計算機アーキテクチャ 120－6
（1996．10．31）

細粒度スケジューリング方式研究のための オープンな評価システム

高木浩光†

細粒度並列処理においてオブジェクトコードの最適化は避けることのできないものである。最適化の対象としては命令パイプラインの最適化、レジスタ割り当ての最適化、機能ユニットへの割り当てと実行順序の最適化、通信スケジュールの最適化、同期スケジュールの最適化、各命令について投機実行をさせるべきか否かの最適化、条件実行すべきか/分岐すべきかの最適化、などがあり、これらは互いに複雑に影響しあっていてもはや発見的な手法の積み重ねによる経験によってしか解決は困難である。今こそこれらを統一的に同じ条件の下で評価するための枠組が必要である。本稿ではこのような細粒度最適化方式研究のためのオープンな評価システムについてその概略を紹介する。

Open Platform for Evaluation of Optimizing Methods on Fine-grained Parallel Execution

HIROMITSU TAKAGI†

For instruction-level parralel processing, the optimization of object code by a compiler is essential. So, many techniques of optimization such as pipleline scheduling, register allocation, task scheduling, communication scheduling, synchronization scheduling, speculative scheduling, etc. have been proposed. These techniques should be compared with each other under the common condition. So, in this paper, an open platform for evaluating these optimization techniques is presented.

1. はじめに

細粒度並列処理においてオブジェクトコードの最適化の必要性は避けることのできないものである。如何なる新しいアーキテクチャを考案しても、そのアイデアによってコード生成に新たな自由度が生ずる限り、新たな最適化の余地が生ずる。このとき、この最適化が十分になされたコードで評価を行なわなければそのアーキテクチャの正しい評価とはならない。

現在の命令レベル並列処理においては最適化の対象は多岐にわたっており、一般的にはこれには

- 命令パイプラインの最適化。
- レジスタ割り当ての最適化。
- 機能ユニットへの割り当てと実行順序の最適化。
- 通信スケジュールの最適化。
- 同期スケジュールの最適化。
- 各命令について投機実行をさせるべきか否かの最適化。
- 条件実行すべきか/分岐すべきかの最適化。

などが挙げられる。これらは互いに複雑に影響しあっており、もはや発見的な手法の積み重ねによる経験によってしか解決は困難と考えられる。

しかし、これらに関する研究報告の現状として以下の問題点があると考える。

- 特定のアーキテクチャのための事例報告にとどまっている。
- 特定の特殊なプログラムモデルの上での変換を前提としている。
- 論文中の記述からだけでは提案アルゴリズムの内容が曖昧で再現することが困難。
- 提案アルゴリズムの実装プログラムのソースコー

† 名古屋工業大学
Nagoya Institute of Technology

ドは公開されない場合が多い。

- ソースコードが公開されていても理解が困難で別の目的で使用することが困難。

新アーキテクチャの提案に付随して始まったコンパイラ研究ではそのアーキテクチャにしか適用できない議論に終っている場合がある。多くの場合、非巡回有向グラフを拡張した○○グラフが独自に定義されその上での議論となる。このグラフモデルがその特定のアーキテクチャに強く依存している場合、アーキテクチャ固有ではない一般的な最適化を対象としているにもかかわらず、議論内容は一般性を失ってしまっている場合がある。

アルゴリズムの内容が曖昧で再現が困難であるという問題は、本来論文としてこのような曖昧さは許されないものであると考えるが、アルゴリズムが複雑化している現状では、論文の誌面の有限性からやむを得ないところがある。この曖昧さを解決する一つの手段は、提案方式を実装したソフトウェアを公開することであるが、それがなされない場合が多い。また、ソフトウェアが公開される場合でも、コンパイラや実行シミュレータまで含めたシステム全体が公開された場合は、その中から必要な部分すなわち対象としている最適化アルゴリズムの部分だけを取り出すことが容易でないことがある。

理想的には、論文中では数式等を用いて簡潔にアルゴリズムを記述するとともに、実際に動作するソフトウェアを別途インターネットを介して公開することが望ましいと考える。このとき公開するソフトウェアが、提案アルゴリズムの部分だけに限定されるならば、ソフトウェアを公開したくない理由にコンパイラ全体を公開することは避けたいという動機があったならば、その部分は公開しなくて済むことから、公開を見送ることも少なくなるのではないかと考える。

しかし提案アルゴリズムの部分だけが取り出されて公開された場合は、そのアルゴリズムに対するインターフェイスが何らかの方法で提供される必要がある。しかもこの部分は全てのアルゴリズムに共通であった方が望ましい。このような背景から、このような細粒度最適化方式研究のためのオープンな評価システムの構築をすすめている。ここではそのシステムの概略を紹介する。

2. システムの特長

公開するアルゴリズムを記述するための言語としてJavaを選択した。これは以下の理由による。

- それなりのオブジェクト指向設計を可能としてくれる。
- それなりに一般に普及しつつある。
- ネットワークプログラミングのためのライブラリが充実している。

このような評価システムでは、提供されるアルゴリズムをコンポーネントとしてシステムに組み込むことが可能でなくてはならない。そのために、オブジェクト指向ソフトウェア構成技術を利用してこれを容易にする。また、公開されるアルゴリズムは、読んで理解が容易であるために、十分に高度に抽象化されていなくてはならない。そのためにもオブジェクト指向技術が有効である。一方で、このようなシステムをできるだけ多くの研究者に利用していただくためには、特定の特殊な言語のための知識を必要とするようであってはならない。Javaは、現在さまざまの理由から急速に普及しつつある言語であり、これを採用することにより、多くの方々に利用していただけるものにできると考える。

また、Javaのネットワークプログラミングための特性により、公開するアルゴリズムを、その公開者の手元に置いたまま、ネットワークを介して評価システムにかけることが可能になる。こうすることによって、アルゴリズムの著者がURLによって明示されることになり、より積極的にアルゴリズムを公開することを動機付けられると考える。

3. 入力プログラム記述モデル

評価システムへの入力となるサンプルプログラムをどのようなモデルで記述するかは、このシステムがよ

り一般的なものであるために重要である。本システムでは、まず最初の段階として、対象プログラムをループ構造を含まないものに限定した。したがって、実際のプログラムから最内ループのブロック (条件文を含んだ複数の基本ブロックは含んで構わない) を抽出して本システムへの入力とすることになる。ループ構造を含めた最適化については次の段階で検討するものとする。

もし条件文を含まない単一の基本ブロックのみを対象とするならば、プログラム記述モデルは単純である。命令 (タスク) をノードとした非巡回有向グラフによる記述で十分に一般的である。そのためこのような条件の下では、同じモデルの上で様々のスケジューリング方式が提案、比較検討されてきている。しかし、条件文を含んだ場合のモデルとなると、様々なモデルが提案されており、異なるモデル上での議論を公平に比較することを困難にする要因となっている。これまでに提案されている条件文を含むプログラム記述モデルは、以下の観点から分類できる。

- 基本ブロックを越えた単一代入化がなされているか否か。
- 条件処理が制御フローグラフによって示されているか、各命令にその実行される条件を示す情報が付加されて示されているか。

基本ブロックを越えた単一代入化とは、例えばSSA (Static Single Assignment) によって実現されるもので、条件値によってどの命令によって定義されたデータを参照するのかが変化するような場合を、ϕ function と呼ばれる疑似命令によって、ひとつの依存関係として表現可能にするものである。例えば図1 (a) のようなプログラムを (b) のように表現することにより、変数 a, b に対する代入が単一化されている。このような単一代入化がなされることは、最適化のフェーズにおいて次のような利点をもたらす。

- 基本ブロックを越えてレジスタ割り当てを最適化することを容易にする。
- ある条件において命令を実行するようにスケジュールするか実行しないようにスケジュールするかの選択 (基本ブロックを越えたコード移動に相当する) を自由に決定することができる。単一代入化がなされていない場合にはレジスタの再利用によって生じている「反フロー依存」関係によって制約を受けてしまう。

したがって、本システムでも入力プログラムの記述にSSA 形式を採用する。

条件文の記述方式については次のことがいえる。制御フローグラフを用いる手法よりも、各命令についてそれが実行される条件を記述される手法の方が、最適化フェーズにおいて問題を取り扱い易くする。プログラムにループ構造を含む場合には、制御フローグラフを用いることの方が自然であるかもしれないが、プログラムにループ構造が含まれない場合に限定すると、制御フローが表している情報は各命令が実行されるか否かの情報と同値であり、最適化フェーズにおいて直接的に必要とされるのは各命令が実行されるか否かという情報である。例えば図1(b) 右のプログラムにおいて、「 a2 = y; 」が実行されるならば z には a2、実行されないのならば a1 の値が与えられることが保証されるようスケジュールをすればよいことが直接的に判定できる。制御フローグラフで表現されている場合には、グラフの有向辺をたどることによってようやくこのことを判定できる。

```
a = x;                    if (w == 0) {
if (w == 0) {                 b = y;
    a = y;                } else {
}                             b = x;
z = a;                    }
                          z = b;
```

(a)

```
a1 = x;                   if (w == 0) {
if (w == 0) {                 b1 = y;
    a2 = y;               } else {
}                             b2 = x;
z = \phi(a1, a2);         }
                          z = \phi(b1, b2);
```

(b)

図1　SSA による基本ブロックを越えた単一代入化

また、制御フローグラフによるモデルは、分岐命令によって条件文が実装されるというプロセッサアーキテクチャを直接反映したものである。しかし、VLIWなどの命令レベル並列性の高いアーキテクチャにおいては条件実行 (命令の実行/非実行の条件が各命令に指示されている) の命令セットを持つものがあり、条件文は分岐命令と条件実行の組合せによって実装されることがある。これを制御フローグラフによるモデルで表現することは、グラフ構造の複雑な変換を定義する必要が生じ直接的でない。一方逆に、各命令の実行条件から分岐命令によるオブジェクトコードを生成するには、同じ実行条件を持つ命令をグルーピングすることによって変換可能である。そしてここで、グルーピングの組合せから適切なものを選択することによって、基本ブロックを越えたコード移動に相当する最適化を行うことができる。

SSA によりデータ依存関係を記述し、制御依存を命令の実行/非実行の条件によって記述するというモデルは、既に文献[2]などにおいて提案されているが、本システムでは制御依存の表現モデルとして、各命令の実行/非実行の条件によるものではなく、以下に提案する「データ参照の必要性条件」によって記述するモデルを採用する。

提案モデルは、SSA における ϕ function の部分を次のように解釈するものとする。例えば

```
z = φ(c1:c2:i, c1:!c2:j, !c1:k);
```

が与えられた場合は、z に代入されるべき値は、条件値 c1 が真かつ c2 が真ならば i、c1 が真かつ c2 が偽ならば j、c1 が偽ならば k の値となることが保証されなければならないことを意味する。図 2 の例で説明すると、(a) のソースが与えられたとき、提案モデルでは (c) のように記述される。(b) は命令の実行/非実行の条件を示すモデルで記述した場合である。「`!c: b2 = x`」は c が偽のときのみ「`b2 = x`」が実行されるということを意味している。(b) では (a) のソースプログラム中の条件文が、そのまま命令の実行条件に変換されている。これに対し、提案モデルの (c) では、各代入文がどのような条件で実行されるかといったことは直接的に記述していない。各命令の実行が必要であるかどうかは、z の右辺である ϕ function の内容によって決定される。例えば、まず z の値は必要であると仮定すると、c の値が真であると確定した時点で、a1 の値が必要であることを意味する。すなわち `a1 = x;` の実行が必要であることを意味する。ここで重要なのは、どの命令に対しても「実行しない」ということを規定していないことである。c の値が偽であっても `a1 = x;` を実行してもかまわない。また、c の値が未定の段階で `a1 = x;` や `a2 = y` を実行してもかまわないことを意味している。

```
a = x;                  if (w == 0) {
if (w == 0) {               b = y;
    a = y;              } else {
}                           b = x;
z = a;                  }
                        z = b;
```

(a) ソースプログラム

```
c = w == 0;             c = w == 0;
a1 = x;                 c: b1 = y;
c: a2 = y;             !c: b2 = x;
z = \phi(a1, a2);       z = \phi(b1, b2);
```

(b) 命令の実行条件 (predicate) による記述

```
c = w == 0;             c = w == 0;
a1 = x;                 b1 = y;
a2 = y;                 b2 = x;
z = \phi(c:a1, !c:a2); z = \phi(!c:b1, c:b2);
```

(c) データ参照の必要性条件による記述

図 2　制御依存情報の表現形式

提案モデルのこのような特性は、各命令を投機実行させるべきか否かの最適化を扱いやすくする。一般に「投機的命令移動」と呼ばれるものは、プログラムソース中に書かれた条件文の中にある命令を条件文の外へ移動させることを指しており、ちょうど図 2 (a) において右のプログラムを左に変換することがこれに相当するものである。しかし、提案モデルによってひとたびプログラムが記述されると、もはや「投機的命令移動」は意味をなさなくなる。これは、モデル上で命令の非実行を規定していないためであるが、このこ

とは、図2の (c) が右も左も全く同一であることからも理解できる。提案モデルで書かれたプログラムから実際のコードへの変換は次のように行なうことができる。全く投機実行をしない場合 (逐次プロセッサを対象とする場合など) では、依存関係にある ϕ function の引数に示されている条件値が確定した後の時刻に、条件が真になったデータを生成する命令の実行を開始可能とするようスケジュールする。逆にすべてを投機実行させるならば、ϕ function の引数とは無関係にすべての命令を実行可能としてスケジュールすればよい。実際の資源が有限であるプロセッサに対してコード生成する場合には、資源に余裕がある時刻には、ϕ function の引数の条件値が確定する前のタイミングでスケジュール可能とし、余裕がない時刻には、条件値が確定して実行が必要であることが確定した時刻でスケジュール可能とするようにすればよい。この柔軟性は、命令の移動による最適化方式に比べ、他の最適化とくにレジスタ使用の最適化や分岐/条件実行の最適化と同時に組み合わせて最適化を行なうことを容易にする。

4. 評価システムのユーザインターフェイス

評価システムのユーザインターフェイスとして Java Applet も用意する。Applet には図4に示すように、入力プログラムのソースを URL で入力したり、また評価するアルゴリズムを、コンパイルした class ファイルの URL によって指定することができる。Applet は、実行時に、入力された URL から評価対象のアルゴリズムの class ファイルをネットワーク経由で取り寄せ☆、これを呼び出し、入力プログラムも URL で指定されたものをネットワーク経由で取り寄せ、これを用いて評価のための計算を行なう。発案したアルゴリズムをこのシステムを用いて評価するためには、本システムの API に従ってスケジューラクラスを作成し、これをどこかの Web サーバに置き、その URL をこの Applet に入力して実行させるという手順をふむことになる。

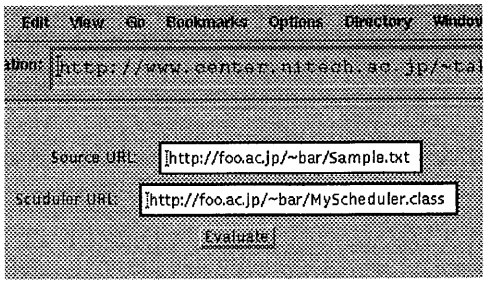

図3 評価システムの Java Applet 版

5. おわりに

本システムはまだ完成しておらず、本稿ではその URL を公開することはできなかった。本来ならば API について詳しく説明すべきであったが、間に合わず、概略のみの紹介となってしまった。完成したらこれを広く一般に公開する予定である。

今後はシステムを完成させるとともに、参考文献に挙げた論文に示されている各種最適化アルゴリズムをクラスライブラリ化することを検討している。これによって、同じ条件の下で、様々な最適化アルゴリズムを比較することができ、さらなる改良のための分析に役立てることができると考えている。

また、これを利用したスケジューリングコンテストを開催することなども検討している。

参考文献

1) 本多, 水野, 笠原, 成田: Fortran プログラム基本ブロックの並列処理手法, 電子情報通信学会論文誌, Vol. J73-D-1, No. 9, pp.756–766, (1990).
2) 小松, 小関, 深澤: 命令レベル並列アーキテクチャのための大域的コードスケジューリング技法, 情報処理学会論文誌, Vol. 37, No. 6, pp.1149–1161 (1996).
3) Fisher, J.A.: Trace Scheduling: A Technique for Global Microcode Compaction, IEEE Trans. Computers, Vol. C-30, No. 7, pp.478–490 (1981).
4) Bernstein, D. and Rodetch, M.: Global Instruction Scheduling for Superscalar Machines,

☆ Applet からネットワーク経由でクラスをロードするために ClassLoader を使用することは security の制約からできないが、http://www.center.nitech.ac.jp/~takagi/java/NetworkClassLoader/ に示されているトリックによってこれを可能にしている。

Proc. the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, pp.95–105 (1991).
5) Callahan, D. and Koblenz, B.: Register Allocation via Hierarchical Graph Coloring, Proc. the SIGLAN'91 Conference on Programming Language Design and Implementation, pp.192–203 (1991).
6) Chow, F. and Hennessy, J.: Register Allocation by Priority-based coloring, Proc. SIGPLAN'84 Symposium on Compiler Construction, (1984).
7) Chaitin, G., Auslander, M., Chandra, A. K., Cocke. J., Hopkins, M. E. and Markstein, P. W.: Register Allocation via Coloring, Computer Languages, pp.47–57 (1981).
8) Cytron, R., Ferrante, J., Rosen, B. and Wegman, M.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, ACM Transactions on Programming Languages and Systems, Vol. 13, No. 4, pp.451–490 (1991)
9) Gibbons, P. B. and Muchnick, S. S.: Efficient Instrucition Scheduling for a Pipelined Architecture, Proc. SIGPLAN '86, Symposium on Compiler Construction, (1986).
10) Goodman, J. R., and Hsu, W.: Code Scheduling and Register Allocation in Large Basic Blocks, Supercomputing '88, pp.442–452, (1988).
11) Hennessy, J. L., and Gross, T.: Postpass Code Optimization of Pipeline Constraints, ACM Transactions on Programming Languages and Systems, Vol. 5, No. 3, pp.442–448 (1983).
12) Gross, T. R. and Hennessy, J. L: Optimizing Delayed Branches, Proc. IEEE MICRO-15, (1982).
13) 小松, 神力, 小関, 深澤: 命令レベル並列アーキテクチャのためのレジスタ割り付け技法, 情報処理学会論文誌, Vol. 37, No. 6, pp.1149–1161 (1996).
14) 小松, 小関, 百瀬, 深澤: 命令レベル並列プロセッサのためのコードスケジューリングおよびレジスタ割付けの協調技法, 並列処理シンポジウム JSPP '96, pp.73–80 (1996).
15) 高木, 有田, 曽和：実行タイミングの動的変動に強い静的プロセッサスケジューリングアルゴリズム, 電子情報通信学会技術研究報告, CPSY90-85, PP. 23-29 (1990).
16) 高木, 有田, 曽和: 重複可能なバリア型同期のためのスケジューリングアルゴリズムとその性能, SWoPP '91 並列/分散/協調処理に関する「大沼」サマー・ワークショップ (1991).
17) 高木, 有田, 川口, 曽和: バリアを唯一の同期手段とした場合のタスクスケジューリング, 電子情報通信学会技術研究報告, CPSY93-22, PP. 73-80 (1993).