システムＬＳＩ設計技術　105－４
（２００２．３．４）

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　　Java 実行環境の持つ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注目されている。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は、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を目的としている。本稿では、オブジェクトキャッシュの設計をするにあたり行った、予備評価に関して報告をする。
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 †
School of Information Technology and Electronics, 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

組み込み機器分野において Java 実行環境を実現するために、Java プロセッサを用いる方法が注目されている。快適な Java の実行環境を実現するためには、CPU の能力、メモリ容量が必要になってくるが、組み込み機器分野では消費電力、発熱、機器のサイズなどの理由により、十分な能力を得ることは難しい。Java プロセッサを用いることで、これらの問題をまとめて解決できる可能性がある。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期待できる。今回、オブジェクトキャッシュを設計するにあたって、期待される必要メモリ容量の低減に関して、いくつかのベンチマークプログラムを用いて評価を行った。本報告では、まずオブジェクトキャッシュへ要求する事柄を、次にオブジェクトキャッシュの仕様について説明する。そして、我々が行ったオブジェクトキャッシュの評価について説明する。

2 オブジェクトキャッシュへの要求

Java 仮想マシン命令中には、図 1 に示されるような、実行時にオブジェクトへの参照の解決を行なわなければならない命令が存在する。この参照の解決は、具体的には、文字列によるシンボリックリンクを、ポインタによるダイナミックリンクに置き換えることである。最初にその命令を実行するときに参照の解決を行うことにより、2 回目以降のその命令の実行は、参照の解決を行わなくてよい分高速に行える。

ここで、参照の解決が行われる命令の 2 回目以降の実行のために、参照の解決が行われたかどうかを判別する必要がある。これを実現するために、Sun Microsystems の初期の Java 仮想マシンでは、_quick 擬似命令を用いていた。_quick 擬似命令は、最初に参照の解決が行われる命令が実行された際に、その命令コードを書き換えることで、参照の解決が行われた命令を判別することができるようにする。また、その命令が指し示すコンスタントプールエントリを、ダイナミックリンク、すなわちポインタに置き換えることで、命令実行を高速化する。

しかし、この方法で参照の解決を行う命令を扱おうとすると、生成されるオブジェクト毎に新たな命令コード列と、コンスタントプールを生成する必要がある。つまり、同一のクラスから生成されたオブジェクト間でも、命令コードとコンスタントプールの共有ができないことになり、ここで大きくメモリを消費することになってしまう。そこで、我々はオブジェクトキャッシュを用いることで、命令コード列と、コンスタントプールの共有ができるようにする。そのために、オブジェクトキャッシュには次に挙げるようなデータを保持しておく。

- 参照の解決が行われたかどうかを判定するためのデータ
- その後の処理の流れの振り分けをするためのデータ
- 参照の解決を行った命令が実行時に必要なデータ

このようなことを考慮し、オブジェクトキャッシュの仕様設計を行っていく。

3 オブジェクトキャッシュ

図 2 にオブジェクトキャッシュの概要を示す。オブジェクトキャッシュは、ダイレクトマップのキャッシュシステムとし、検索のキーには命令の直渡しオペランドとなる、コンスタントプールへのインデックス番号と、現在実行しているメソッドを持っているオブジェクトのオブジェクト ID 番号を用いる。コンスタントプールへのインデックス番号は、Java 仮想マシンの仕様として定まる 16bit のデータである。オブジェクトに付けるオブジェクト ID 番号は実装依存であり、これは 16bit のデータであると定めた。この 2 つを使って検索をかけ、データが存在する場合にはそのエントリを引いてくる。図 3 に、オブジェクトキャッシュ内に格納されているデータを示す。オブジェクトキャッシュ内のデータは、データ部とアドレス部と大別することができる。エントリ内のデータの有効性を示すために、1bit のバリッドビットが保持されている。参照の解決を行う命令は、大別すると 5 つの種類に分類することができる。この 5 つの種類毎に異なったデータが保持されている。それぞれについて説明する。

静的なデータの参照

`getstatic` 命令、`putstatic` 命令、`ldc` 命令、`ldc_w` 命令、`ldc2_w` 命令がこの種類にあたる。これらの命令は、メモリー中の静的なデータ、すなわち一意にアドレスの決まるデータにアクセスする。この場合、データにアクセスするため、そのデータのデータ型と、直接のアドレスを保持する必要がある。また、静的なデータであることを示すため、`static` フラグを 1 にしておく。

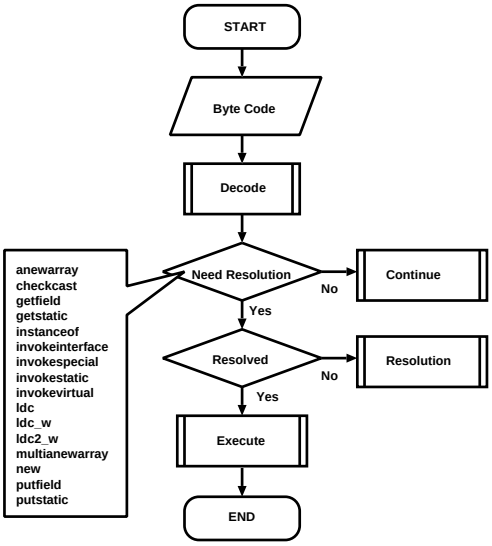

図 1: 参照の解決を行う命令

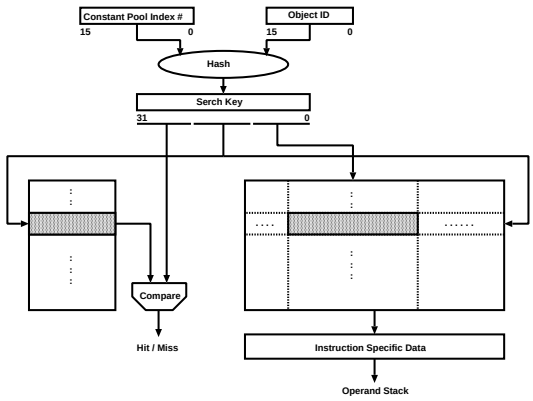

図 2: オブジェクトキャッシュ

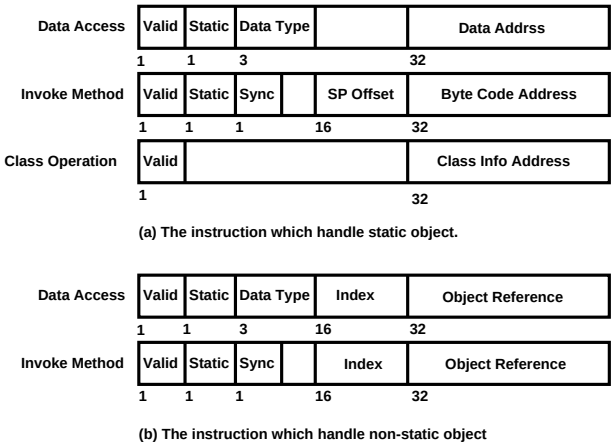

図 3: オブジェクトキャッシュ中のデータ

静的なメソッドの呼び出し

invokestatic 命令と、一部の invokespecial 命令がこの種類にあたる。<init> メソッド、private メソッドなどが、invokespecial で呼び出せる静的なメソッドである。静的なメソッド呼び出しでは、スタック上に積まれたオブジェクト参照と、既に解決されている参照が、同様のものであるかどうかを判定する必要が無い。呼び出すメソッドが synchronized である場合、スレッドはそのメソッドを持つオブジェクトのモニターを獲得する必要がある。このモニターの獲得の必要性を判定するために、キャッシュ内のデータには synchronized フラグを設ける必要がある。また、新たなスタックフレームの生成のために、現在のスタックポインタと、新たなスタックポインタのオフセット数を保持しておく。アドレス部には、メソッドの命令列の先頭アドレスを保持しておく。

クラスの参照

anewarray 命令、checkcast 命令、instanceof 命令、multianewarray 命令、new 命令がこの種類にあたる。これらの命令は、クラスの型の検査、新

たなオブジェクトの生成などを行うため、クラス構造体へのポインタを保持する。

動的なデータの参照

getfield 命令、putfield 命令がこの種類にあたる。この命令で扱うfieldは、オブジェクトとして動的に生成されていくため、オペランドスタックに積まれたオブジェクト参照と、すでに解決されたオブジェクト参照が同一のものであるか照合する必要がある。そのため、解決されたオブジェクト参照を、オブジェクトキャッシュ内に保持する必要がある。オブジェクト参照は、オブジェクトへのポインタであると考えられるため、fieldへのアクセスはインデックス番号を用いて行うことにする。また、動的なデータであることを示すため、static フラグを 0 にしておく。

動的なメソッドの呼び出し

invokevirtual 命令、invokeinterface 命令、一部のinvokespecial 命令がこの種類にあたる。オペランドスタック上のオブジェクト参照と照合するためのオブジェクト参照、モニタの獲得の必要性の有無を判別するための、synchronized フラグを保持する。命令列へのアクセスは、オブジェクト中の命令列へのポインタへのインデックス番号で行うこととする。

これらのデータが、キャッシュヒットした場合オペランドスタックへ積まれる、あるいはハードウェアによって直接解釈され処理を続行していく。

このようにオブジェクトキャッシュは、参照の解決が行われる命令をサポートする。参照の解決が行われる命令に関する個々のオブジェクト固有のデータは、全てオブジェクトキャッシュ内に含まれる。これにより、メソッドコード列とコンスタントプールを、同じクラスから生成されたオブジェクトの間で共有することが可能になり、新たなオブジェクトを生成する際に、メソッドコード列とコンスタントプールのコピーを行わなくてもよい。生成されるオブジェクトのサイズが小さくなることから、Java 実行環境の実行時に必要なヒープのサイズを低減することが期待できる。今回、オブジェクトキャッシュの導入によって得られる、メモリサイズへの効果について評価を行った。次の節で詳細に述べる。

4 オブジェクトキャッシュの効果に対する評価

実際に Java 実行時のメモリ容量に対し、オブジェクトキャッシュがどの程度の効果を持つかを、いくつかのベンチマークプログラムを例として評価を行った。今回用いたのは、LinpackLoopOpt[7]、DhryStoneBench[8]、そして SpecJVM98[9] によって提供される 10 個のプログラムの、計 12 個である。これらのプログラムに対して、次のような解析を行った。

1. クラスファイルを解析し、メソッドコードとコンスタントプールの大きさを調査する。また、参照の解決が行われる命令の数を調査し、消費されるオブジェクトキャッシュの大きさを算出する。
2. プログラムを実行し、動的に確保されるオブジェクトを調査する。

ここで、確保されるデータ量を算出するにあたり次の仮定をしている。

- 1 メソッドコードに対し、1 バイトのデータを必要とする。
- 1 コンスタントプールエントリに対し、4 バイトのデータを必要とする。
- 1 オブジェクトキャッシュエントリに対し、8 バイトのデータを必要とする。

この調査と仮定を元に、メソッドコードとコンスタントプールをコピーするために必要とされるデータ量と、オブジェクトキャッシュを導入した場合、オブジェクトキャッシュに要求されるデータ量を比較する。

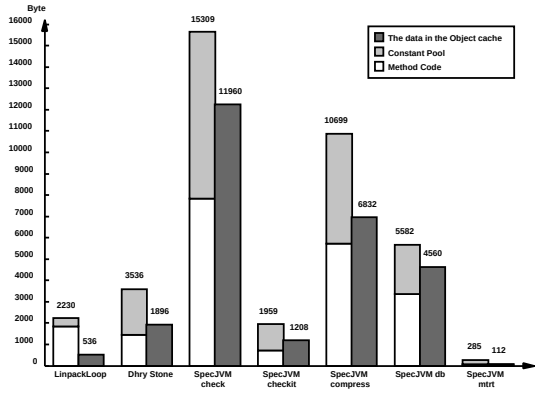

図 4: 静的に確保される領域　1

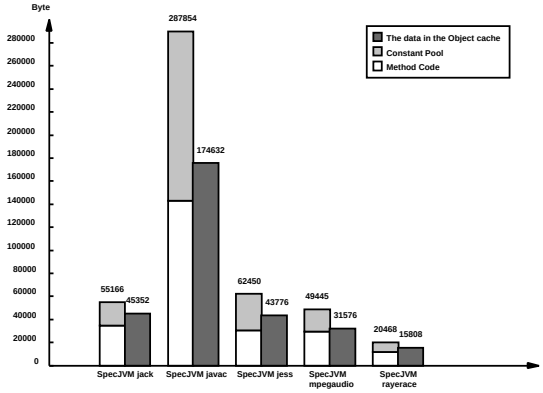

図 5: 静的に確保される領域　2

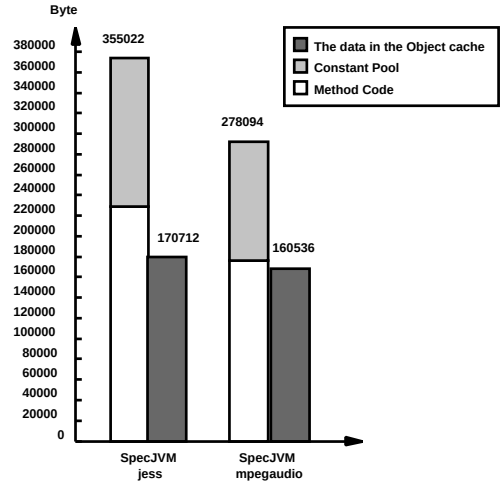

図 6: 動的に確保される領域　1

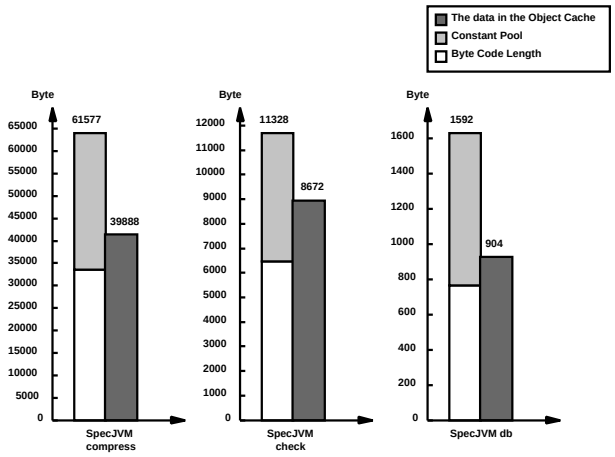

図 7: 動的に確保される領域　2

静的に確保されるデータ

まず、1 の調査をすることによって得られたデータを元に、それぞれの静的に確保されるデータ量を比較した (図 4、図 5)。グラフは、それぞれのプログラムのクラスファイルに含まれるメソッドコードとコンスタントプールのデータ量の合計と、命令列の中の参照の解決を行う命令 1 つにつき、オブジェクトキャッシュの 1 エントリを確保すると考えて算出したデータ量の合計とを比較しているグラフから読み取れるように、どのプログラムにおいても、オブジェクトキャッシュを用いた場合の方が、必要なデータ量が少なくなっている。これより、クラスファイルが静的に存在するシステムにおいては、オブジェクトキャッシュを導入することで、確保されるデータ量を低減することが出来ると言える。

動的に確保されるデータ

次に、2 の調査を行うことによって動的に確保されるデータ量を比較した (図 6、図 7)。なお、グラフに登場しないプログラムは、動的に確保するオブジェクトが無い、または動作中に例外が起こりプログラムがストップしたなどの理由により、測定ができなかったため除外してある。ここでは、プログラムが実行される中で、動的に生成されたオブジェクトが持つメソッドコードとコンスタントプールのデータ量の合計と、参照の解決を行う命令 1 つにつき、オブジェクトキャッシュの 1 エントリを確保すると考えて算出したデータ量の合計とを比較している。

グラフより、確保する必要があるデータ量が大きくなるほど、両者の差は大きくなっていくこと

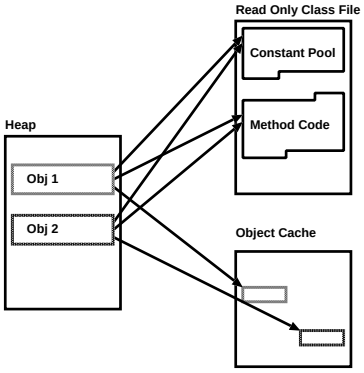

図 8: 例　1

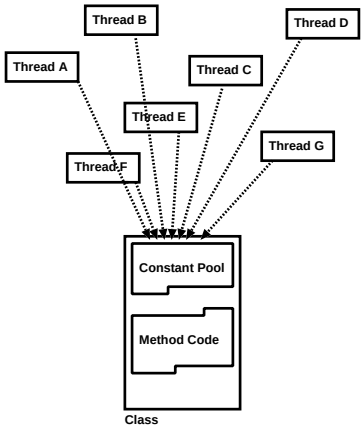

図 9: 例　2

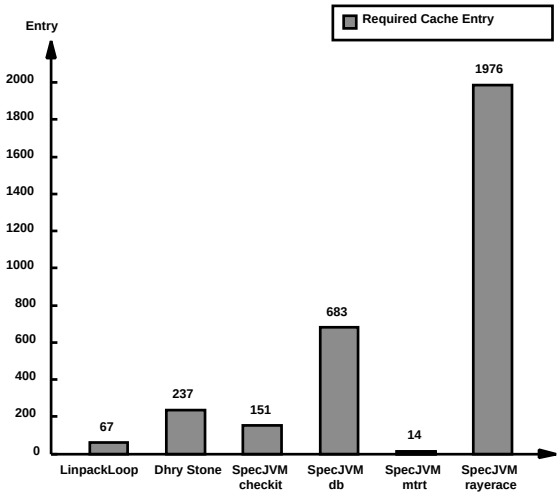

図 10: 必要なエントリ数　1

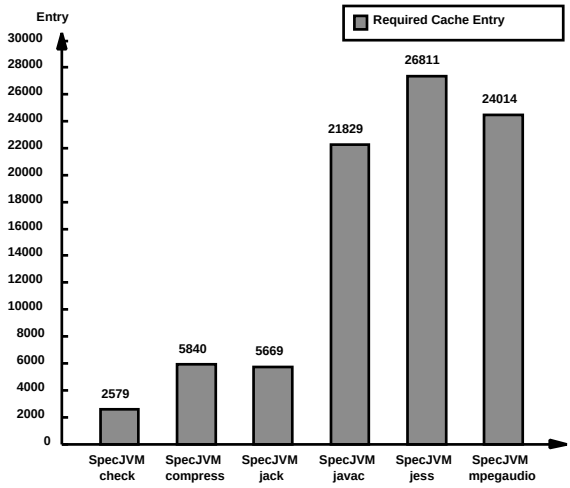

図 11: 必要なエントリ数　2

がわかる。これより、動的にみると、生成されるオブジェクトが多くなればなるほど、オブジェクトキャッシュの効果も大きくなっていくということが言える。

5　考察

前節で行った評価より、次のようなシステムでオブジェクトキャッシュの効果が期待できると考えられる。

1. 組み込みシステムで、図 8 のように、実行するクラスファイルを、ROM などによって静的に持っている場合を考える。オブジェクトキャッシュを用いることで、コンスタントプールとメソッドコード列の書き換えを行わなくてもよいため、この 2 つは RAM にロードせず、静的な領域からそのまま使用することができる。静的な評価の結果より、オブジェクトキャッシュの導入によって、必要な RAM の容量を低減することが期待できる。
2. 組み込みシステムで、図 9 のように、多くのスレッドが同じクラスのメソッドを実行する場合を考える。オブジェクトキャッシュの導入により、スレッド間でメソッドコード列とコンスタントプールの共有が可能なため、1 つのスレッドが持つオブジェクトは小さくなる。スレッドが増加しても、メモリの使用量を抑えることができる。

以上のように、前節で行った評価結果から、オブジェクトキャッシュを導入することで、実際に効果が期待できることがわかる。

6 まとめ

オブジェクトキャッシュについて説明し、その評価と、評価結果から考えられる効果について述べた。評価結果より、メソッドコードとコンスタントプールに必要とされるデータ量に対し、オブジェクトキャッシュに必要なデータ量は、静的に確保されるデータ領域では平均で 63%、最小で 24%のデータ量で、動的に確保されるデータ領域では平均で 61%、最小で 48%のデータ量で済むことがわかった。これより、Java 実行環境に必要なメモリ容量の低減という点で、オブジェクトキャッシュの導入で効果が期待できることがわかった。

オブジェクトキャッシュに必要なエントリに関して、4 節で行った評価より図 10、図 11 のような結果を得た。これは静的な解析で得られた必要なオブジェクトキャッシュのエントリ数に、動的な解析が行えたプログラムには、動的に確保されるオブジェクトキャッシュのエントリ数を合計したものである。これを見る限りは、必要なエントリ数にはばらつきがあり、必要十分なエントリ数を特定するのは困難である。一部のプログラムは、20000 以上ののエントリ数を必要としている。ただし、4 節で行った評価を考えると、オブジェクトキャッシュを用いた場合の方が明らかに必要なデータ量が少ない。必要なエントリ数が多い場合にも、オブジェクトキャッシュを RAM に実装すれば、効果は得られるものと考えられる。

今後は、実際にオブジェクトキャッシュを構成し、TRAJA に導入していく予定である。

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] 近 千秋, 清水尚彦:”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” 情処学会研報 ARC-145-7 2001

[4] 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

[5] 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

[6] 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

[7] http://www-2.cs.cmu.edu/~jch/java/optimization.html

[8] http://www.c-creators.co.jp/okayan/

[9] http://www.spec.org/