ハイパフォーマンス
コンピューティング 72－1
（1998. 8. 6）

ループ並列投機実行の Java 仮想マシンへの適用

美添一樹† 松本尚† 平木敬†

命令レベルより大きい粒度のブロックにプログラムを分割し、各ブロックを投機実行することによりブロックレベル並列性を得るハードウェアについて、いくつかの論文で提案がなされている。我々は投機実行の手法を適用した Java 仮想マシンを共有メモリマシン上で実装した。投機実行の対象はループに限定した。単純なループについて実験を行なった結果、インタプリタ Java 仮想マシンでも 10000 命令以上のループであれば高速化が可能であった。

Implementing Parallel Speculative Execution of Loops on JVM

KAZUKI YOSHIZOE ,† TAKASHI MATSUMOTO †
and KEI HIRAKI†

There have been several proposals about hardware speculative executions, in a larger granularity than instruction level parallelism, by partitioning the target program into blocks. We have applied speculative execution onto Java Virtual Machine. We implemented it on a shared memory machine. The target for speculative execution is limited to loops. We measured speedups for simple loops and found that it is possible to gain speedups for loops which contains more than 10000 instructions by an interpreter Java Virtual Machine.

1. はじめに

ブロックレベルでの投機実行による並列化は、2), 5), 6), 9) など多数提案されている。これらは基本的に On-Chip MIMD をベースとしたマルチプロセッサ上での提案であり、専用ハードウェアの支援を利用している。

図 1 に示したものが、投機実行のアウトラインである。

ブロックレベル並列化を行なう際にハードウェアの支援が最も必要とされる部分は、ブロック間のデータ依存関係の処理である。

データ依存関係には、大きく分けてレジスタを介するものとメモリを介するものの 2 種類が存在する。細粒度の投機実行では通常、レジスタを介したデータ依存関係のためには、内容を先のブロックへパスするための専用ハードウェアが用意される。図 1 において Processor Element 間の矢印がレジスタ間のデータの forwarding を示す。

メモリを介してのデータ依存関係を解消するためには、メモリアクセスの履歴を保持しておき、アドレスの照合を行なう必要がある。そのためには、例えば write は投機実行の際には実際には行なわれず、投機実行が成功したことを確認してから行なわれるようにするなどの工夫が必要である。提案ごとに詳細は異なるが、2), 6) ではメモリアクセス命令を記録するバッファを用意しており、5) では静的にメモリアクセスを解析しておき、データ依存関係を侵害する可能性のあるメモリアクセスだけをチェックするようになっている。

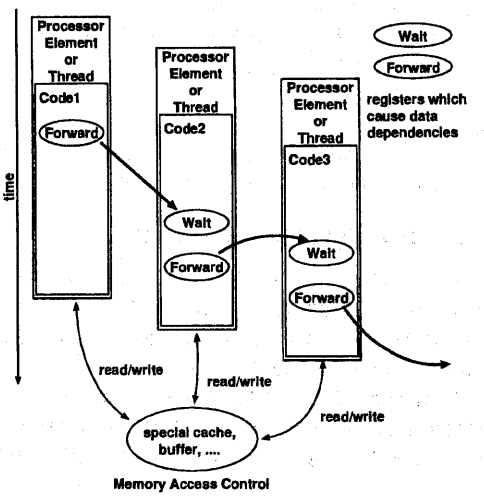

図 1 一般的な投機実行の例

† 東京大学 大学院 理学系研究科 情報科学専攻
Department of Information Science,Faculty of Science
University of Tokyo

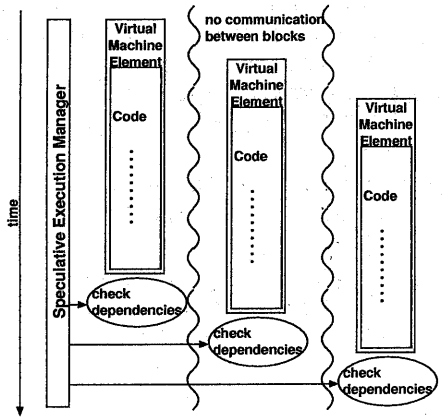

図 2　仮想マシンによる投機実行

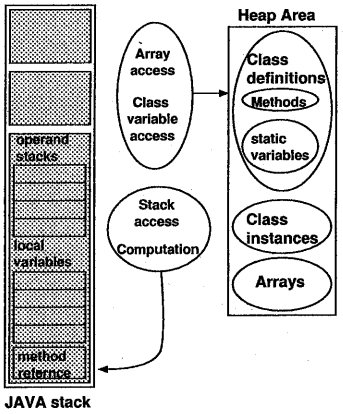

図 3　JVM の構造

本論文とより関連が深い研究としては 10) がある。これは、実行時動的再構成により動的にループを検出し、ループの各 iteration を並列に投機実行するものである。

これらの投機実行では、原理的にはブロックの実行中にデータ依存関係を解決するための通信を行なっていると言える。(実際にはキャッシュなどを用いる。) しかし、通信をソフトウェアで実現するにはハードウェアでの実現と比較して、投機実行のオーバーヘッドが大きくなると予想される。我々は投機実行ブロック間の通信を極力少なくすることによって、ソフトウェアで実装することによるオーバーヘッドを軽減することにした。投機実行の管理は図 2 に示したように、各ブロックの終了時点でのみ行なう。データ依存関係が検出された場合、ブロック全体を再実行する。このため投機実行を行なうブロック間でのデータ依存関係の頻度が十分小さくなければ高速化は期待できない。よって我々が実装した Java 仮想マシンによる投機実行では、投機実行の対象を粒度の大きいループに限定している。

我々のループ投機 Java 仮想マシンは、既存のクラスファイルを静的に解析して投機実行の対象とするループの情報を取得し、それを元にループを並列投機実行する。

なお、本実装の Java 仮想マシンは完全なインタプリタであり、JIT コンパイルの機能は搭載していない。

2. Java 仮想マシンのメモリ構造

Java 仮想マシン仕様[8] に基づき Java のアーキテクチャの中から投機的実行の実装と関係がある部分について解説をする。なお、以後 Java 仮想マシンを JVM (Java Virtual Machine) と呼ぶ。

JVM がランタイムに使用するメモリは大きく 2 つに分類される。Java スタックと呼ばれる局所変数と演算用のスタックの領域、および通常のメモリ空間と同等の存在と言える Heap 領域である。

図 3 に示されている Java スタック領域は、JVM の各 thread に 1 個ずつ存在する。この領域は、ポインタの存在しない Java の世界では自分以外のメソッドおよび他のスレッドから、完全に保護される。

実際の Heap 領域へのアクセスは、配列アクセス専用の命令とクラスのメンバ変数をアクセスするための命令だけによって行なわれる。スタック領域へのアクセスと Heap 領域へのアクセスが命令の種類によって明確に区別されていることにより、ハードウェアで投機的実行を実現する場合と比較して、メモリアクセスの履歴は小さいサイズですむ。

3. ループのプロファイル

我々は投機実行の対象となるループを静的に決定することにした。対象とするループに求める性質は以下の通りである。

- ループの誘導変数の変化が完全に、あるいは高確率で予測可能。
- メモリアクセスの頻度が低い。
- オペランドスタックを介してのデータ依存関係が発生しない。又は発生する確率が低い。
- Heap 領域を介してのデータ依存関係が発生しない。又は発生する確率が低い。

図 4 の外側のループは理想的である。このプログラムをコンパイルすると図 5 のようになる。

投機実行を行なうために必要な情報は以下の通り。

(1)　ループの誘導変数の値の変化。
(2)　ループ開始点及び終了点の指定。
(3)　投機実行の開始及び終了点の指定。

図 5 のように、ループの位置をプログラムカウンタ*の値で指定する。また、pc:24 の命令 `iinc 2 1` により局

* プログラムカウンタの値が連続していないのは、命令が可変長であるからである。

```
public class loop {
  public static void main(String args[]) {
    int k;
    for(int i=0;i<96;i++) {
      k = 0;
      for(int j=0;j<100000;j++) {
        k++;
      }
    }
    System.out.println("good bye");
  }
}
```

図 4　理想的なループ

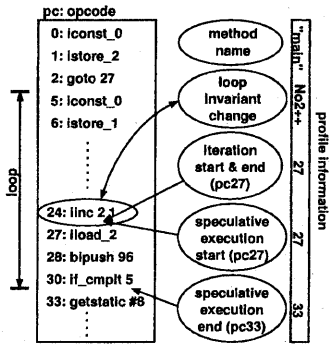

図 5　バイトコードとプロファイル情報

所変数の 2 番が 1 インクリメントされているので、これも指定する。

現在は人間の手でプロファイルを作成しているが、参考文献 1) にループの誘導変数の検出法が述べられている。

なお、誘導変数について、JVM が完全にソフトウェアで実現されていることを考慮すると、単純なインクリメント以外にも例えばリストの要素をたどるなどの複雑な操作も可能であるが、現在の実装では整数の誘導変数のインクリメント以外には対応していない。

4. 投機実行の管理

図 **6** に示したものが通常の JVM である。各スレッドが Java スタックとプログラムカウンタを持つ。

我々のループ投機 JVM は、通常の JVM のスレッド (図 6 の VM thread) に類似した Virtual Machine Element を複数用意して、ループの各 iteration を並列に投機実行するものである。アウトラインを図 **7** に示す。我々はこれを共有メモリマシン上で複数のスレッドを用いて実装した。

なお、用いたマシンは Sun Microsystems の SMP マシン、Enterprise 10000 である。

図 7 の中で Speculative Execution Manager (以後 SEM) と書かれている存在が、ループの投機実行を管理する。これは JVM 本体とは別個のスレッドである。SEM が担当する機能は以下の通りである。

(1)　投機実行モードの開始と終了。
(2)　プロファイル情報を元にループの各 iteration での誘導変数を生成し、各 VM element に提供。
(3)　各 iteration が終了した時点で誘導変数の予測の正否をチェック。
(4)　投機実行時のメモリアクセスの履歴を Speculative Access Buffer に保存し、各 iteration 終了時にデータ依存関係をチェック。
(5)　各 VM element とのコミュニケーション。

以下、SEM の機能について述べる。

4.1 投機実行の開始と終了

JVM は実行時にクラスファイルをロードするが、同時にプロファイル情報を含むファイルを検索する。ファイルの内容は以下の通りである。

(1)　投機実行を行なうループを含むメソッドの名称。
(2)　ループの誘導変数の値の変化。
(3)　ループ開始点及び終了点の指定。
(4)　投機実行の開始及び終了点の指定。

メソッドが実行される際に、投機実行を行なうループを含んでいることが分かると、我々の JVM はプログラムカウンタを監視しながら実行を行なう。プログラムカウンタが投機実行の開始点に達すると、SEM は JVM の現在の状態を保存し、それを元に投機実行を開始する。

投機実行を終了する条件は、以下の通りである。

(1)　プログラムカウンタが投機実行の終了点に達すること。
(2)　対象のループを含むメソッドからの return。
(3)　対象のループを含むメソッドでの uncaught exception の発生。

ある iteration で終了条件が満たされ、かつその iteration に先行する iteration が全て正常終了したことが確認されると、SEM は投機実行モードを終了し、通常実行モードへ移る。SEM は、投機的メモリアクセスの実行を確認し、ループの最後の iteration での状態を通常の JVM へ書き戻す。以後、通常の JVM として動作を続ける。

投機実行の開始と終了および各 iteration の開始と終了は、現在の実装ではプログラムカウンタを監視して行なっているが、これは当然実行時オーバーヘッドを伴う。当初は既存のバイトコード中に、投機実行用の命令を挿入して実現していたが、互換性を崩さないことを優先した結果、バイトコードの改変を必要としない方法に変更したためである。

4.2 ループの誘導変数の生成

プロファイル中には 1 iteration で、ループの誘導変数がどのように変化するかの予測が記されている。その

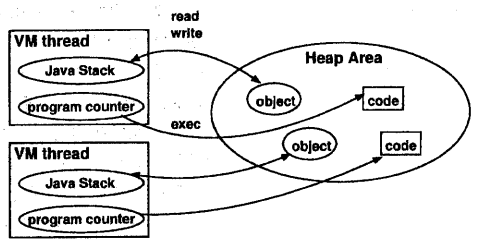

図 6　スレッドが複数存在する JVM

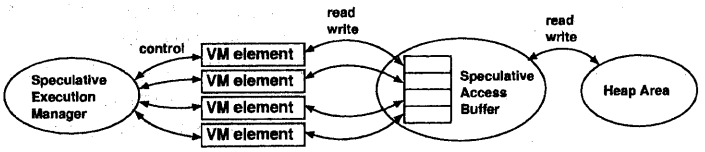

図 7　ループ投機 JVM

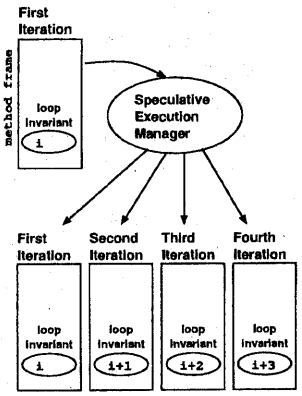

図 8　ループの誘導変数の生成

内容を元に、各 VM element が所有する局所変数、およびオペランドスタックの内容を生成し、それぞれの iteration を開始する。

現在の実装では、整数の誘導変数に対する定数インクリメントにのみ対応している。図 8 を参照。

4.3　誘導変数の予測のチェック

各 VM element へ提供した局所変数およびオペランドスタックの内容が正しく予測されているかどうか、実行時にチェックする。

ある iteration が終了した時点で、終了時の局所変数およびオペランドスタックの内容と、予測した内容を照合する。後続の iteration は予測に基づいて動作しているので、もし予測が外れていた場合には、以後の投機実行は失敗しているので、後続の iteration を実行中の VM element へ停止命令を送り、再度実行する。

4.4　メモリアクセスのチェック

データ依存関係が侵害されるのは、先行する iteration で write されたオブジェクトからの read が実行されていた場合である。ハードウェアでの投機実行の場合はブロック間での通信が可能であるので、図 9 の左側の例のように、実行時に時間的に write が先であれば、データ依存関係を解消することが可能である。しかし、我々の実装はソフトウェアによるものであるので、速度を考えると通信は極力抑える必要がある。そのため、メモリアクセスのチェックは iteration の終了時にのみ行なうようにして、基本的に投機実行中のブロック間通信を排除した。結果として図 9 の右側の例のように時間的に先に実行された write であってもデータ依存関係を侵害してしまうことになる。

投機実行中に Heap 領域へのメモリアクセス命令がされる場合には、write 命令は Speculative Access Buffer (以後 SAB) に履歴を取るにとどめ、実際に Heap 中のオブジェクトに write の結果を反映させるのは各 iteration の終了後である。履歴の内容は、実行された命令、命令への引数、および read または write された値、である。履歴の取り方は以下の通りである。

現在ある iteration の途中であり、あるオブジェクト O が Heap 領域上に存在するとする。

(1)　現在の iteration 中の、O からの最初の read を保存する。

(2)　現在の iteration 中の、常に O への最後の write の履歴も保存する。

(3)　write が起こった以後の O からの read は、最後に実行された write の履歴から値を得る。

ある iteration が終了した時点での操作は以下の通りである。

(1)　iteration が終了した時点でメモリアクセスの履歴をチェックする。データ依存関係の侵害がなければその時点で履歴にある write を実行する。

(2)　終了した iteration で実行された write は、後続の iteration にもその結果が反映されることが確実になるまで記録される。

以上のチェックを経てデータ依存関係の侵害が検出さ

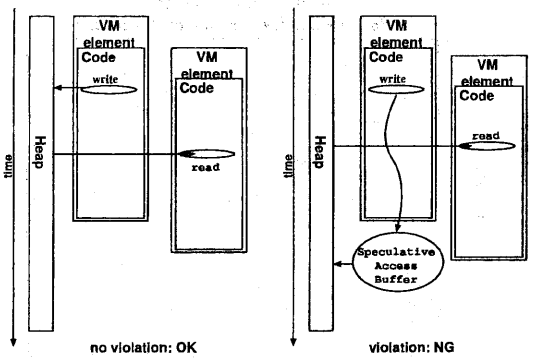

図 9 データ依存関係の侵害

れた場合には、終了した iteration も含め、実行中の全ての iteration を破棄する。また、後続の VM element には停止命令を送る。

通常 JVM はガーベジ・コレクションの機能を備えている。そのため、ガーベジ・コレクションの結果、同一のアドレスに同一のクラスのインスタンスがアロケートされる可能性があり、reference の値だけではオブジェクトを一意に決定できない場合もあり得る。しかし、我々の実装ではガーベジ・コレクションの機能はないため、reference の値によりオブジェクトを一意に決定できる。

4.5 各 VM element とのコミュニケーション

SEM は、誘導変数の生成が終了し iteration の実行を開始してよい状態になると実行開始コマンドを各 VM element へ送信する。各 VM element は iteration の実行が終了すると、1 iteration の終了か又は投機実行モードの終了を知らせるステータスを SEM へ送信する。SEM は投機実行モードの終了を示すステータスを受信すると、残りの VM element に向けて停止コマンドを送信する。

5. 結果と考察

2 種類のプログラムについて実行時間の計測を行なった。

- メモリアクセスの頻度がゼロである。
- メモリアクセスは 1 iteration につき、read 1 回 write 1 回。ただし、10 iteration に 1 回データ依存関係が発生する。

以上の 2 種類のループに対してループの部分のみの実行時間を測定し、性能向上を測定した。**図 10** は、ループ中の JVM インストラクションの数に応じた性能向上のグラフである。

予想されたことではあるが、命令数が 1000 程度のループではほとんど性能向上が見られない。これはソフトウェアで並列化していることを考えると当然のことと言える。特にメモリアクセスがない場合にも 1 iteration 1000 命令のグラフを見ると、1 より低いところから始まっている。これは VM element が 1 個であるにもかかわらず、複数存在する場合と同様の投機実行方式を無理矢理当てはめた結果である。

今回はメモリアクセスの頻度が高い場合の実験は行なっていないが、メモリアクセスの頻度が高ければ、性能向上はより小さくなると予想される。

6. 実装の問題点

我々が今回実装した JVM は、JVM インストラクションに従って動作している限りは、理論的には問題はない。しかし、ここで問題となるのが JVM においてもネイティブのメソッドが使用されることである。特に明らかな例は入出力である。仮に標準入出力が完全に Java のコードの中で行なわれるとすれば、標準出力もメモリアクセスとして管理できるので問題はない。しかし実際には System.out.print メソッドなどを呼ぶと、最終的にはネイティブのメソッドが実行される。

ループの内部で入出力が行なわれることに対する効果的な解決策は見い出せていないが、専用のライブラリを用意して入出力を監視し、バッファリングにより通常と同様に動作させることは可能である。しかしそのためには既存の入出力ライブラリ全てに対して新たなライブラリを定義する必要がある。本論文の我々の JVM ではこの問題に対する対策は取られておらず、今後の課題である。

7. 関連研究

先に述べたように、参考文献 2), 5), 6), 9) などで投機実行を行なうハードウェアの研究がなされている。また、参考文献 10) では動的にコードを分析することにより、ループの並列投機実行を行なうハードウェアの研究が述べられている。投機実行の対象がループに限定されている点で、ソフトウェアとハードウェアの相違はあるものの、本論文の内容と類似している。

JVM 自体の最適化については参考文献 3), 7), 8) などを参照されたい。

Sun Microsystems から発表されている Java HotSpot JIT コンパイラ[4] では、実行時に統計を取ることによりコードの中で重要な部分を検出し、重点的に最適化コンパイルをするようである。*

8. まとめ

以上の結果より、純粋なソフトウェアでも条件が整えば投機実行による高速化が可能であると言える。

今後の課題としては、実際のアプリケーションにおけるメモリアクセスについて調査し、現実に則した条件での実験を行なうことが挙げられる。また、現在の実装は

* 現在 (1998 年 7 月) 発売もされておらず、特許権の問題により詳細は不明である。

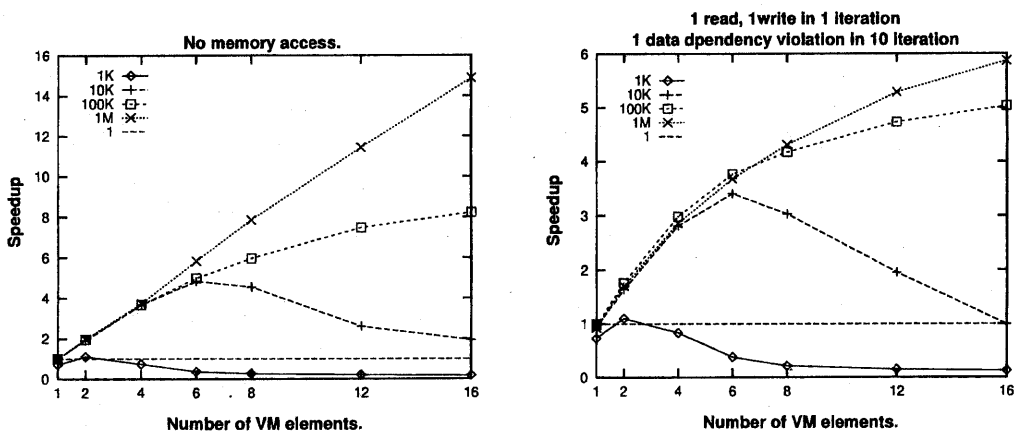

図10 性能向上の比較

単純なインタプリタであるので JIT コンパイラとの組合せなど、既存の JVM の最適化とのコンビネーションについての実験をする必要もある。

参考文献 8) によれば、Java のクラスファイル中にはカスタムの情報を attribute として自由に挿入しても良いとされており、特定の Java コンパイラ、Java 仮想マシンの組合せでのみ有効な最適化も許容されている。よって、現在の実装では既存のバイトコードに対して静的に行なっているループ情報の作成を、今後はソースコードのコンパイルの時点で行なうようにすることも可能である。さらには、Java コンパイラが JIT を支援するための情報を作成し、クラスファイル中に格納しておくことも可能である。

または参考文献 10) や HotSpot^{TM} 4) のように実行時に動的にループの解析をする方法なども考えられる。その際、並列性が高いと判明したループは、動的に並列度を上げるような操作も可能である。

謝辞 本論文の執筆に当たって、当研究室の私の先輩に当たる、玉造さんには投機実行について、詳細な教えをいただいた。ここに深く感謝の意を示しておきたい。

参 考 文 献

1) Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers Principles, Techniques, and Tools*, Addison-Wesley (1986).
2) G.Sohi, S.Breach and T.Vijaykumar: Multiscalar Processors, *proceedings of the 22nd Annual International Symposium on Computer Architecture* (1995).
3) Hsieh, C.-H. A., Gyllenhaal, J. C. and mei W. Hwu, W.: Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results, *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 90—99 (1996).
4) Java^{TM} HotSpot^{TM} VM Team: JDK Software Performance Technologies (1998). slides are available at www.javasoft.com.
5) J.Y.Tsai and P.C.Yew: The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation, *proceedings of Internatinal Conference on Parallel Architectures and Compilation Techniques* (1996).
6) K.Olukotun, B.Nayfeh, L.Hammond, K.Wilson and K.Y.Chang: The Case for a Single-Chip Multiprocessor, *proceedings of the 7th International Symposium on Architectural Support for Parallel Languages and Operating Systems* (1996).
7) Krall, A. and Grafl, R.: CACAO - A 64 bit JavaVM Just-in-Time Compiler, *PPoPP'97 Workshop on Java for Science and Engineering Computation* (Fox, G.C. and Li, W.(eds.)), Las Vegas, ACM (1997).
8) Londholm, T. and Yellin, F.: *The Java Virtual Machine Specification*, ADDISON-WESLEY ISBN 0-201-63452-X (1996). Also available at http://java.sun.com/docs/books/vmspec/.
9) 鳥居淳, 近藤真巳, 本村真人, 西直樹, 小長谷明彦: On Chip Multiprocessor 指向制御並列アーキテクチャ MUSCAT の提案, *proceedings of joint symposium on parallel processoin 1997* (1997).
10) 玉造潤史, 松本尚, 平木敬: Loop を並列実行するアーキテクチャ, 情報処理学会研究会報告計算機アーキテクチャ, Vol. 119, No. 11, pp. 61–66 (1996).