計算機アーキテクチャ 108－6
（１９９４．１０．２７）

ここいらで、計算機アーキテクチャを再考しよう

田中 英彦

東京大学工学部

計算機性能の進歩は著しい。一つのプロセッサ性能が今や、300MHz、500MFLOPS に迄達し、並列マシンでは、1TFLOPS に達する勢いである。しかし、現在の計算機アーキテクチャはこのままでよいのであろうか。、並列処理、分散処理は基本的な要求であるし、パーソナル、マルチメディアも重要な要求である。これらを満たす計算機アーキテクチャを今一度基本から考え直してみたい。本文では、その一つの試みを行なう。

Reconsideration of Computer Architecture

Hidehiko Tanaka

University of Tokyo

The progress of computer performance is marvelous. The clock frequency of micro processor reaches 300MHz and the performance is 500MFLOPS. The peak performance of parallel machine is about 1TFLOPS. However, we need to reconsider the computer architecture when many kinds of requirements arise such as parallel processing, distributed processing, personal processing and multi-media processing. This paper discusses the computer architecture from the fundamental mechanism.

1 はじめに

コンピュータの発達は著しい。単一のマイクロプロセッサの性能は、今や、300MHz、500MFLOPSに達しているし、並列マシンの性能は、1TFLOPSに達しつつある。しかしながら、あらゆる問題に対してこの性能をユニバーサルに達成することは難しい。これらの性能は、科学技術計算等、定型的は数値計算にのみ可能な性能であって、汎用の計算を対象とはしていないことに注意する必要がある。

しかし、コンピュータの応用は著しく拡張しており、画像・音声・データを含むマルチメディア処理、文字列処理、ニューロ計算モデル、学習、多量知識処理、等今後の発展が期待されている分野が多く出現している。これらの分野では、今までに増して広くコンピュータが使われるとともに、大規模な処理能力が要求される分野も多い。確かに、500MFLOPSは、過去のマシンを考えれば高速ではあるが、これで満足とは言い難い。高度な科学技術を支える新しい設計法を考えるとき、従来マシンとは桁違いに高性能なマシンに対する要求には切実なものがある。

また、今後の処理は、通信と融合した処理となるのが一つの特徴で、さまざまなマシン間の協調が不可欠の要素である。従って、コンピュータも単独ではあり得ず、マシンインタフェースの標準化が求められる。

さて、このように情報処理を考えたとき、現在のコンピュータは、21世紀をしょって立つに十分なものであろうか。コンピュータアーキテクチャに絞って考えてみれば、今はやりは、RISCであり、また、命令レベルの並列処理を組み込んだSuperscaler技術であり、そのようなマイクロプロセッサを数多く組み合わせた並列マシンである。

この種のアーキテクチャは確かに優れた特徴を持ち、今まで性能を発揮してきた。しかし、これ以上の並列度を目指したSuperscalerは、そろそろ限界であるし、それが将来の並列マシンに向いたものであるかは甚だ疑問である。過去、コンピュータアーキテクチャの分野では、キャッシュ、仮想メモリ、データフローマシン、並列推論マシン、等優れた基本技術の積み重ねがある。これらを広く検討し、あらゆるレベルの並列性を効率的に引き出す柔軟なアーキテクチャを提案することは、正に現在求められていることではなかろうか。

この文では、そのような意識で、コンピュータ処理を基本から考え直す試みをしてみようと思う。それが、すぐ新しいアーキテクチャを発見することにならなくとも、考え直す契機にはなりそうだからである。

2 計算の進め方

コンピュータアーキテクチャを考える前に、まず、一般的に計算の進め方について考える。それを具体化する為に、πの計算を例に取って、考える。

2.1 アルゴリズムの決定

$$\pi = 4 \times \int_0^1 \frac{1}{1+t^2} dt$$

2.2 積分公式の決定

Simpson公式を使うことにすれば、

$$Sum = \frac{h}{3}\left\{ f(0) + 4\sum_{j=1}^{n} f((2j-1)h) + 2\sum_{j=1}^{n-1} f(2jh) + f(1) \right\}$$

但し、

$$h = \frac{1}{2n}, \qquad f(x) = \frac{4}{1+x^2}$$

2.3 使用コンピュータの決定とそれに応じた最適化

2.3.1 単一プロセッサでの最適化

例を挙げれば次のような処理が行なわれる。

- 関数 $f(x)$ のループ内への埋め込みとレジスタ割り当て

$$f(0) - f(1) + \sum_{j=1}^{n} \{4f((2j-1)h) + 2f(2jh)\}$$

これをプログラム化するのに、変数として、 x, j, x2, x3, delta, sum, n を設定し、それを用いたアセンブラプログラムを生成する。その内、 $4f((2j-1)h)$ の計算部分のみを書くと以下の通りである。

```
    x ← h
    j ← 1
l1: x2 ← x * x
    x3 ← 1 + x2
    delta ← 4 / x3
    sum ← sum + delta
    j ← j + 1
    x ← x + 2h
    if (j ≤ n) goto l1
    halt
```

- 二つのループの融合
  $4f((2j-1)h)$ と $2f(2jh)$ の計算を融合させて二つのループを一つにし、その間の無駄 (j の更新、条件判断) を省くことが出来る。

- 条件分岐の最適化
  条件分岐の予測等により、ループの先頭部分を先取りすることが出来る場合は、パイプラインハザードを避けることが出来るし、 j と x の更新を if 文と位置交換して無駄を省ける。

- プログラム変換による最適化
  これらのプログラム内で出現する変数は、アルゴリズムの記述で現れたもの j, n, h と、新たに追加されたもの

    x, x2, x3, delta1, sum, y2, y3, delta2

  とに分類される。アルゴリズムを固定すれば、変数 j, n, h は given であるが、他の変数は、処理の都合で導入したものである。主記憶に置いてもよいし、一時変数としてレジスタをアサインしてもよい。即ち、これらのアサインは、処理を効率化する為の自由度である。

  更に、変数 j, n, h についても、もしプログラム自体を変換等によって変更することが可能であれば、これらの変数を主記憶に設けて、その存在を最適化の制約条件とすることの必要性は無くなる。元々の要求は、π の値を求めることのみであり、最終結果を変えないという条件のみ

を守る最適化を考えれば、これらもレジスタ等にアサインすることが可能となる。この最適化では、プログラムの構成をかなり大幅に変更し、より優れた効率良いプログラムを見い出すことが期待出来る。

2.3.2 ベクトルプロセッサの割り当て

ベクトルプロセッサでは、ループをパイプラインに割り当てるが、n の値が余り小さくない場合は、ループを幾つかに分割して、複数のパイプラインに割り当てることが行なわれる。この場合、一時変数

x2, x3, delta1, y2, y3, delta2

は、ベクトルレジスタに割り当てられ、主記憶には存在しない。

2.3.3 並列コンピュータにおける処理割り当て

$4or2f(jh)$ の計算を No.j processor に割り当てることが考えられる。すべての結果を加算する為のプロセッサ間通信を実現するには、一つのプロセッサに集める方法や、小計を分散して作成し集める手法がある。

2.4 望ましいプロセッサアーキテクチャ

上に述べたような様々な最適化を行ない易く、命令レベルからプロセスレベルに至る様々な並列性を取り込むのに効率的でフレキシブルなアーキテクチャが望まれる。単一プロセッサの単体としての効率を向上させることは重要であるが、その余り、並列処理を実現するオーバヘッドが大きくなり過ぎるものは問題であろう。今後の実時間処理の重要性を考えたとき、キャッシュ等完全な制御が難しい構造は再考する必要があるように思われる。基本機能をしっかりと備えた、単純で、拡張性に富む構造が欲しいのではないか。

3 データフローグラフと処理ハードウエア

3.1 理想処理装置

プログラミング言語で書かれたプログラムをデータフローグラフに変換すれば、それが処理の流れを表す本質的な部分であって、計算途中結果をレジスタや主記憶に取るというような「処理の都合上付け加えた」部分は見えなくなる。従って、このグラフを最も効率良く処理することを考えればよい。

即ち、理想的に考えれば、データフローグラフに出現する各演算要素、+、−、*、/、and、or、not、shift をハードウエア要素と考え、それらがデータを流す配線によって結ばれたハードウエアが存在し、それにデータを流すことで処理結果を得る図1のような装置があれば、最適であろう。

この装置では、データを流す配線による遅延時間は自然考慮され、適当に同期を取って処理が進むが、処理を時間的に明示化するには、各配線の時間遅延を、一種の遅延回路を挿入することによって表すことにすればよい。加算器等の各ハードウエア要素についても、その入力データが揃ってから結果を出力する迄の遅延時間を添付しておけば明示化される。

また、このような装置は、データフローグラフに従って作成されるため、コンパイラがハードウエアの構造を指示し、それに従って、配線を張り全体の装置を作ればよい。

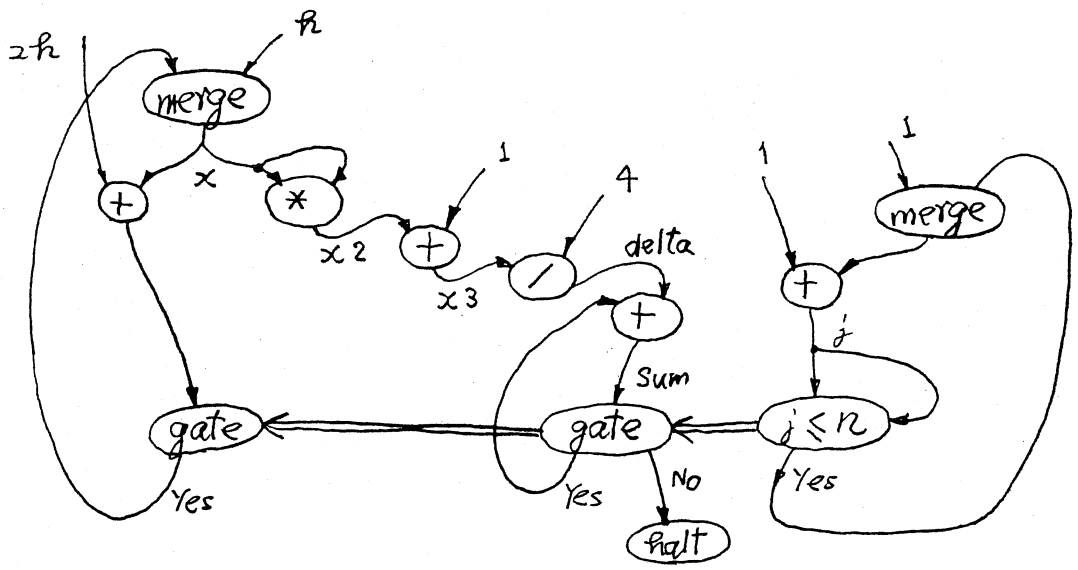

図 1: 理想的処理装置

3.2 現実回路への折り畳み

しかし、実際のデータフローグラフは、非常に大きく、それをそのままハードウエアで用意するのは大変である。従って、データフローグラフを適当なサイズに分割し、それ毎にハードウエアを用意して処理し、図2のようにそれらの間をメモリによって結合することが必要となる。

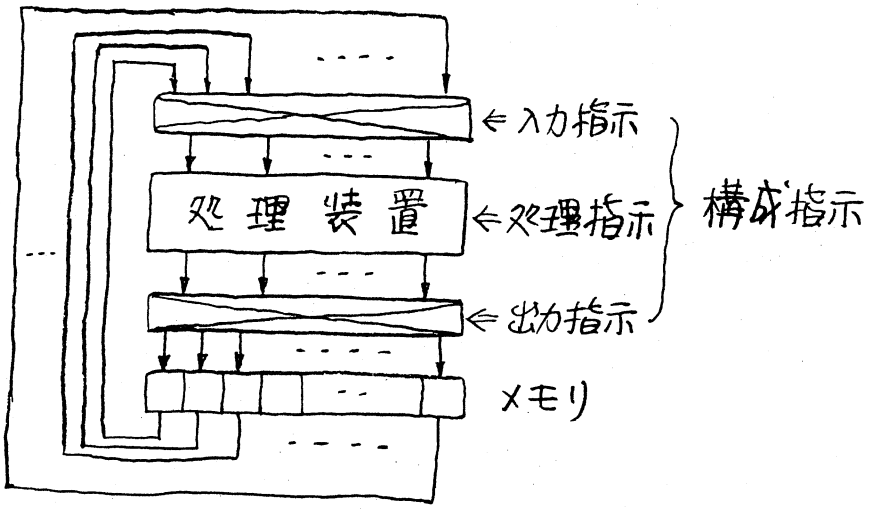

図 2: 現実的な処理装置

図中、構成指示は、その分割された部分的なデータフローグラフを解釈・デコードすることによって定められる。構成指示は、3種類に分けられ、入力指示、処理指示、出力指示からなる。入力指示は、メモリ内のデータのどれを各演算が使うかを定めるもの、処理指示は、演算要素を組み合わせる形の指示、出力指示は、各演算結果をメモリのどこにしまうべきかを指定する。

このような形態が有効であるためには、データフローグラフの分割をうまく行なう必要がある。すなわち、元の大きなデータフローグラフは、その各部が適当に発火し、データを流すが、その時間タイミングに応じてグラフを分割する必要がある。前述のように、各部の処理時間を設定することによって、その各部が発火する時間と結果を出力する時間を定めることが出来る。この結果を解析し、ある時間間隔

$t \sim t+T$

の間に処理が行なわれるものを集めて、その時間の処理内容とする。

しかし、一般の処理には条件分岐があり、その分岐先は、その時点の条件が定まらないと決まらない。従って、分岐先の処理に着目すれば、その

処理をする必要がある。

又は、

処理は不必要である。

のどちらかであって、処理をする必要がある場合には、その開始時刻がその分岐処理時間から推定できる。従って、基本的には、あらゆる分岐を解析することにより、すべての部分処理の開始時刻を推定できる筈であるが、実際には、多くの分岐が組み合わされるので、全体のグラフには、処理が行なわれない部分が多く含まれ、その同定は、実行時にしか可能ではない。

よって、構成指示の一部には、動的な解析が含まれ、前の処理結果に基づき条件分岐の不確定さを可能な限り確定し、次の処理を定めて行く必要がある。従って、構成指示は、図3のような機構により行なわれる。

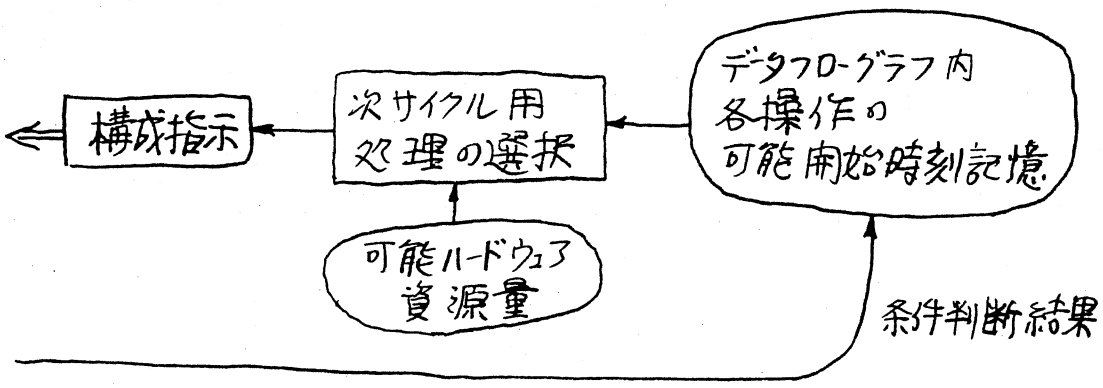

図 3: 構成指示の作成機構

3.3 メモリ構成

前述の装置は、メモリを介して過去の処理結果を現在の処理入力としている。その場合、メモリとしては、一つ一つの同定が出来さえすれば何でもよい。しかし、ある時点での処理結果を蓄えて後、それを使う時点では、すぐ入力に使える状態になっている必要がある。データを利用する時点でのメモリからのアクセス遅延は、処理性能を落すことになる。

従って、実際のメモリ構成としては、それを幾つかの種類に分け、各データの利用緊急度に応じて、データを配置する必要がある。前述の各処理開始可能時刻解析は、この緊急度判定に使われ、次のように処理結果の記憶配置が行なわれる。

1. 高速レジスタ: 次のサイクルで利用するデータ
2. 高速メモリ: レジスタが不足する場合のデータ、及び、近未来で使われる予定のデータ
3. 中速メモリ: このプログラムで利用するが、取り敢えずは使わないデータ
4. 低速メモリ: 将来の為に残して置くデータ

この場合、高速 / 中速メモリ内のデータは、使われる時刻が近付くと、前もって読み出しの指令が構成指示から送られ、高速レジスタや高速メモリに蓄えられる。これは、いわゆるキャッシュの明示化である。

また、同じ時刻に必要とするデータが複数あるが、それを収める高速メモリ容量が十分ない場合、その内どのデータを優先するかは、そのデータ遅延が及ぼす将来の処理遅延の影響が少ない方を選ぶ。

4 ハードウエア構成

上述の装置を構成する場合、幾つかの設計ポイントがある。

4.1 構成指示とのパイプライン構成

構成指示の処理時間は次のように分けられる。

- 構成指示指令の作成
- 構成指示に従って回路を設定すること
- 設定された回路にデータを流すこと

これらは、使用回路部分が異なるので、パイプライン構成とすることが考えられる。通常の命令パイプラインの拡張である。

4.2 処理装置の構成

加算器、乗算器、論理演算器、シフタ等、演算の基本的な回路を多く集め、それらの間の配線を自由に行なえるようにしたものがこれである。例えば、図4のような構成が考えられる。

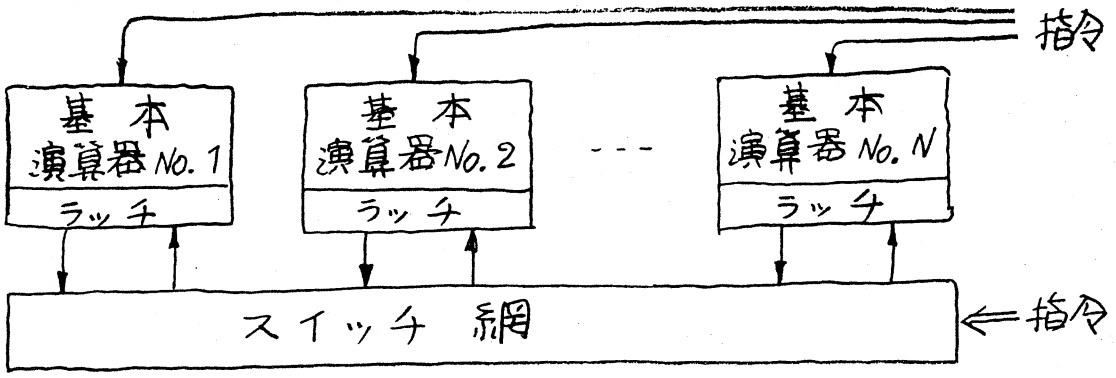

図 4: 処理装置の構成

これを適切に分割し LSI によって実装する。

4.3 構成指示装置の構成

データフローグラフを保持し、その各部の処理開始時刻を管理することが中心となる。従って、開始時刻をキューとして保持し、その各エントリと元のデータフローグラフとは、双方向ポインタで結んでおく。処理装置からの、条件判断結果が送られて来ると、それに従って、このキューを操作し、不要部分を削除するとともに、適切な順序に入れ替える。

そのキューから、利用可能なハードウエア資源の数 (N) で賄うことが可能な処理要素を貫出して、構成指示を作成する。同時に、将来必要なデータで、中速メモリに存在するものの内、今、起動をかけておくべきものを選び、先読み指令を作成する。

4.4 ハードウエア実装

前述の議論は、処理装置一般論であるが、このような装置を実装する行き方には幾つかの選択があり得る。

1. 単純実装: この極端は、処理装置としてALU一つのものを考える形態であり、通常のマイクロプロセッサ構成となる。

2. 小規模実装: ALUを複数、例えば、数個並べるもので、スーパースケーラや、VLIW等がこれに相当する。

3. 中規模実装: ALUを数十ヶのオーダで並べるもので、演算パイプラインを複数持つ、スーパーコンピュータがこれに相当する。

4. 大規模ALU実装: 将来の超高速コンピュータの実装形態の一つと考えられる。構成指示装置の数は小数で、ALUの数が数千から十万にのぼる構成を取る。

5. 多数プロセッサ実装: 構成指示と処理装置とを対にして、それを単位に数多く集め、メモリを共有することによって大規模装置とした構成。通常の高速マイクロプロセッサを多く集めた超並列コンピュータがこれに相当する。

4.5 議論

大規模装置の二つの構成法のどちらが良いであろうか。多数プロセッサ方式は、従来の単一プロセッサ方式の自然な拡張と捉えられ、より親しみ易い方式であろう。しかし、この行き方では、多くのプロセッサを協調させるための方法がadhocになり勝ちで、多くの発見的手法(その一つがキャッシュ)で対処することが多い。

一方、大規模ALU方式は、その構成指示や、大規模実装の問題が余りクリアーではないものの、処理を全体として効率良く行なう為の指針が明解である。

大規模ALU方式は、プログラムの解析手法の発達が不可欠であるが、今後のコンパイラ等解析手法の発達を考えるとき、大規模ALU方式がより自然で、性能を出し易い構成ではなかろうか。解析プログラムに取って、構成指示を複数に分割することは、問題を組み合わせ的に複雑にする要因となり得る。並列処理の研究に於いて、変数の配置問題や、スケジュール問題、負荷分割問題等があり、それらはいずれも、決定的な手法に欠けるという状況にあるが、この主要な原因は問題の複雑さにある。この解決無しには、大きな進歩は難しいのではなかろうか。

5 おわりに

この小文では、情報処理機構を始めから考え直してみるという試みを紹介した。21世紀、それは、情報処理が隅なく行き渡り、広く使われる時代である。ここいらで、情報処理の基本を振り返り、将来のベースとなるべき機構を反省してみるのも意義あることではなかろうか。従来からの研究成果は多く、それらはそれぞれに大変優れた方式提案になっている。この文での検討は、それらの成果を再度、位置付ける試みと見ることも出来よう。