ASTro による JIT コンパイラの試作

笹田 耕一 [1]

[1] STORES 株式会社
ko1@st.inc

概 要

本稿では、部分評価による高速化を容易に実装可能な言語実装フレームワーク ASTro 向けに、Just-in-Time（JIT）コンパイラを試作した経験について述べる。

抽象構文木（AST）のノードを辿る単純なインタプリタは、実装は容易だが低速である。具体的な AST に対してインタプリタを部分評価（PE）することで、インタプリタの実装は単純なまま、高速な実行コードを出力する手法が知られており、われわれは PE を C コンパイラを用いてポータブルに実現する ASTro フレームワークを提案している。しかし、C コンパイラを用いる都合上、技術的課題があり、JIT コンパイラの開発には着手できていなかった。

そこで、ASTro 向けに分散キャッシュを基本とする JIT コンパイラを試作し、その評価結果について紹介する。予備評価によって、実現可能性は示すことができたが、課題は多い。

1 はじめに

我々は、C コンパイラによる最適化を活用し、部分評価によるインタプリタの高速化を行うフレームワークである ASTro (AST-based Reusable Optimizer)[10] を提案している。

ASTro フレームワークでは、ターゲット言語 L のインタプリタを、抽象構文木（AST）のノードごとの挙動を定義することで自動的に構築する。また同時に、そのインタプリタを、与えられた AST（本稿では、プログラム全体の AST に限らず、その部分木も含めて AST と呼ぶ）で部分評価する部分評価器も自動生成する。部分評価した結果は C ソースコードとして生成され、これを C コンパイラを用いてコンパイルすることで、当該 AST を効率的に実行するネイティブコードを得ることができる。

先行研究では、生成された部分評価器を、事前（AOT: Ahead-of-Time）コンパイルや、実行後に得られたプロファイル情報を用いたプロファイル誘導型最適化（PGO: Profile-Guided Optimization）として利用した。具体的には、インタプリタの実行前あるいは実行後に部分評価によって生成されたネイティブコードを事前に組み込んでおき、次回以降の実行でそれを再利用する手法である。その結果、ベースラインインタプリタと比較して数倍〜数十倍の性能向上を達成できることを示した。

一方で、ASTro フレームワークは実行時（JIT: Just-In-Time）コンパイルも想定しているが、C コンパイルを実行時に行うことは一般に大きな実行時オーバヘッドを伴う。そのため、C コンパイラを用いた JIT コンパイルが現実的に成立するかどうかは明らかではなかった。そこで、本稿では、ASTro フレームワークにおいて JIT コンパイルを実現するための仕組みを試作し、その設計および予備評価の結果を報告する。

試作した JIT コンパイラは、インタプリタプロセス内で動作する JIT コンパイル用スレッド L0、同一計算機上に常駐する L1 プロセス（デーモン）、および別の計算機上で動作する、実際に C コンパイラを起動する L2 プロセス（デーモン）から構成される。ネイティブコードの保持の有無、およびネイティブコードのコンパイルについて、インタプリタは L0 に問い合わせを行い、L0 は L1 に、さらに L1 は L2 に問い合わせを転送する。L1 および L2 は複数のインタプリタプロセス間で共有され、同一 AST に対して生成されたコンパイル済みネイティブコードを再利用可能とする。

2　ASTro フレームワーク

ASTro フレームワーク（図 1）は、比較的少ない記述量で、高性能かつ多様な実行環境に対応可能なインタプリタを生成することを目的としている。

2.1　動機

高性能なインタプリタを記述するためには、一般的に次のような手順を踏む。AST をその言語向け仮想マシン（VM）の命令セット向けに変換し、VM 上で実行する。VM の実行においては、そのバイトコードを、さらにネイティブコードへ変換する（JIT、もしくは AOT）コンパイラを用いることで、高性能な処理系を実現している。VM は、言語ごとに VM を開発するだけでなく、共通の VM（JavaVM や.NET CLR）向けにコンパイルすることもある。

この一連の変換、VM の設計と実装、その上での JIT コンパイラの設計と実装は、一般的に開発コストがかかる。また、前段（言語文法の変化による AST の構造や VM の命令）で仕様変更があった場合、後段（VM や JIT コンパイラ）への変更も波及して行う必要がある。これは、とくに VM や JIT コンパイラなどのコンポーネントの開発を分業しているような場合、調整コストが高い作業にもなる。

そこで、インタプリタを対象プログラムに関して部分評価する手法に着目する。この手法は 第一二村射影 [3] として知られている。部分評価器 PE によってインタプリタ I をプログラム P に関して部分評価することで、P に特化した残余プログラム $PE(I, P)$ を得ることができる。特化したプログラムは高速に実行できる可能性が高く、VM や、それをベースにした JIT コンパイラを作らずに高性能なインタプリタを作ることができる。

I を AST 各ノードをどのように評価するかを記述したインタプリタ、P をインタプリタで実行したいプログラムの AST とする、部分評価を用いた高速なインタプリタを生成するものとして、Truffle フレームワーク/Graal コンパイラの組み合わせ [15, 12, 13] がある。Truffle フレームワークがインタプリタ記述向け DSL（Java ベース）と、その部分評価を担当し、Graal コンパイラがその部分評価結果をネイティブコードに変換する。これらは、Java プラットフォーム上に実装され、`x86_64`、`ARM64` 環境向けのネイティブコード生成に対応する。

Truffle/Graal コンパイラは、ネイティブコード生成まで、独自の処理系を構築することで、高い性能のインタプリタを構築することができるが、例えば新しい CPU アーキテクチャを追加する、といったことは開発コストがかかる。また、Java の基盤上で実装されているので、利用可能な環境は限られる。

2.2　ASTro の提案

ASTro フレームワークは、部分評価の結果を C コンパイラのインライン化によって最適化しやすい C のソースコードとすることで、多くの場所でも実行できる、高性能なインタプリタの生成を可能にしている。なぜなら、C コンパイラは多くの計算機アーキテクチャ向けに成熟した最適化基盤として提供されており、バックエンド実装を新たに開発することなくネイティブコード生成を行えるためである。

ASTro フレームワークは、AST のノードごとに、どのように実行するか、その挙動を定義する `node.def` というファイルから、ASTroGen というツールによって、ベースラインインタプリタ（図 1 の (1)）、および部分評価器などを生成する。

部分評価器は、ノードの挙動（`node.def`）と AST をデータとして部分評価した結果として、C コンパイラにとってインライン化しやすい C のソースコード（図 1 の (2)）を出力する。評価関数とディスパッチ関数を分離することで、特化したディスパッチ関数を作るだけでインライン化・最適化がされやすい C のソースコードを出力する部分評価を行う。部分評価に、ノードの挙動定義に利

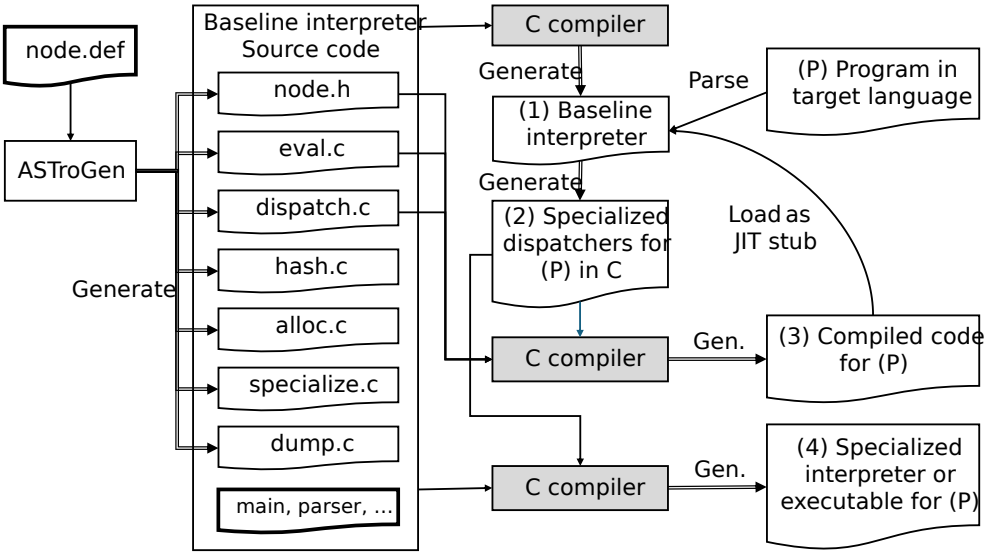

図 1. ASTro フレームワークの生成物のワークフロー（論文 [10] から引用）

用した C プログラムの解析を必要としないため、容易に部分評価器を実現できる点が、ASTro の重要な特徴である。具体的なディスパッチ関数の特化手法は、先行研究 [10] を参照していただきたい。

C のコンパイラを用いると、一般的にコンパイル時間が問題になる。ある AST に対するネイティブコードは、同じコンパイル結果となることから、同じ AST に対しては、コンパイル結果を共有することができる。そのため、ASTro では、AST のすべての部分木について Merkle 木 [5] によるハッシュ値を計算しておくことで一意な識別子を付与し、プロセスをまたいだコンパイル結果の共有を可能としている。

部分評価した結果の C ソースコードをコンパイルした結果は、そのプログラム（の一部）に特化した実行ファイルやインタプリタの生成に利用したり（図 1 の (4)）、JIT stub として利用できる（図 1 の (3)）。本稿が取り組むのが、まさにこの部分である。

本稿を読むにあたり必要な知識は次のとおりである。

- AST の各ノードはディスパッチ関数を持ち、それを実行することで、AST ノードの実行を行う（木を辿る単純なベースラインインタプリタ）。
- 各ノードのディスパッチ関数について、ASTroGen が生成した部分評価器（PE）により、特定の AST に特化したディスパッチ関数を表現する、子ノードを含めてインライン化などが可能な C ソースコードを生成できる。
- 特化したディスパッチ関数をコンパイルしたネイティブコードを利用することで、ベースラインインタプリタよりも高速に実行できる。
- 特化したディスパッチ関数は AST ノードの構造（Markle 木）によるハッシュ値によって名前がつけられ、プロセスをまたいでも一意に区別でき、共有可能である。

3 ASTro JIT の設計

本稿では、試作した JIT コンパイラを ASTro JIT と呼び、これによりインタプリタ実行の高速化を図る。具体的には、インタプリタ実行中に、与えられた AST（ノード）でインタプリタを部分評価した結果を C コンパイラによってネイティブコードに変換し、それを実行時にロードする。

なお、ここで議論する ASTro JIT は、ASTro フレームワークを利用しない場合でも、実行対象のプログラム片を C ソースコードに変換可能であり、その C ソースコードにプロセスをまたいで

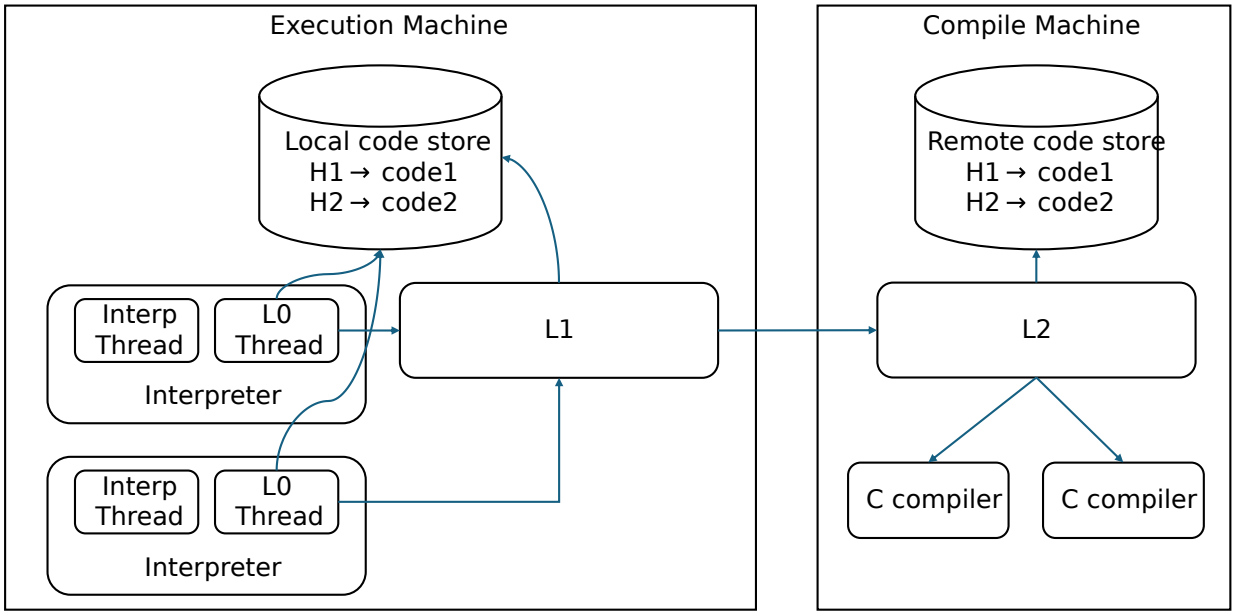

図 2. 階層型キャッシュのためのインタプリタおよび L0, L1, L2

ユニークな名前がついている、という条件を満たす別のフレームワークでも同様のアイディアを適用可能である。

3.1 階層型キャッシュプロセス

当然ながら、C コンパイラによる特化関数のコンパイルは遅い。そこで、コンパイルはインタプリタとは別プロセスで行うことができるようにする。また、そもそもインタプリタを動かす計算機と別の計算機を用いることができるよう、ネットワーク越しの別マシンを利用できる構成とする。インタプリタがキャッシュプロセスと直接通信を行うのは、ASTro JIT の実装および保守の複雑性が高くなるため、別マシンと通信するプロセスを別途用意することとした。まとめると、JIT のために、次の 3 種類のコンポーネントを用意する（図 2）。

- L0: インタプリタ内に存在する、L1 と通信するためのスレッド。
- L1: L0 と同じ計算機内に常駐するプロセス（デーモン）で、L0 から要求を受け取り、必要であれば L2 へ要求を送る。複数の L0、1 つの L2 と通信する。
- L2: L1 からソースコードを受け取り、実際に C コンパイラによってネイティブコードを生成するプロセス（デーモン）。複数の L1 と通信する。

構成としては、L1 は L2 を兼ねても良い。また、L1 と L2 は同一マシンにあっても、別マシンであっても良い。L2 は C コンパイラを呼び出すが、`distcc`[8] などの分散コンパイル基盤を使って、別計算機上で実行しても良い。また、L3、L4、... とリレーする先を増やすことは可能である。例えば、データセンター内で 1 つ中継プロセスを置くといった用途が想定される。

L0、L1、L2 はそれぞれコンパイル済みコードを格納するコードストアを参照し、AST のハッシュ値に対応するネイティブコードがあれば、それを利用する。なければ、L0 は L1 へ、L1 は L2 へソースコードを送り、そのソースコードのコンパイルを要求する。

この構成により、異なるプロセス、異なる計算機上で走る複数の L0 が、同じハッシュ値の AST ノードに対するコンパイル済みコードを共有することができる。これは、コンパイルをリクエストしたプロセス以外でも共有可能ということで、たとえば次のようなケースで有効に機能すると考えられる。

- あるプログラムを長時間動かすケース（通常の JIT コンパイラが対象とするケース）
- 同じプログラムを連続して起動と終了を繰り返すケース
- 同じプログラムを `fork` によって複数同時実行するケース
- 同じプログラムを複数の計算機上で何個も起動するケース（たとえば、複数マシン上でスケールアウトするようなウェブアプリケーション）

また、コンパイルする粒度をどのように選ぶかによって（例えば、対象言語の関数単位でコンパイルするなど)、AST の部分木を共有可能なので、同じプログラムではなく、少し変更したプログラムにおいても、コンパイル済みネイティブコードを共有可能な部分は相当量あると思われる。

L1 と L2 を分けることで、セキュリティ上 C コンパイラをプロダクション環境に置きたくない、といった場合でも、コンパイルを行う計算機を分けることで状況が緩和される。

しかし、L1、L2 といったインタプリタ以外のプロセスを管理しなければならない、というのは、通常インタプリタプロセス内にある JIT コンパイラに比べて利用時の運用コストを増加させる。

3.2 コードストアの表現

コードストアは、AST のハッシュ値とコンパイル済みネイティブコードの組を管理し、ハッシュ値を渡されれば、対応するネイティブコードをできるだけ高速に返すコンポーネントとする。JIT コンパイラとしてインタプリタプロセスが活用するには、コンパイル済みネイティブコードをロードし実行可能でなければならない。

実現方法はいろいろ検討できるが、今回は簡単にファイルシステムと共有オブジェクトをベースに実装した。詳細は次章で述べる。

3.3 階層型キャッシュの通信メッセージ

L0、L1、L2 は次のようなメッセージを送受信する。

- $query(h)$: L0 は、あるハッシュ値 h に対応するコンパイル済みネイティブコードがないか、L1 に問い合わせる。L1 は L2 に同様に問い合わせる。あれば、それぞれそのネイティブコードを返す。
- $compile(h, src)$: L0 は、あるハッシュ値 h に対応するソースコード src のコンパイル済みネイティブコードを得るため、L1 に問い合わせる。同様に、L1 は L2 に問い合わせる。L2 はそのコードをコンパイルし、コンパイルした結果を返す。

インタプリタプロセス（インタプリタスレッドと L0 スレッド）の挙動は次のようになる。まずは、ある AST のロード時、もしくは最初の実行時の挙動である。

- インタプリタスレッドは、ある AST（ハッシュ値 h）をロード、もしくは実行しようとしたとき、L0 スレッドへ h に対応するコンパイル済みの特化ディスパッチ関数があるか問い合わせる。
- L0 スレッドは、コードストアを検索し、発見できなかったとき、$query(h)$ で L1 に問い合わせる。
- L0 へ L1 から存在すると返答された場合、再びコードストアを検索し、発見した特化ディスパッチ関数を、その AST のディスパッチ関数へと置き換える。存在しないと返答されたら、発見できなかった、という情報をその AST に記録する。

次に、ある AST が頻繁に実行されてると判断されたときの挙動である。

- インタプリタスレッドは、ある AST（ハッシュ値 h）が頻繁に実行されたと判断したとき、L0 にそのディスパッチ関数を特化しコンパイルすることを要求する。
- L0 スレッドは特化したディスパッチ関数のソースコードを部分評価器によって生成し（src）、L1 に $compile(h, src)$ と問い合わせる。
- L1 からコンパイル済みコードができたという通知を受ければ、それをその AST のディスパッチ関数へと置き換える。

インタプリタスレッドと L0 スレッドは互いに並行に実行し、インタプリタプロセスは L0 スレッドからの返信を待つことはせず、L0 が特化ディスパッチ関数に差し替えるまで、デフォルトのディスパッチ関数を用いて実行を継続する。

特化ディスパッチ関数のコンパイルを要求するタイミングは調整しがいのあるテーマである。多くの JIT コンパイラは実行回数が閾値をこえたときに行うが、例えば、プログラムのパースを終了した後に、存在する AST（各部分木）のディスパッチ関数をすべて特化すれば、それはそのプログラムについての AOT コンパイラとなる。そのコンパイル結果を共有することで、そのプロセスだけでなく、同じプログラムを実行する他のプロセスが、起動直後からコンパイル済みコードを利用可能となる。

3.4 AST の状態遷移

ある AST（ノード）について、インタプリタプロセスは次のように状態を記録する。

Unknown まだ *query* リクエストをしていない AST。

Querying *query* リクエストをしたが、まだ返答がない AST。

NotFound *query* リクエストをした結果、コンパイル済みネイティブコードがないと判断された AST。

Compiling *compile* リクエストをしたが、まだ返答がない AST。

Compiled コンパイル済みネイティブコードをもつ AST。

これらの状態に基づいて、どのリクエストを依頼可能か、もしくはできないのかを管理する。具体的には、*Unknown* 状態の AST について、*query* リクエストを行うことができる。また、 *Compiling* と *Compiled* 以外の状態からは、*compile* リクエストを行うことができる。

4 ASTro JIT の実装

前章で述べた設計に基づき、評価のために実装した ASTro JIT の実装について述べる。

4.1 階層型キャッシュの実装

L0 は C 言語で、L1 および L2 は Ruby スクリプトで実装した。

L0 スレッドを Pthread を用いて実装した。インタプリタスレッドから L0 へのリクエストは、同期キューを用いて行い、L0 スレッドは非同期に AST のディスパッチ関数を差し替えることで、それ以降のその AST の実行時には、特化されたネイティブコードが実行されるようにした。

プロセス間の通信方法であるが、今回は L0 と L1 は Unix domain socket、L1 と L2 は TCP ソケット通信を行うプログラムとして実現した。送りあうメッセージは、「メッセージの種類、サイズ、対象とする AST のハッシュ値、ペイロード（*compile* 時のソースコード）」の 4 つ組を送りあうバイナリメッセージプロトコルとした。実装の単純化を優先したためであるが、例えば GRPC などを利用しても良い。

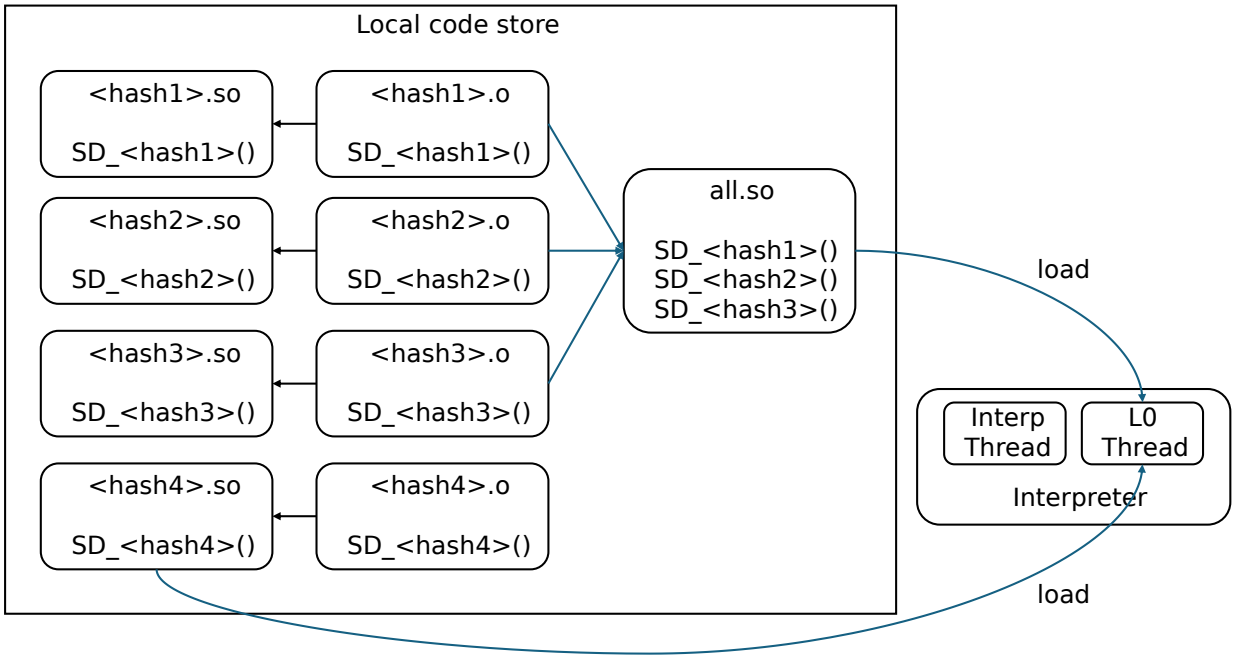

図 3. 共有オブジェクトを用いたコードストアの実装

すでにコンパイル済みの特化関数があればそれを利用するため、本実装ではプログラムのロード時に各関数の AST について $query(h)$ メッセージを発行するようにした。

頻繁に実行する AST をコンパイル対象とするため、本実装ではある関数を 100 回以上実行したとき、その関数の AST を頻繁に実行するものとして、*compile* メッセージを用いてディスパッチ関数を特化して C コードを生成し、それを L0 が L1 にコンパイルを要求するようにした。

4.2 コードストアの実装

コードストアおよびロード可能なネイティブコードを実現するために、ファイルシステムと共有オブジェクトをベースに実装した（図 3）。なお、Linux 環境に限定した実装であり、他の環境で動くかどうかは検証していない。

- L0, L1 が参照するローカルコードストアには、コンパイル済みネイティブコードは共有オブジェクトファイル（以降 SO、ファイル名は`<`ハッシュ値`>.so`）として、ファイルシステム上に置く。L0 は `<`ハッシュ値`>.so` ファイルをファイルシステム上で探索し、あればコンパイル済みコードがあることがわかる。SO は `SD_<`ハッシュ値`>()` という関数名で特化したディスパッチ関数を持っているため、`dlsym()` で関数ポインタを検索することができる。
- L2 が参照するリモートコードストアには、C ソースコードをコンパイルしたオブジェクトファイル（`<`ハッシュ値`>.o`）を置く。そのファイルがなければ、C コンパイラによってそのファイルを作成する。
- L2 から L1 へ、*query* もしくは *compile* メッセージの返信としてオブジェクトファイルを返送する。L1 はリンカを用いてそのオブジェクトファイルを SO にし、ローカルコードリポジトリに保存する。
- L1 は L0 にあるハッシュ値のネイティブコードを発見したと L0 に返す。L0 はローカルコードストアを探索し、特化ディスパッチ関数に差し替える。

1 点だけ、次のような効率化を行った。インタプリタが複数の SO ファイルをロードするのはメモ

	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）

リ使用量の観点から効率が悪い[1]ため、あるタイミング（例えば、L1 が暇となったタイミング。本実装では L1 への通信が 3 秒間行われなかったとき）でローカルコードストアに存在するオブジェクトファイルをまとめた `all.so` ファイルをリンカで生成する。つまり、インタプリタ内の L0 は、まず `all.so` をロードし、これに対して関数名 `SD_<ハッシュ値>()` を `dlsym()` によって検索する。なければ `<ハッシュ値>.so` をファイルシステム上で探索し、さらになければ L1 へ問い合わせを行う。

効率のためには SO ではなく、独自のネイティブコードアーカイブ形式、および専用のネイティブコードのローダーを用意するのが良いと思われるが、本実装ではそこまでの最適化は行っていない。

コードストアにファイルシステムおよび SO を用いる利点は、全体的な実装が容易であること（移植性も高い）、デバッグの容易さ（デバッグ情報を付加し、デバッガで普通のプログラムとして調査することができる）、欠点はメモリ効率が悪いこと、Copy & Patch[16] などの用途には向かないことがある。

5 予備評価

ASTro JIT の実現可能性を評価するため、簡単な予備評価を行った。本格的な評価は、ASTro フレームワークを用いて実装している言語が簡易的なものであり、まだ本格的なベンチマークセットがないため、行えていない。

5.1 評価環境

実験対象プログラミング言語は、Ruby の文法だが、Ruby から多くの機能を制限した naruby[10] を用いた。計算機環境は次のとおり。

インタプリタ実行用計算機（L0, L1） AMD Ryzen 9 5900HX (16 threads), Ubuntu 24.04

コンパイル用計算機（L2） Intel i7-13700H (20 threads), Ubuntu 24.04, gcc version 13.3.0

計算機はそれぞれスイッチングハブ（1Gbps）で接続されている。

5.2 JIT コンパイルの効果

図 4 のような、計算量が $O(n^2)$ となるような簡単な数値計算プログラムを作成し、JIT コンパイルの効果を確認した。実行した結果は表 1 となった。

実行時に `prime?` 関数を部分評価した C ソースコードを L2 へ転送し、それを L0 へ送り、実行を差し替えていることを確認した。実行時間を見ると、JIT コンパイルによる速度向上を確認できた。2 回目の実行結果は、すでに `all.so` としてすべてのコンパイル結果をキャッシュ済みであり、インタプリタプロセス内で完結する。しかし、この規模では JIT コンパイルのオーバヘッドはほぼ観測できなかった。

L1 が L2 を兼ねる（つまり、コンパイルをインタプリタを実行する計算機で行う）構成でも試したが、この規模だと結果に違いはなかった。

[1] SO は、最低でも OS のページサイズ（典型的には 4KB）の倍数のサイズとなり、実行する機械語が 1B でも、余分な空間（ストレージおよび実行に利用するメモリ）を使う。

```
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
```

図 4. 素数判定関数を 0〜1M において適用する簡単な naruby プログラム

キャッシュがない状態でプログラムのコンパイル時間[2]、インタプリタが L0 に依頼してから、L0 が特化ディスパッチ関数に置き換えるまでの時間を計測したところ、30 マイクロ秒だった。2 回目にすでにキャッシュが残っている場合、つまり `all.so` からロードする場合、1 マイクロ秒未満だった。

5.3 ストレージの利用量

次のようなプログラムを作成し、f0〜f999 の 1000 個の関数が、どの程度容量を消費するかチェックした。生成された SO ファイルからは、`strip --strip-unneeded` によって実行に不要な個所を削除した。

```
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
```

[2] `clock_gettime(CLOCK_MONOTONIC)` で計測。

結果、<hash>.so は約 14 キロバイトのファイルが 1,000 個作成された。それぞれに特化されたディスパッチ関数 SD_<hash> が格納されており、10〜12 バイトであった。まとめた all.so は約 120 キロバイトであり、all.so のようにまとめる仕組みの有効性が確認できた。

5.4 外部ライブラリをコードストレージとする性能劣化

前節で述べたプログラムを、各関数すべてコンパイル済み（.so、もしくは all.so がある状態）で実行した実行時間は表 2 という結果となった。

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）

インタプリタ実行よりも、コンパイル済みバイナリ（.so）をロードしたほうが遅くなっている。

(b) が顕著に遅いのは、f0 などの各関数に相当する命令数バイトが、各.so がページアラインメント（4KB など）の先頭に配置されるため、命令キャッシュのヒット率が激減するためである。f0 などの各関数に対するネイティブコードは十分に小さい機械語（2 命令）となり最適なコードが生成できているが、最適化の効果を命令キャッシュミスのデメリットが大きく上回っているため性能低下が顕著となった。そのため、独自ローダを用いて密にメモリ上にロードする必要性が確認できた。

(c) が遅いのも、インタプリタからコンパイル済みの 1000 個の関数を呼ぶため、命令キャッシュのヒット率が下がっていたためであった。これは、AOT コンパイルによって all.so ではなく、実行ファイルにコンパイル済みのネイティブコードを埋め込んだ場合も同じような特性を示していることを確認している（つまり、JIT に関係ない性質である）。特化した命令数が十分小さい場合はインタプリタ実行にとどめる、といった工夫も検討したい。

6 議論

ASTro JIT の課題について議論する。なお、ASTro フレームワークがもつ課題、例えばハッシュ値の衝突についてなどは先行研究 [10] での議論も参照いただきたい。

独自ローダ 4.2 で述べたように、共有オブジェクトを用いたロード可能なネイティブコード表現は冗長であり、メモリ効率の面で課題がある。また、5.4 で示した通り、ページ境界にロードされる個別の.so ファイルを読むという実装では性能に問題がある。本実装では、実験的評価を優先し、移植性およびデバッグ容易性を重視して共有オブジェクトを採用したが、オブジェクトファイルを直接実行可能にする独自ローダの開発は重要な検討課題である。ただし、この種のローダは計算機環境ごとに個別の実装が必要となるため、「どこでも容易に動く」という目標から一歩後退する。それでも、計算機環境ごとのコンパイラを一から書くよりは容易であろうと思われる。

コードストア上限サイズの設定 現在の実装では、コードストアに一度生成したネイティブコード（SO およびオブジェクトファイル）を削除しないが、実際に利用するには、ある閾値をもって削除する必要がある。例えば、コードストアの総サイズが一定の閾値（例えば 1GB の上限）を超えた場合、LRU を用いて削除する、といった具合である。LRU で削除するためには利用履歴が必要となるが、その履歴をどのように管理するかは自明ではない。

AOT コンパイルとの併用　ASTro フレームワークは、事前コンパイル（AOT コンパイル）を容易に利用可能である [10]。本稿では JIT コンパイルと AOT コンパイルを併用する実験は行わなかったが、(1) 実行時情報がなくてもできる AOT コンパイルと、(2) 実行時情報を加味した JIT コンパイル、と分担できる。これによって、1 度しか実行しない AST は (1) でそこそこ速いコードが実行され、頻繁に実行する AST は (2) で速い、それでいて JIT プロセスへの負荷は (1) によって少ない（初期ロード時のバーストクエリがなくなる）、といった構成が可能であるので今後試したい。

投機的プリフェッチ　L2 への *query* リクエストが h1, h2, h3 と続いていた時、統計情報などから、続いて h4, h5, h6, ... へのクエリが届く、と推論できる。例えば、複数の計算機で同じプログラムを実行すると、このような推論をするのに妥当性がある。そこで、まだクエリが来ていない段階で、投機的に L2 が L1 に対してコンパイル済みコードを送ることで、*query* もしくは *compile* リクエストを減らすことができる。ただし、アグレッシブに行うと余計な転送コストおよびストレージコストがかかる。

実行環境とコンパイル環境の整合性　インタプリタと C コンパイラを実行する計算機が異なるため、たとえば RISC-V を用いた計算機を、ARM64 環境でコンパイルしたい、といった場合、L2 から適切なクロスビルドを行う必要があり煩雑となる。また、L0〜L2 間のメッセージを拡張し、ビルド環境を含んだものにしなければならないかもしれない。

セキュリティ　L1、L2 がインタプリタと独立して動作し、さらに L2 が別の計算機上で実行されることから、アタックサーフェースが増大し、セキュリティ上の懸念が生じる。本稿では特にセキュリティ機構を設けていないが、実運用を想定する場合には、適切な対策について検討が必要である。

7　関連研究

JIT コンパイルを別の計算機で実行したり [6]、JIT コンパイルの結果を複数のインタプリタプロセス間で共有する研究が多く報告されている [17, 4]。本研究は、共有の単位を AST をベースに検討している点がそれらと異なる。また、これらの研究では分散キャッシュの設計を AST 単位で体系的に扱ってはいない。

ある言語で書かれたプログラムを高速化のために C 言語に変換（もしくは C バックエンドを利用）して実行するというのは、すでに多くのアプローチがある [9, 2, 1]。JIT コンパイラのバックエンドとして利用する、というのは最近では Ruby の MJIT[11] で挑戦されてきた[3]。しかし、JIT コンパイラとしては、コンパイル遅延の問題から、実運用において広く利用されるには至らなかった。ASTro JIT では、生成されたコードを AST のハッシュ値により共有できるという構造、およびコンパイルを別の計算機で実行するという性質から、これらの問題に対する一つの解決策となり得る。

コンパイルの実行、および結果を分散キャッシュする仕組みは、`distcc`[8] や Bazel など多数提案されている [7]。ただ、これらは JIT コンパイラで利用するという文脈では語られていない。

Truffle/Graal フレームワークにおいても AST ノード単位での置き換えや部分評価が行われている [12, 14, 13]。それらは単一プロセス内で完結しており、プロセス間や計算機間での再利用は想定されていない。彼らの提案している実行時プロファイルに基づいたノード置き換えといったテクニックなどは、ASTro でも今後積極的に取り入れていくが、ノードのハッシュ値というアイディアは、置き換え結果を含めてのコンパイル結果の共有を可能にする。

[3] ちなみに、インタプリタから C コンパイラを直接呼び出すと、C コンパイラプロセスが終了するタイミングで `SIGCHLD` シグナルが届くといった副作用が観測されたという Ruby/MJIT の経験が、L0 と L1/L2 を分離した理由の一つである。

8 まとめ

本稿では、AST を辿るインタプリタを対象として、部分評価により特定の AST から C コードを生成可能な ASTro フレームワークに対し、JIT 機能を追加する試作を行った。分散キャッシュを実現するために、L0、L1、L2 からなる階層型構成と、それらの通信方式を設計・整理した。実装としては、共有オブジェクトおよびファイルシステムを用いた比較的単純な構成を採用し、設計の実現可能性と現状の問題点を確認した。

6 章で議論したように検討課題は多いため、今後も研究開発を進めていきたい。

参考文献

[1] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science and Engg.*, 13(2):31–39, March 2011.

[2] Andrea Corallo, Luca Nassi, and Nicola Manca. Bringing GNU emacs to native code. *CoRR*, abs/2004.02504, 2020.

[3] Yoshihiko Futamura. Partial evaluation of computation process—anapproach to a compiler-compiler. *Higher Order Symbol. Comput.*, 12(4):381–391, December 1999.

[4] Meetesh Kalpesh Mehta, Sebastián Krynski, Hugo Musso Gualandi, Manas Thakur, and Jan Vitek. Reusing just-in-time compiled code. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

[5] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, page 369–378, Berlin, Heidelberg, 1987. Springer-Verlag.

[6] Fumika Mochizuki, Tetsuro Yamazaki, and Shigeru Chiba. Bluescript: A disaggregated virtual machine for microcontrollers. *The Art, Science, and Engineering of Programming*, 10(3), October 2025.

[7] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. Build systems à la carte. *Proc. ACM Program. Lang.*, 2(ICFP), July 2018.

[8] Martin Pool. distcc, a fast free distributed compiler. https://www.distcc.org/distcc-lca-2004.html, 2004.

[9] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications - a way ahead of time (WAT) compiler. In *Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS 97)*, Portland, OR, June 1997. USENIX Association.

[10] Koichi Sasada. ASTro: An ast-based reusable optimization framework. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL '25, page 22–32, New York, NY, USA, 2025. Association for Computing Machinery.

[11] Jonan Scheffler. Mjit: A method based just-in-time compiler for ruby. https://www.heroku.com/blog/ruby-mjit/.

[12] Christian Wimmer and Thomas Würthinger. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, page 13–14, New York, NY, USA, 2012. Association for Computing Machinery.

[13] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. *SIGPLAN Not.*, 52(6):662–676, June 2017.

[14] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 662–676, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, page 73–82, New York, NY, USA, 2012. Association for Computing Machinery.

[16] Haoran Xu and Fredrik Kjolstad. Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021.

[17] Xiaoran Xu, Keith Cooper, Jacob Brock, Yan Zhang, and Handong Ye. Sharejit: Jit code cache sharing across processes and its practical implementation. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.