計算機アーキテクチャ 145−7
（２００１．１１．２９）

オブジェクトキャッシュを用いた JVM命令互換プロセッサの設計

近 千秋 † 清水 尚彦 ††
東海大学 大学院 工学研究科 † 東海大学電子情報学部コミュニケーション工学科 ††
259−1292 神奈川県平塚市北金目 1117
E-mail:{1aepm024, nshimizu}@keyaki.cc.u-tokai.ac.jp

概要 組み込みシステムで Java を用いようという動きが高まっている。組み込み分野の Java の実装においては、実行速度、リソースの制限といった事柄が問題となってくる。これらの問題の解決策のひとつとして、Java 仮想マシン命令をハードウェア実行する Java プロセッサを用いる方法がある。我々の研究室では、1995 年から独自の Java プロセッサ、TRAJA の開発と研究を続けてきた。この TRAJA プロセッサに、オブジェクトキャッシュ機構を追加する。このオブジェクトキャッシュ機構を実装することで、Java 実行環境に必要な RAM 容量を低減させることが可能である。本稿では、この”オブジェクトキャッシュ”機構の仕様と効果について報告をする。
キーワード：Java, Java プロセッサ, 組み込み Java

The Design of a Java Processor with the Object Cache

Chiaki Kon † Naohiko Shimizu ††
Graduated School of Engineering, Univ. TOKAI †
Dept. Communication Engineering, Univ. TOKAI ††
E-mail:{1aepm024, nshimizu}@keyaki.cc.u-tokai.ac.jp

Abstract We have continued development and research of our original Java processor, named TRAJA, since 1995. We implement the ”Object Cache” system to the TRAJA processor. It is possible to reduce the required RAM capacity for Java execution environment with the ”Object Cache”. In this paper, we described the design of the Java processor with the ”Object Cache”.
Keyword: Java, Java Processor, Embedded Java

1 はじめに

情報家電、携帯情報端末、工業機器といった組み込み機器分野で、Java を用いようという動きが活発になっている。これは、Java の持つプラットホーム非依存性、安全性、ネットワークとの相性の良さといった特長を、組み込みシステムの上でも活かしたいという流れから来るものである。一般的に Java はその実行速度に問題があるとされてきたが、JIT コンパイラなどの方法で改善することが可能である。しかしながら、組み込みシステムへの Java 実行環境の実装は、いくつかの組み込み分野固有の問題を抱えている。この固有の問題により、サーバ環境、デスクトップ環境の Java では実装されている解決策も、組み込みシステム上では実装が難しい場合が多数ある。

組み込み機器が抱える問題のひとつとして、ハードウェア資源の制限が挙げられる。組み込み機器では、消費電力、発熱、機器のサイズなどの理由により、CPU の能力、メモリ容量が押さえられる傾向

にある。このような環境の中で、Java の実行を改善する方法のひとつとして挙げられるのが、Java プロセッサを用いる方法である。一般に Java プロセッサは、Java 仮想マシン命令を直接実行、または最適なネイティブコードに変換することにより実行時間の高速化を計り、最適な設計をすることで消費電力の低減も期待できる。我々の研究室では、1995 年以来 TRAJA と呼ばれる Java プロセッサの開発、研究を続けてきた [1] [2]。TRAJA プロセッサは、Java 仮想マシン命令をハードウェアにより直接実行できるパイプラインプロセッサである。この TRAJA プロセッサに、我々が”オブジェクトキャッシュ”と呼ぶ機構を追加することで、組み込み Java のひとつの問題であった、実行時のメモリ使用量を低減することが可能になる。本報告では、オブジェクトキャッシュに要求される仕様、オブジェクトキャッシュを用いることで期待される効果について論じていく。

2 Java 実行環境の問題点

組み込みシステム上に Java の実行環境を実現する際に問題となる点の中から、以下に示すものに着目した。

実行速度

Java は通常 Java 仮想マシンと呼ばれるソフトウェア機構によって、バイトコードと呼ばれる Java 仮想マシン命令を逐次実行することにより動作する。このため、ネイティブコードで実行されるプログラムより、多くの処理時間を必要とする。解決策として、バイトコードを最適なネイティブコードに変換してプログラムを実行する方法がある。これらの方法は、JIT コンパイラ、動的コンパイラなどに代表される。この方法を用いることで、実行速度を大幅に改善することが出来る。しかしながら、これらの方法は本質的に Java プログラムの処理と関係のないプログラムがメモリに存在することになる。JIT コンパイラをハードウェアで実装する方法もあるが、コンパイラの出力するネイティブコードはバイトコードに比べ数倍の大きさになり、メモリの消費量が格段に増える。

メモリ容量の制限

サーバ環境、デスクトップ環境においては、使用できるメモリの量は比較的大きくとることができるため大きな問題にはならないが、組み込み機器においては消費電力、ハードウェア実装サイズといった理由から、使用可能なメモリ量は制限される。後述の _quick 命令のようなものへの対応するためには、バイトコードを含むクラス全体をインスタンス生成時にコピーする必要が生じ、一般の OS で行われていたコード共有が出来ないためメモリ要求量が増加する。

参照の解決を行う命令の扱い

Java 仮想マシン命令の中には、その命令の動作中に参照の解決を行なわなければならない命令がある。Java の参照は初期状態では文字列を用いたシンボリック参照である。一度使用された参照はダイナミック参照に置き換わり、二度目の実行からは高速な処理が行われる。静的な参照はクラスのロード時に解決が可能であるが、動的な参照は実行時に決定されなければならない。解決された命令を、最初に実行したときとは異なるルーチンで処理することで、二度目の実行からは高速な処理を行える。図 1 に簡単なフローチャートを示す。バイトコードがデコードされ、参照の解決が必要である命令である場合、すでに参照が解決されているのかどうかを何らかの方法で判定し、解決されていないのならば解決ルーチンへ、解決されてるのであれば、それに応じた実行ルーチンへ処理が移される。

実装例として、初期の Sun Microsystems の Java 仮想マシンにおける _quick 擬似命令がある [5]。この方法では、_quick 擬似命令が適用される命令が最初に実行されるとき、その命令を対応する _quick 擬似命令で置き換えることで解決された命令と、解決されていない命令の区別をする。

実行速度の問題については、TRAJA プロセッサにより Java 仮想マシン命令を直接実行することで解決を行う。プロセッサが十分な処理性能を持っていれば、この問題を解決することが期待できる。メモリ容量と、参照の解決を行い命令について、TRAJA プロセッサにオブジェクトキャッシュを付加するこ

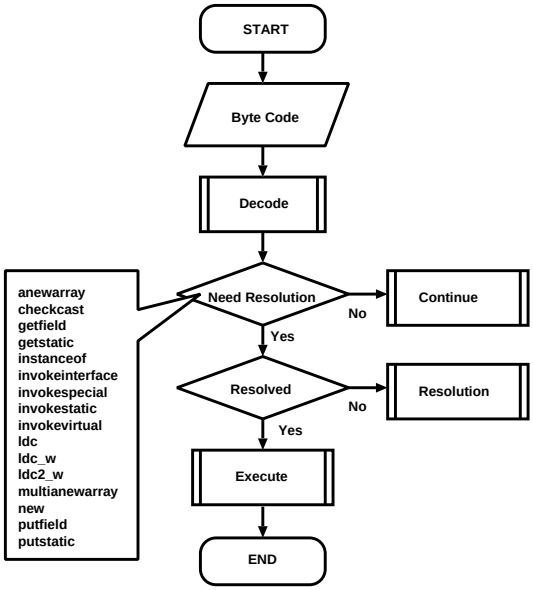

図 1: 参照の解決が必要な命令

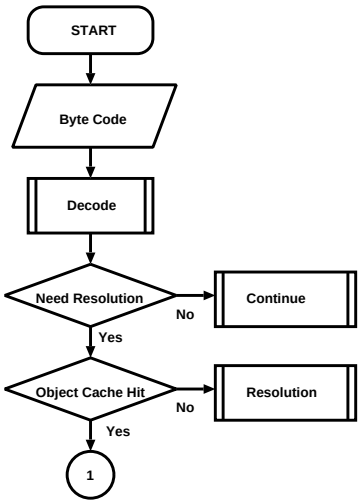

図 2: キャッシュ上のデータの有効性

とで解決を行う。オブジェクトキャッシュを用いて、解決の行われた命令と行われていない命令の区別をする。オブジェクトキャッシュを用いることで、Java の実行に必要なメモリ容量が低減できると考えている。次の節からオブジェクトキャッシュについて説明をしていく。

3 オブジェクトキャッシュ

オブジェクトキャッシュは、参照の解決を必要とする命令の処理を行うための、ハードウェアによるサポート機構である。次のような方針でオブジェクトキャッシュを設計していく。

1. 参照の解決を行った命令に関する情報を保持し、これらの命令の実行時には処理の流れのディスパッチャとして機能する。
2. 参照の解決が行われた命令が実行時に必要なデータを保持する。
3. バイトコード配列の書き換え、また新たな命令列の生成は行わない。

図 2 から図 4 に、命令処理のフローチャートを示す。図 2 において、バイトコードがデコードされ該当する命令であることが判断されると、オブジェクトキャッシュ上のデータの有効性をチェックする。キャッシュヒットしなかった場合は、参照の解決ルーチンに処理が移され、キャッシュヒットした場合は、オブジェクトキャッシュ上のデータに基づいて以後の命令処理を行っていく

図 3 では、まず命令が静的な参照を扱うか、動的な参照を扱うかを判定する。静的な参照の場合は、命令の対象は一意に決まるため `objectref` の一致判定を行う必要はない。命令動作はデコード情報より決定できる。フィールド、及び定数へのアクセス命令の場合、該当データのアドレスとデータ型が判明すればデータへのアクセスが可能である。メソッド呼出し命令の場合、まず該当するオブジェクトのモニタを獲得すべきかどうかが判断される。モニタの獲得が必要だと判断された場合、`monitorenter` 命令で該当するオブジェクトのモニタを獲得しなければならない。メソッドを呼び出す準備が整った後、新たなメソッドフレームを生成し、プログラムカウンタを呼び出したメソッドのバイトコード配列の先頭アドレスにセットする。新たなオブジェクトの生成、オブジェクトのチェック命令には、クラス情報を保持する構造体へのポインタを渡し処理を続ける。

図 4 には動的な参照を扱う場合を示す。命令が動的な参照を扱っている場合、スタックに積まれている `objectreference` が、参照の解決を行った

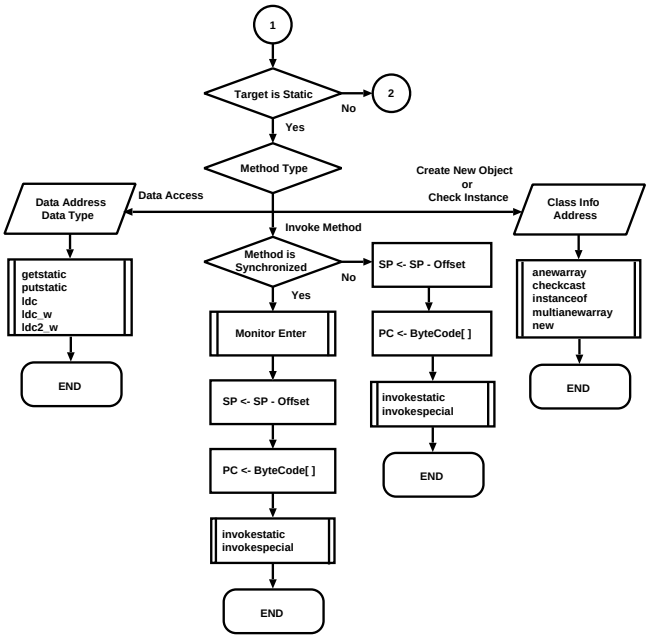

図 3: 静的参照の処理

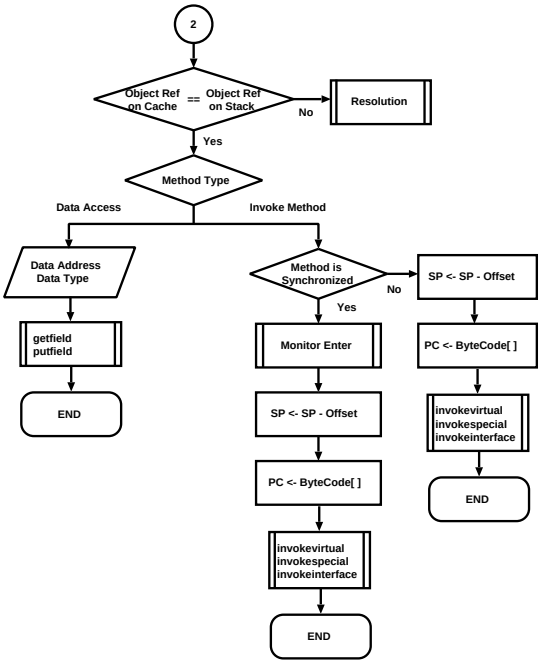

図 4: 動的参照の処理

objectreference と同一であるかを判定する必要がある。このため参照の解決が行われたオブジェクトを識別するデータが必要になる。メソッド呼出し命令では、引数の下に積まれている objectreference をスタック上から取り出すために引数の数が必要になる。その後の処理は静的な参照を扱う場合と同様に、モニタの獲得の必要性を調べ、メソッドを呼び出す準備が整ったところでプログラムカウンタを呼び出すメソッドのバイトコード配列の先頭にセットする。

以上の考察とフローチャートより、参照の解決が行われた命令を実行するため次のようなデータが用意されなければならないことが判る。

1. エントリの有効ビット
2. 静的 / 動的参照の判別ビット
3. 動的参照の場合
   (a) 解決されたオブジェクトの識別情報
4. フィールド、定数アクセス命令の場合
   (a) データのアドレス
   (b) データ型情報
5. メソッド呼び出し命令の場合
   (a) モニタ獲得の必要性の判別ビット
   (b) スタックポインタのオフセット (フレーム生成のため)
   (c) バイトコード配列の先頭アドレス
6. オブジェクトの生成、チェック命令の場合
   (a) クラス情報構造体のアドレス

これらのデータの中で、1,2,3(a),5(a) はその後の命令処理のために必須であり、必ずオブジェクトキャッシュ内に保持しておかなくてはならない。。静的なデータにアクセスする場合、4(b) が判明していれば、その後の処理を高速に行うことが可能である。そこで、4(b) もオブジェクトキャッシュ内に保持する。命令が静的なデータを処理するとき、objectreference の一致判定は必要なく、各アドレス情報は一意に決定することが出来る。そこで、静的なデータを処理するときには objectreference の代わりに直アドレス情報を保持する。ただし、モニタの獲得を必要とするときは静的に生成されたオブジェクトの objectreference が必要である。動的なデータを処理する場合は、直アドレス情報の代わりにオブジェクト構造中のメソッド配列、フィールド配列へのインデックスでアクセスを行う。以上をまとめて、オブジェクトキャッシュが保持するデータをまとめたものが図 5 である。図の下に書かれた数字がその情報のために必要なビット数を表す。

参照の解決が行われたエントリに固有なデータは、そのデータを持つオブジェクトと、コンスタン

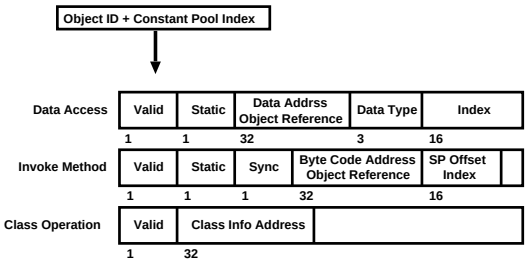

図 5: オブジェクトキャッシュ内のデータ

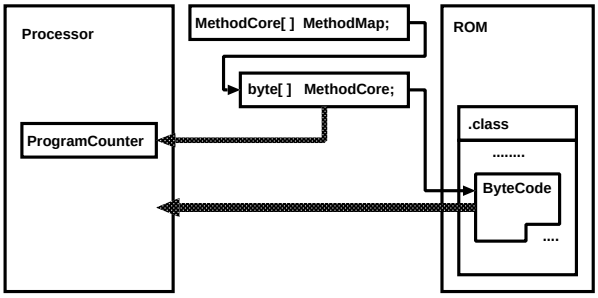

図 6: ROM からの命令実行

トプールのインデックスである。そこで、オブジェクトには各々Object IDという識別番号をつけ、この Object IDとコンスタントプールのインデックスを結合したものを、オブジェクトキャッシュ検索のための鍵とする。

4 RAM容量の低減

オブジェクトキャッシュを付加した Java プロセッサを用いることで、Java 実行環境に必要な RAM 容量を低減することが可能となる。命令実行は Java プロセッサが Java 仮想マシン命令を直接解釈、実行を行うので、JIT などが生成するネイティブコードを格納するための領域は必要ない。参照の解決が行われた命令に関する情報はオブジェクトキャッシュ内に保持し、オブジェクトキャッシュのヒット情報で参照の解決が行われたかどうかの判定が出来るため、参照の解決が行われた命令の書き換えを行わなくてもよい。従って、オブジェクトキャッシュを用いることで新たな命令列の生成は不必要となり命令のバイトコード配列は実行環境の中で静的なデータとなる。つまり、同じクラスから派生した各インスタンス間で、同一のコンスタントプールと、バイトコードを共有することが可能である。

この特長を利用することで、ROMからクラスファイルを読み出して実行するようなシステムでは、図 6 に示すように、ROM に格納された.class ファイル中のバイトコード列を、命令列として ROM から直接実行することができる。前述したとおり、オブジェクトキャッシュを用いればバイトコード配列は書き換えの必要がなく、かつ静的なデータとして存在することが可能であるため、メソッドの構造体の中から ROM の中のバイトコード列をバイトコード配列として参照することできる。実行時に必要になる静的なデータを ROM の中に展開して保持しておけば、実行に必要な RAM 容量を低減することが可能である。

これらのことから、オブジェクトキャッシュを用いることで、実行時に命令コードとコンスタントプールのために消費するメモリ容量を低減することが可能であり、低容量の RAM で Java を実行することが可能である。

5 まとめ

オブジェクトキャッシュの構成と、オブジェクトキャッシュを Java プロセッサに付加することにより期待できる効果について説明した。オブジェクトキャッシュは、Java の実行環境に必要な RAM 容量を低減することが可能であり、この特長はハードウェア資源が制限される組み込みシステムに適している。今後の予定として、オブジェクトキャッシュを用いたシステムの定量的な評価、オブジェクトキャッシュを実装した Java プロセッサの実現を行っていく。

参考文献

[1] http://shimizu-lab.et.u-tokai.ac.jp/

[2] 内藤 亮, 清水尚彦:”パルテノンによるパイプライン JAVA チップの設計” 第 12 回パルテノン研究会予稿集

[3] Jon Meyer, Troy Downing 著, 鷲見 豊訳:”Java バーチャルマシン” オライリージャ

パン

[4] Tim Lindholm, Frank Yellin 著, 村上雅章訳:”Java 仮想マシン仕様第 2 版” ピアソン・エデュケーション

[5] Tim Lindholm, Frank Yellin 著, 野崎裕子訳:”Java 仮想マシン仕様” アジソン・ウェスレィ

[6] 村山敏清 ”Java Press vol.17-20:Java 仮想マシン入門” 技術評論社