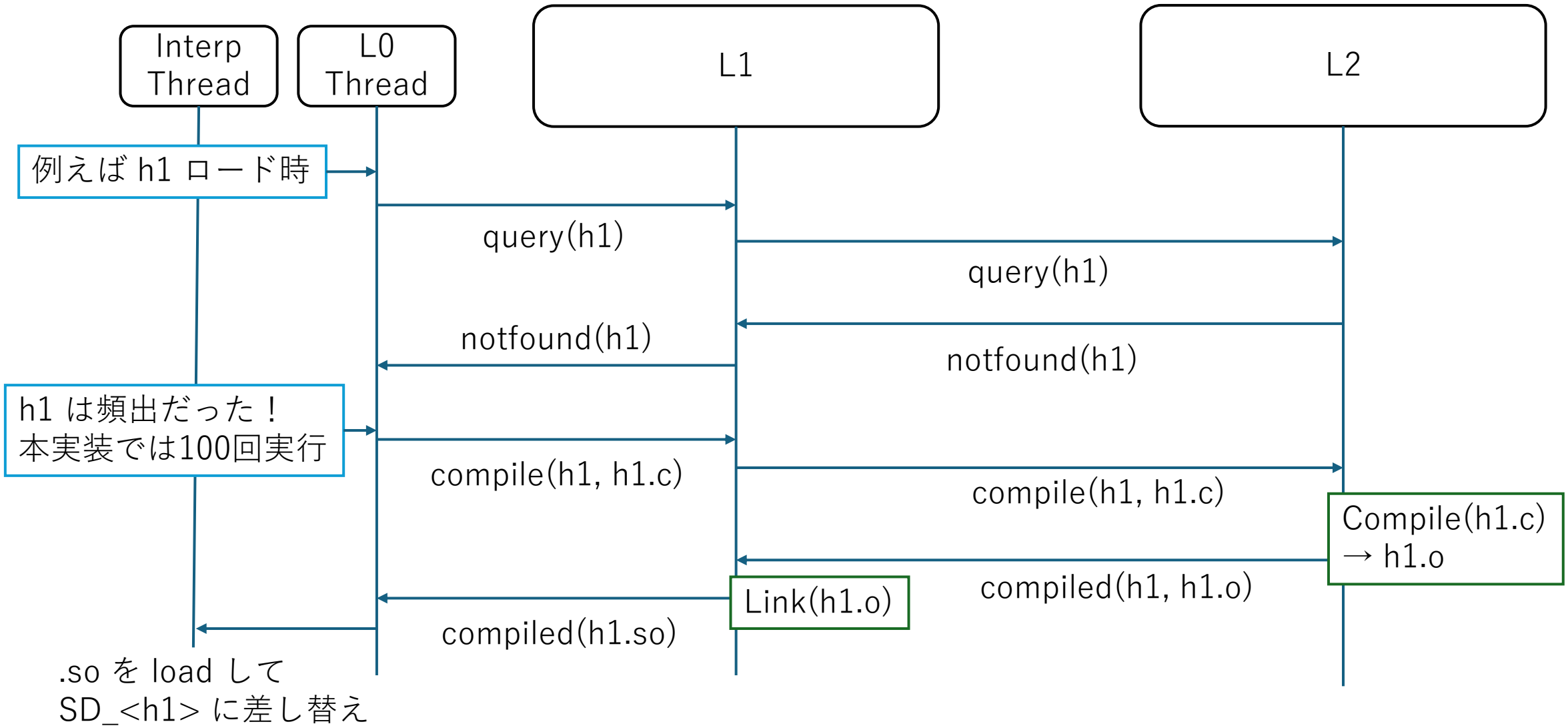
ASTro JIT 構成


Execution Machine
Local code store
H1 → code1
H2 → code2
Interp Thread
L0 Thread
Interpreter
Interp Thread
L0 Thread
Interpreter
L1
Compile Machine
Remote code store
H1 → code1
H2 → code2
L2
C compiler
C compiler

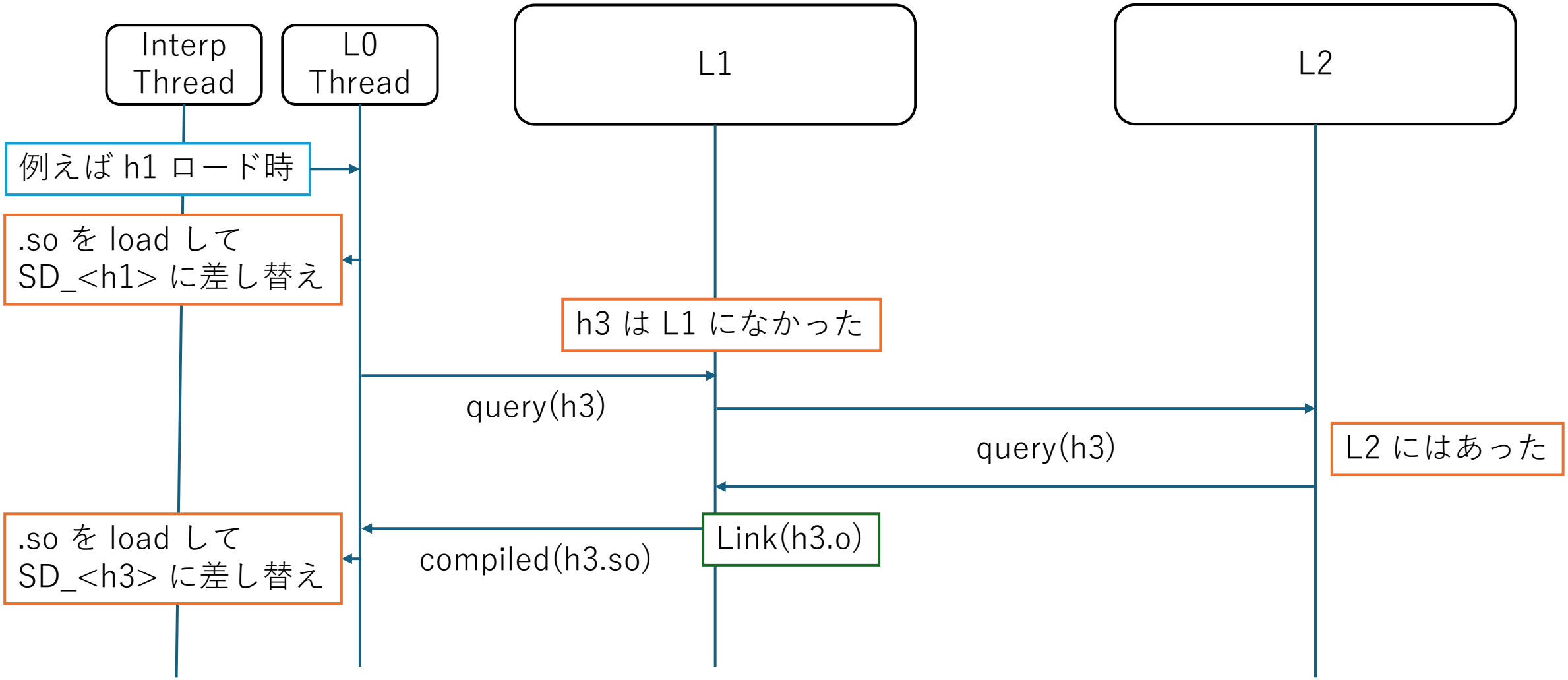
この構成の利害得失

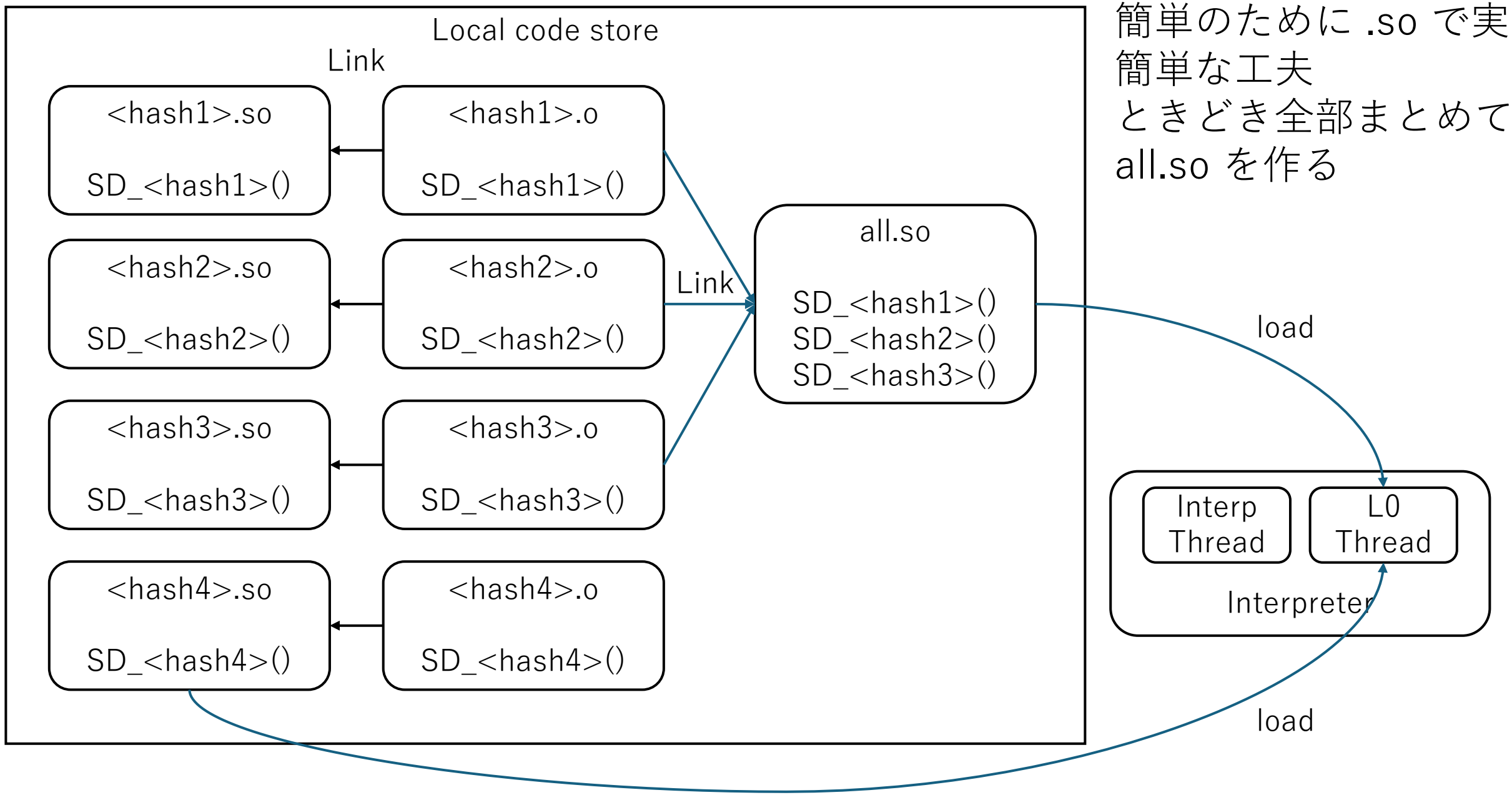
- 良い点
  - 同じプログラム（同じ AST 部分木）を共有するプログラムは、コンパイル済みコードを共有可能
    - 同じプログラムを複数マシンで実行、などの場合、コンパイルは1か所に集約し、無駄を省ける（C Compiler でも許されそう）
    - 2回目実行時にはそのままコンパイル済みコードを再利用可能
  - Cコンパイルを L2 マシンに移譲できる
    - 強い L2 マシンを作ったり、L2 をさらに分散したりできる
  - 事前コンパイル（AOT, PG）のみの構成にフォールバックも容易
- 悪い点
  - L1, L2 デーモンプロセスを用意するのが大変
    - たとえば、Ruby をカジュアルに使う人がこれをやるんか？

ASTro JIT Protocol なかったとき



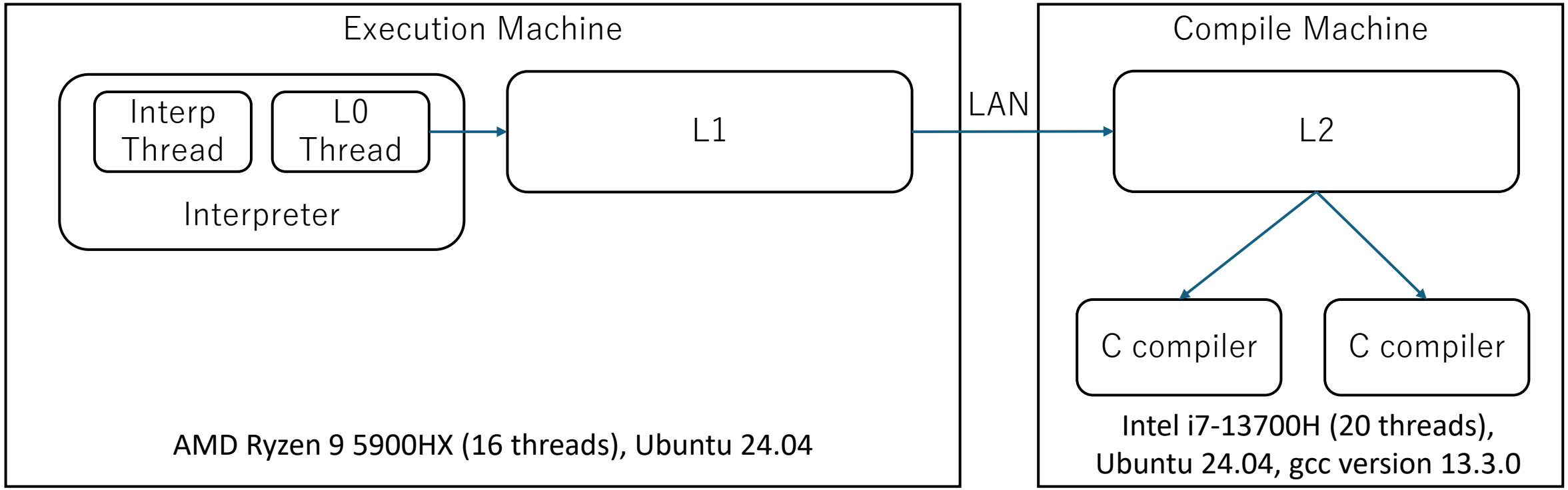
ASTro JIT Protocol
コンパイル済みの時





簡単のために .so で実装
簡単な工夫
ときどき全部まとめて
all.so を作る

評価　かんたんな予備調査



性能評価（実行時間）

- 簡単なプログラムで評価
  - naruby: Ruby 文法で C っぽい言語[10]
    - 整数演算のみ
  - 簡単な素数判定の繰り返し
- 1回目からだいぶ速い
- 2回目はちょっと速い
  - コンパイル済みコードを利用
  - この差がコンパイル時間
- 大きいサイズだと変わりそう

```
# ナイーブな素数判定
def prime?(n)
  if n < 2
    0
  else
    prime = 1
    i = 2
    while i * i <= n
      prime = 0 if n%i == 0
      i += 1
    end
    prime
  end
end
```

```
i = 0
while i < 1_000_000
  prime?(i)
  i += 1
end
```

	JIT なし	JIT あり
1 回目の実行	13.64	1.11
2 回目の実行	上に同じ	1.05

表 1. JIT の有無による実行時間（秒）

性能評価（サイズ）

- 小さなメソッドを1000個
  → 1000個の約14KB の .so を作成
  （各機械語は10〜12バイト）
- all.so は 120KB でだいぶ小さい

```
n = 1_000_000
def f0(a) = a + 0
def f1(a) = a + 1
...
def f999(a) = a + 999

i=0
while i<n
  f0(i); ...; f999(i)
  i += 1
end
```

性能評価（時間その２）

- 実行してみたらとても遅い (b)
  - (a) .so を 1000 個ページアラインメントで配置するため、CPUキャッシュミス多発
    → 独自バイナリローダが必要
- まとめても結構遅い（c）
  - 命令キャッシュミスを観測（関数ポインタ）
  - 小さい命令はコンパイルしない、積極的な言語レベルインライン化が必要

```
n = 1_000_000
def f0(a) = a + 0
...
def f999(a) = a + 999
i=0
while i<n
  f0(i); ...; f999(i)
  i += 1
end
```

条件	実行時間（秒）
(a) インタプリタ実行	13.27
(b) <hash>.so を 1000 個ロード	46.74
(c) all.so を 1 個ロード	22.47

表 2. コードストレージの種類による実行時間の比較（秒）

議論

- 課題
  - .so 使って楽してはダメ
    - 独自バイナリフォーマットとローダーが必要
  - リクエストでプロファイルをとることで投機的なコード返信
    - 「h1, h2, h3 が欲しいなら、きっと h4, h5, h6 が欲しいだろう」
    - バルク返送できて効率化
  - コードストア上限の設定
- AI (Coding agent) の目覚ましい発展
  - 「**楽に速い処理系を作る**」目的の研究だが、AI に任せてできるなら不要では…？　真面目に比較する必要がありそう

関連研究

- コンパイルを他計算機でオフロードする研究[6]
- Cコンパイラを使ったJITの研究[9, 2, 1]
- (JIT) コンパイル済みコードを共有する研究[17, 4, 7, 8]

ASTro のアイディアである**マークル木によるハッシュ値**を名前として分散環境でAST（部分木）の単位で部分木を共有する、というのは多分ユニーク

(ASTro の名前の由来です)

まとめ

- ASTro により、高速な C コードへの変換を容易に実現
- ASTro JIT の提案
  - 生成物を実行時にコンパイルし、JITできることを確認
  - レイヤー構成にすることで、コンパイル済みコードを複数マシンで共有可能
- 課題
  - 独自バイナリフォーマット＋ローダが必要
  - 大規模プログラムでの評価