C 言語を用いたマシン非依存な JIT コンパイラ作成フレームワーク

栗田 洋輔　　千葉 滋

東京工業大学大学院 情報理工学研究科 数理・計算科学専攻

要 旨

近年、動的言語の普及とともに JIT コンパイラの需要が高まっている。JIT コンパイラは有用だが、それを作成するには、アセンブリや機械語に関する深い知識を必要とする。また、アセンブリや機械語を用いて実装した JIT コンパイラは、マシン依存であり移植が難しいという問題もある。そこで我々は、C 言語のみを用いることでマシン非依存な JIT コンパイラを作成することができるフレームワークを提案する。このフレームワークのユーザーは、テンプレートと (テンプレートの) 組み立て情報生成コードを C 言語で記述する。インタープリター内で組み立て情報生成コードを用いて組み立て情報を生成し、それをフレームワークが提供する最適化コード生成器に渡すことで最適化コードを生成することができる。局所的な最適化を行えるだけでなく、大域的な最適化機構をユーザーが組み込み可能な点が大きな特徴である。我々は、単純化した lisp インタープリターに対し、このフレームワークを用いて JIT コンパイラを実装し、有用性を確認した。

1　はじめに

近年、動的言語が注目を浴びている。新しい動的言語を作るときはインタープリターで実装が行われるが、インタープリターでのプログラムの実行は、コンパイルされたプログラムに比べ実行速度が遅いという欠点がある。

実行時 (Just-in-Time, JIT) コンパイルという技法を用いれば、インタープリターにおけるこの欠点を解消することができるが、JIT コンパイラの作成には困難な点が多い。

まず、JIT コンパイラの作成には、マシン固有のアセンブリ言語に関する知識を必要とする。また、ポータビリティを実現するためには、マシン非依存な中間表現を学習し、それを用いて実装しなければならなかった。

本論文では、C 言語のみでマシン非依存な JIT コンパイラを作成するためのフレームワークを提案する。このフレームワークのユーザー (JIT コンパイラの開発者) はテンプレートと、テンプレートの組み立て方法を指定するコードを記述する。フレームワークは、組み立て方の情報を元に、最適化コードを生成する。インタープリタの開発者は、本フレームワークを用いることで、C 言語のみを用いて、ポータブルな JIT コンパイラを容易に作成することができる。大域的な最適化機構をユーザーが組み込むことが可能な点が特徴である。

2　典型的なインタープリター

コンパイラに比べ、インタープリターには様々な実行時オーバーヘッドが存在する。この実行時オーバーヘッドは、特定のインタープリターに固有のオーバーヘッドと、典型的な実装であればどのようなインタープリターにも存在する本質的オーバーヘッドがある。この章では、バイトコードを解釈するバイトコードインタープリターと、抽象構文木をそのまま巡回する非バイトコードインタープリターのそれぞれにおいて、本質的オーバーヘッドを説明する。

2.1　典型的なバイトコードインタープリター

典型的なバイトコードインタープリターの核となるループは、非常に単純である。次のバイトコードをフェッチし、switch 文を用いて適切な実装にディスパッチが行われる。図 1 は、典型的なバイトコードインタープリターの核となるループと、「x + 3」を計算するバイトコードの配列である。

```
compiled code:
unsigned char code[] = { ...,
  PUSH_GLOBAL_VAR,
  x,
  PUSH_CONST,
  3
  ADD, ...};

bytecode implementations:
int *stackPointer;
unsigned char *instructionPointer = code - 1;
for (;;) {
  unsigned char bytecode = *++instructionPointer;
  switch (bytecode) {
  /* ... */
  case PUSH_GLOBAL_VAR:
    *++stackPointer =
        getVarFromSymbol(*++instructionPointer);
    break;
  case PUSH_CONST:
    *++stackPointer = *++instructionPointer;
    break;
  case ADD:
    --stackPointer;
    *stackPointer += stackPointer[1];
    break;
  /* ... */
  }
}
```

図 1: 典型的なバイトコードインタープリター

このインタープリターは、次のバイトコードにディスパッチを行う switch 文を含んだ無限ループである。switch 文の内部の各々の case は、一つのバイトコードを実装している。また、各々の case では、次のバイトコードを実行するために、switch 文を break し、無限ループの始めに戻るようになっている。

このインタープリターには、以下のようなオーバーヘッドが存在する。

- instructionPointer のインクリメント
- メモリからの次のバイトコードのロード
- オペコードによる switch
- case 文から戻るジャンプ
- 定数のロード

ネイティブコードの実行では、CPU によって、命令レジスタが自動的にインクリメントされ、次のマシン命令がフェッチされる。バイトコードインタープリターの場合は、そのような処理はソフトウェアによって明示的に行われるため、オーバーヘッドが発生してしまう。また、典型的な実装では、バイトコードを実行する度にオペコードのチェックを行うので、それもオーバーヘッドとなる。最後に挙げた「定数のロード」は、図 1 におけるバイトコードの中に埋め込まれた「3」や「x」をロードする処理のことである。

典型的なバイトコードインタープリターは作成や理解が容易で移植性が高いが、このようなオーバーヘッドのために処理は遅い。特にバイトコードが、(図 1 における PUSH_CONST のような) 単純な処理を行う場合は、実行時間の大部分が、バイトコードに対応する本質的な処理ではなく、ディスパッチのオーバーヘッドに消費されてしまう。

2.2 典型的な非バイトコードインタープリター

```
expr: (+ x 3)

int eval(List* expr) {
  switch (testElementType(expr)) {
  /* ... */
  case NUMBER :
    return getIntegerElement(expr);
  case SYMBOL :
    return getVarFromSymbol(expr);
  case PLUS :
    return eval(getSecond(expr)) + eval(getThird(expr));
  /* ... */
  }
}
```

図 2: 典型的な非バイトコードインタープリター

抽象構文木を巡回する典型的な非バイトコードインタープリターによる処理は、バイトコードインタープリターによる処理に比べて実行時間やプログラムのサイズという点で不利である。しかし、バイトコードコンパイラを作成する手間が省けるという実装の容易性から、現在でもプログラミング言語 Ruby 等でこのような非バイトコードインタープリターは用いられている。

図 2 は、「x + 3」を計算するために必要となる、典型的な非バイトコードインタープリターの実装の一部である。eval 関数は式 expr をとり、expr の型に応じて異なる処理を行う。

このインタープリターには、以下のようなオーバーヘッドが存在する。

- 式の型による switch
- 部分式を eval するための call/return
- 定数のロード

典型的な非バイトコードインタープリターは、移植性が高く、バイトコードインタープリター以上に作成や理解が容易だが、call/return のオーバーヘッドが大きいので処理は非常に遅い。また、バイト

コードインタープリターと同様に、式のオペランドに対応する処理が単純であるほど、全体の計算時間に占めるオーバーヘッドは大きくなる。

2.3 JIT Compiler の有用性

インタープリターには以上で述べたように、多くのオーバーヘッドが存在する。そのため、計算処理の速度は遅い。この問題を解決するために、多くのインタープリターには、評価される式 (またはバイトコード) を実行時にネイティブコードに置き換える JIT コンパイラが組み込まれている。

JIT コンパイラの実装方法は様々であるが、一般的にはアセンブリや機械語に関する深い知識を要求する。そのような知識は、インタープリターの実装に用いる知識とは異なるため、JIT コンパイラの作成は一般的には困難である。また、アセンブリや機械語を直接用いた実装はマシン依存であるため、移植性が低いという問題もある。

3 JIT コンパイラ作成フレームワーク

2 章で述べたように、典型的なインタープリターには本質的なオーバーヘッドが存在する。このオーバーヘッドは、基本的に、どのような言語を実装するインタープリターにも存在している。そこで、我々はそのような本質的オーバーヘッドを除去することを目的とした単純な JIT コンパイラを作成するためのフレームワークを提案する。このフレームワークの特徴は、C 言語のみを用いて JIT コンパイラを作成できる点と、JIT コンパイラが (制限はあるが) マシン非依存である点である。

3.1 このフレームワークで作成できる JIT コンパイラの例

このフレームワークを用いれば、インタープリターの本質的オーバーヘッドを除去する JIT コンパイラを作成することができる。このフレームワークで作成した JIT コンパイラを用いて、図 1 のインタープリターを「x + 3」に対応するバイトコードに対して特化し、本質的なオーバーヘッドを除去したものが図 3 である。(JIT コンパイラによって生成される実際のコードはネイティブコードであるが、わかりやすく C 言語で表記した)

```
*++stackPointer = getVarFromSymbol(x);
*++stackPointer = 3;
--stackPointer;
*stackPointer += stackPointer[1];
```

図 3: x + 3 を実行する最適化されたコード

バイトコードの評価を行う実装が直列化されたことで、instructionPointer のインクリメント、メモリからの次のバイトコードのロード、オペコードによる switch、case 文から戻るジャンプを省略している。また、2 行目では、stackPointer が指すメモリに定数を直接ストアしている。このような定数のストアは、ネイティブコードでは即値命令で表現される。図 1 のインタープリターでは、メモリから「3」をロードし、それをストアするという一連の処理を行う必要があったが、図 3 では即値命令でストアするだけである。これによって、定数のロードが省略された。

図 2 の非バイトコードインタープリターも、グローバル変数 stackPointer を追加し、このフレームワークで JIT コンパイラを作成することで、図 3 のようなコードを生成することができる。

3.2 フレームワークの利用法

フレームワークのユーザー (JIT コンパイラの開発者) は、**テンプレート**と、(テンプレートの) **組み立て情報生成コード**を共に C 言語で記述することによって JIT コンパイラを作成する。インタープリターは JIT コンパイル時に、式の表現、あるいはバイトコードを組み立て情報生成コードに渡し、組み立て情報を生成する。そして、その組み立て情報をフレームワークが提供する最適化コード生成器に渡すことで最適化されたコードを取得・実行することができる。最適化コード生成器は、組み立て情報に登録されている情報を元に、テンプレートからマシン非依存な最適化コードを生成する。

ここでは、フレームワークの利用法を説明する例として、図 1 のインタープリターに組み込む JIT コンパイラの作成を考える。「x + 3」に対応するバイトコードが入力として与えられた場合に、図 3 のような最適化コードを生成できればよい。

```
int* stackPointer;

void code_PUSH_CONST() {
  int  ret_val;
  HOLE(code_PUSH_CONST_hole, ret_val);
  *++stackPointer = ret_val;
}

void code_PUSH_GLOBAL_VAR() {
  char* symbol;
  HOLE(code_PUSH_GLOBAL_VAR_hole, symbol);
  *++stackPointer =
   getVarFromSymbol(symbol);
}

void code_ADD() {
  LABEL(code_ADD_section);
  stackPointer--;
  *stackPointer = stackPointer[1];
}
```

図 4: テンプレート

図 3 は、図 1 の各々の case 内の実装を結合したものといえる。そこで、まずユーザーは、case 内の実装に対応するテンプレート (図 4) を記述する必要がある。図 1 の case PUSH_GLOBAL_VAR は、図 4 の code_PUSH_GLOBAL_VAR に、case PUSH_CONST は、図 4 の code_PUSH_CONST に、case ADD は、図 4 の code_ADD に対応している。

テンプレートの記述が完成したら、ユーザーは次にテンプレートを結合するための、組み立て情報生成コード (図 5) を記述する (詳細は後述)。組み立て情報生成コードは、図 1 のインタープリターの制御構造をそのまま借用すればよい。制御が case PUSH_GLOBAL_VAR の中に入った場合は、最適化コードの末尾に code_GLOBAL_VAR の実装を付加すればよい。この際、code_GLOBAL_VAR 内の symbol には、バイトコードから取得した変数名を埋め込む。制御が case PUSH_CONST の中に入った場合は、最適化コードの末尾に code_PUSH_CONST の実装を付加すればよい。この際、code_PUSH_CONST 内の ret_val には、バイトコードから取得した数値を埋め込む。制御が case ADD の中に入ったときは、最適化コードの末尾に code_ADD の実装を付加すればよい。

図 5 の組み立て情報生成コードを、バイトコードへの参照を引数にして呼び出せば、組み立て情報を取得できる。最後に、最適化コード生成器に組み立て情報を渡せば、最適化コードを取得することができる。具体的には、BlockCmpnt を引数にして、関数 generateCode を呼び出すだけでよい。

JIT コンパイラを実装する際は、まず、「どのタイミングで、どのコードを最適化」するかを考える。

```
compiled code:
unsigned char code[] = { ...,
  PUSH_GLOBAL_VAR,
  x,
  PUSH_CONST,
  3
  ADD, ...};

bytecode implementations:
BlockCompnt* analyze(unsigned char* inst_pter) {
 BlockCmpnt* cmpnt;
 for (;;) {
  unsigned char bytecode = *++inst_pter;
  switch (bytecode) {
  /* ... */
  case PUSH_GLOBAL_VAR:
    appendCmpnt(cmpnt,
                initByTemplate(v_code_PUSH_GLOBAL_VAR));
    addHoleInfo(cmpnt,
                v_code_PUSH_GLOBAL_VAR_hole,
                *++inst_pter);
    break;
  case PUSH_CONST:
    appendCmpnt(cmpnt,
                initByTemplate(v_code_PUSH_CONST));
    addHoleInfo(cmpnt,
                v_code_PUSH_CONST_hole,
                *++inst_pter);
    break;
  case ADD:
    appendCmpnt(cmpnt,
                 initByTemplate(v_code_ADD);
    break;
  /* ... */
  }
}
```

図 5: バイトコードインタープリターの組み立て情報生成コード

例としては、関数が呼び出される度に増加するカウンタを用意し、カウンタが一定の値を超えると JIT コンパイルを行うという方法が挙げられる。JIT コンパイルを行うタイミングと、最適化するコードが決定したら、インタープリター内の適切な場所で組み立て情報を生成し、generateCode() を呼ぶようにすればよい。最適化されたコードは、再利用するためにインタープリター内の適切なデータ構造に保存するとよい。

3.3 フレームワークの仕様と実装

テンプレートは、C 言語で書かれたコードの断片であり、無引数の void 関数内に記述する。この void 関数は、機械語列を作成するための材料であり、呼び出されることはない。また、関数名も JIT コンパイル時に識別するために必要なだけであるので、任意でよい。識別する際は、関数名の前に v_を付けた変数を用いる。なお、複数のテンプレートを横断する値にはグローバル変数を使用する。

これらのテンプレート内では、マクロ LABEL と HOLE が使われている。これらは gcc のプリプロ

セッサによってインラインアセンブリに変換される。LABEL は、アセンブリレベルのラベルに変換される。例えば、LABEL(label_name) は、__asm__ __volatile__ ("label_name:") に置き換えられる。このラベル情報は実行ファイル内に残り、JIT コンパイル時に参照することができる。本フレームワークはこの位置情報を利用してテンプレートを操作する。また、HOLE に関しては、第一引数がアセンブリレベルのラベルに変換され、第二引数はその名前の変数に任意の値を代入する即値命令に変換される。即値の部分には、テンプレートを組み立てる際に適切な値が代入される。例えば、アーキテクチャが IA32 の場合、HOLE(hole_name, var_name) は、__asm__ __volatile__ ("hole_name:"); __asm__ __volatile__ ("movl $0x12345678, %0" : "=r" (var_name)) に置き換えられる。二つ目のインラインアセンブリコードは、変数 var_name に $0x12345678 という即値を代入するものである。即値は最適化コード生成時に置き換えられるので、$0x12345678 という数値には特に意味がない。このインラインアセンブリコードは IA32 でしか意味をなさないので、アーキテクチャごとに固有のインラインアセンブリコードを用意し、マクロの展開を切り替えられるようにする。

フレームワークが提供する最適化コード生成器は、構造体 BlockCmpnt が構成するデータ構造を辿ることで最適化コードを生成する。BlockCmpnt は、テンプレートへの参照、テンプレート内に埋め込む定数と埋め込む位置、テンプレート内に挟み込む他の BlockCmpnt の情報と挟み込む位置などを保持する。

フレームワークが提供する関数 initByTemplate は、引数で指定したテンプレートの情報を持つ BlockCmpnt を生成する。関数 addHoleInfo は、第一引数で指定した BlockCmpnt に、「第二引数で指定した位置に、第三引数の値を埋め込む」という情報を付加する。関数 appendCmpnt は第一引数で指定された BlockCmpnt のリストの最後に第二引数で指定した BlockCmpnt を追加する。

最適化コード生成器は、コンパイルされたテンプレートをコピーして並べる。このとき、最後尾以外のテンプレートの最後にある機械語レベルの return 命令は、コピーされない。また、この際に、定数を埋めたり、call 命令や jump 命令の相対オフセットを調節したりといった低レベルな操作を自動的に行う。テンプレート内に他のテンプレートが挿入される場合は、jump 命令の相対オフセットを、挿入されたテンプレートのサイズだけ調整する。jump 命令が short jump であるときは相対オフセットを増加させるだけでは対応できない場合もあるので、そのようなときはオペコードも自動的に変更を行う。このような低レベルな操作は、ネイティブコードを解釈する必要があるが、現時点ではオブジェクトファイル用のディスアセンブラである objdump の出力を解析することで対応している。

3.4 テンプレートレベルでの最適化

```
*++stackPointer = 2;
*++stackPointer = 3;
--stackPointer;
*stackPointer += stackPointer[1];
```

図 6: 2 + 3 を実行する最適化されたコード (テンプレートレベルでの最適化なし)

```
*++stackPointer = 5;
```

図 7: 2 + 3 を実行する最適化されたコード (テンプレートレベルでの最適化あり)

組み立て情報生成コードはバイトコード列あるいは抽象構文木をテンプレートレベルで最適化することができる。例えば、「2 + 3」を実行するコードを素直に組み立て情報に変換し、最適化コード生成器にかけると　図 6 のような最適化コードを得られる。この最適化コードはディスパッチのコストが削除されているが、まだ大きな最適化の余地が残っている。

図 7 は、図 6 に対しテンプレートレベルでの最適化を行ったものである。図 6 ではテンプレートを 3 つ使用しているが、図 7 ではそれを 1 つのテンプレートに置き換えている。フレームワークのユーザーが組み立て情報生成コードに最適化機構を導入することで、以上のようなテンプレートレベルでの大域的な最適化を行うことができる。

3.5 テンプレートを用いる利点

最適化コードの生成において、C 言語で書かれたテンプレートを用いることには様々な利点がある。

まず、アセンブリや機械語を直接記述する必要がないことから、それに関する知識がない開発者でもJIT コンパイラを作成することができる。また、JIT コンパイラの実装は制限はあるながらも、マシン非依存となる。

テンプレートはC 言語で記述されているので可読性が高く、gcc によって静的にコンパイルされるので、gcc の手続き内最適化がそのまま利用可能である。JIT コンパイラが JIT コンパイル時に行う処理は、テンプレートを結合し、一部の内容を書き換えるだけであるので、Register Tranfer Language(RTL) や 3 番地コードなどの中間表現を用いた実装よりも、コード生成のコストが小さい。

テンプレートを用いた場合の欠点としては、ソースコードがバイナリーに変換された際に情報を失ってしまうことで、複数のテンプレートを横断する高度な最適化を行いにくくなることが考えられる。しかし、テンプレートを用いた実装でも peephole optimazion に関しては行うことが可能である。

3.6 実現されたマシンの非依存性

本フレームワークは、マシンアーキテクチャの実装を隠蔽した API を提供しているので、それを用いて JIT コンパイラを作成するユーザーは、アーキテクチャを意識する必要がない。しかし、フレームワーク自体はアーキテクチャ固有の実装に対応する必要がある。フレームワーク内部のアーキテクチャによって可変の部分には、ディスアセンブラ、HOLE マクロ、即値命令の中に定数を埋め込む方法、jump や call の destination の調整方法がある。また、フレームワークの実装は gcc インラインアセンブリに依存しているため、gcc がそのアーキテクチャに対応している必要がある。

4 実験と評価

典型的な非バイトコードインタープリター用の JIT コンパイラを作成し、実験を行った。

```
BlockCmpnt* analyze_eval(List* expr) {
  BlockCmpnt* cmpnt;
  switch (testElementType(expr)) {
  /* ... */
  case NUMBER :
    cmpnt = initByTemplate(v_code_NUMBER);
    addHoleInfo(cmpnt,
                v_code_NUMBER_hole,
                GetIntegerElement(expr));
    return cmpnt;
  case SYMBOL :
    cmpnt = initByTemplate(v_code_SYMBOL);
    addHoleInfo(cmpnt,
                v_code_SYMBOL_hole,
                GetSymbolElement(expr));
    return cmpnt;
  return cmpnt;
  case PLUS :
    cmpnt = initByTemplate(v_code_PLUS);
    addInnerCmpnt(cmpnt,
                  analyze_eval(GetSecond(expr)),
                  v_code_PLUS_section);
    addInnerCmpnt(cmpnt,
                  analyze_eval(GetThird(expr)),
                  v_code_PLUS_section);
    return cmpnt;
  /* ... */
  }
}
```

図 8: 非バイトコードインタープリターの組み立て情報生成コード

4.1 典型的な非バイトコードインタープリターの最適化

図 8 は、図 2 の典型的な非バイトコードインタープリターに対して作成した組み立て情報生成コードである。テンプレートに関しては、図 9 を利用する。関数 addInnerCmpnt は、第一引数で指定した BlockCmpnt に、「第二引数で指定した位置に、第三引数のテンプレートを挿入する」いう情報を付加する。この例では code_PLUS_section に二つのテンプ

```
int* stackPointer;

void code_NUMBER() {
  int  ret_val;
  HOLE(code_NUMBER_hole, ret_val);
  *++stackPointer = ret_val;
}

void code_SYMBOL() {
  char* symbol;
  HOLE(code_SYMBOL_hole, symbol);
  *++stackPointer =
   getVarFromSymbol(symbol);
}

void code_PLUS() {
  LABEL(code_Plus_section);
  stackPointer--;
  *stackPointer = stackPointer[1];
}
```

図 9: 非バイトコードインタープリター用テンプレート

レートが挿入されているが、このような場合は、最初に挿入されたテンプレートが前に来る仕様にしている。

組み立て情報生成コードは、インタープリターの構造をそのまま流用することができ、記述するコードも少ないので実装は容易である。

4.2 実験結果

Intel Xeon CPU 3.06GHz x 2 (Memory 2GB)、Linux 2.6.17、gcc4.1.1、glibc 2.4 という実験環境で、前節で示したコードを元に実験を行った。時間を測定するに当たっては、1000 回実行して平均を計算した。また、gcc でインタープリターをコンパイルする際に最適化オプション O1、O2、O3 を付けた場合も測定している。表 1 は (+ x 3) を計算した実験結果である。なお、実行前に x は 3 に束縛している。

実験から、単純な式において、インタープリターのオーバーヘッドの除去によって最適化が可能なことを確認した。また、コードは省略するが、制御構造を含む複雑な式でも最適化は効果を示した (表 2)。

表 1: (+ x y) の実行結果

最適化オプション	O0	O1	O2	O3
実行時間 (JIT なし)[ns]	103	62	62	52
実行時間 (JIT あり)[ns]	31	27	27	27
JIT コンパイル時間 [μs]	139	142	142	106

表 2: (if (- x 1) (+ x y) (* x y)) の実行結果

最適化オプション	O0	O1	O2	O3
実行時間 (JIT なし)[ns]	642	363	335	264
実行時間 (JIT あり)[ns]	86	84	76	75
JIT コンパイル時間 [μs]	524	451	537	402

5 関連研究

direct threading with selective inlining[1] は、バイトコードインタープリターに対応するポータブルな JIT コンパイラを作成する方法を提案している。インタープリターの本質的なオーバーヘッドを削減すること及びポータブルな JIT コンパイラを少ない労力で開発できるようにすることを主眼にしている点は本研究に近いが、定数のロードについては効率のよいコードを生成できるという点で本フレームワークの方が高度な最適化を行える。また、この手法は非バイトコードインタープリターには適用できない。

Tempo[2] と DyC[3] は、C 言語用の Run-time specializer である。Tempo は、テンプレートを動的に結合することで、実行時定数に特化したネイティブコードを生成することができる。本研究よりも高い次元の記述でコードを生成することができるという長所はあるが、関数単位でしか特化できない。また、Tempo を用いた場合、JIT コンパイラはインタープリターから自動的に生成されるので、本フレームワークのように高度な最適化のためテンプレートレベルでの最適化を手動で組み込むということは困難である。

DyC は、Tempo より低レベルなコードの操作を目的としている点で本研究に近いが、コンパイラをマシンアーキテクチャに応じて高度に拡張することを必要としているので、高いポータビリティを確保するのは困難である。

参考文献

[1] Piumarta, I. and Riccardi, F.: Optimizing direct threaded code by selective inlining. In *Proceedings of Programming Language Design and Implementation*, pp. 291-300 (1998).

[2] Consel, C. and Noel, F.: A general approach for run-time specializatin and its application to C. In *Proceeding of the 23th Annual Symposium on Principles of Programming Languages*, pp. 145-156 (1996).

[3] Auslander, J., Philipose, M., Chambers, C., Eggers, S. J., and Bershad, B.: Fast, effective dynamic compilation. In *Conference on Programming Language Design and Implementation*, pp. 149-159 (1996).