修 士 論 文

並列論理型言語の実行に適した マルチスレッド型プロセッサ アーキテクチャに関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

細井 雅之

2002 年 3 月

修 士 論 文

並列論理型言語の実行に適した マルチスレッド型プロセッサ アーキテクチャに関する研究

指導教官　日比野靖 教授

審査委員主査　日比野靖 教授
審査委員　田中清史 助教授
審査委員　堀口進 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

910102 細井 雅之

提出年月: 2002 年 2 月

要旨

本論文では並列論理型言語のためのアーキテクチャによるサポート及び並列性を内在するパラダイムに適した並列実行機構を提案する。関数型、論理型、オブジェクト指向型のいずれのパラダイムも逐次的なアーキテクチャで効率良くシミュレートされるが、それらのパラダイムに内在する並列実行可能性は無視されている。更に、各々のパラダイムに関する並列実行には多くの研究があるにもかかわらずそれらのパラダイムに共通に適用できる並列実行機構についてはほとんど関心が払われていない。日比野研究室では、関数型プログラムの実行に適したマルチスレッド型プロセッサ・アーキテクチャが提案されている。そこで本研究では、共通に適用可能な並列実行機構を提案するために、まず第一に、GHC と呼ばれている並列論理型言語に着目する。

目 次

第1章　緒言

1.1　研究の背景

高級言語計算機は現在では顧みられなくなり、RISC を中心としたマイクロプロセッサ時代を迎えている。高級言語計算機はソフトウェアの課題をハードウェアで解決するという試みであった。高級言語計算機の存在意義としては以下のことが考えられた。

1. 通常マシンのコンパイラが複雑になりすぎた。
2. 高級言語計算機を用いることでプログラミングが容易になり、ソフトウェアの開発費用を削減する。
3. これまでソフトウェアで処理してきたことを安価になるハードウェアで処理することで上昇するソフトウェア費用を削減する。
4. 計算機システムは高級言語を効率良く実行できるように設計されるべきである。
5. コード能率が良い。
6. 高級言語計算機こそが高級言語を効率良く実行できる費用効率の良いアーキテクチャである。

しかし、これらは全て誤りであった。上記に関しては以下のように考えるのが妥当である。

1. コンパイラが複雑になるのは、コード生成が困難だからではなく、前処理、語い解析、構文処理、最適化、誤り検出、誤り回復等の部分が相当の規模になっているからである。
2. マシンの水準によらず高級言語プログラムの効率の良いデバッグ環境を開発すべきである。
3. ソフトウェアアルゴリズムをハードウェア化するための設計費用、設計時間は膨大であり、ソフトウェアの開発費用を償えない。また、ソフトウェアはその複製費用がただ同然であるのに対し、ハードウェアの複製費用（LSI チップの製造費用）は集積技術がいかに進んだとしてもその低下には限りがある。
4. 第一に、高級言語指向ということは、対象の言語を一つに限るという危険を侵すことであり、第二は言語実行の効率化を目標とするあまり、システムの総合的効率を無視しがちであるということである。
5. コード能力が良いことと、性能が良いこととは一致しない。しかもコード能率の良さは特定の言語を対象にすることで達成される。従って、多種多様の言語に対し、高いコード効率が達成されるかは疑問である。
6. 仮に直接実行マシンが実現できたとしても、デバッグ済みのプログラムを、実行の度に構文解析/ 誤り診断を受けるのは全くの無駄であり、構文解析/誤り診断は、コンパイル時に行ない、実行時は通常レベルのマシンコードを実行した方がはるかに費用効率が良い。

つまり、計算機システムのユーザ評価は、高級言語計算機の達成度ではないのである。総合的な費用効率の良さにある。以上のことから、高級言語計算機の存在意義は否定され、高級言語計算機という捉え方での研究は行なわれなくなり、同時に高級プログラム言語の研究も低調となった。しかしながら、ソフトウェアの課題すなわちソフトウェアの開

発費用がますます増大していくという問題は未解決のままであった。ソフトウェアの課題としては主に以下の 2 つが考えられる。

1. ソフトウェア開発の方法を工学的に捉えようとするソフトウェア工学という課題
2. プログラミングのパラダイムを根本から見直し、これまでのノイマン型アーキテクチャに基づく命令型 (imperative) のパラダイムから、関数型 (functional)、論理型 (logic)、オブジェクト指向型 (object oriented) といったそれまで一般的でなかったプログラミングパラダイムに転換を図るという課題

この中でも関数型プログラミング、あるいは論理型プログラミングは、並列処理あるいは並列計算機アーキテクチャの観点から関心がもたれた。しかしながら、これらも総合的な費用効率の良さとは一致しなかった。費用効率という観点からは、より簡単な構成、より少ない資源で実現されるマシンの方が、性能で若干劣っていたとしても優位に立つ。従って、高級言語計算機は成立し得ない。

一般的にプログラム言語の処理は、以下の 3 つに分けられる。

1. 言語の表層的な構文の処理
2. 演算実行の順序制御
3. 演算実行

この内、第一の構文の処理はコンパイラに任せるのが費用効率の点から優れていることは述べた。一方、ハードウェアによる高速化に最も効果があるのが演算実行部である。ハードウェアによる高速化は並列動作による効果である。高級言語処理に関して残された部分は、演算実行の制御である。これに関しては現在は命令パイプラインを有する逐次型アーキテクチャマシンを対象にした命令生成と命令スケジューリングにのみ関心が払われている。関数型、論理型、オブジェクト指向型のいずれのパラダイムも逐次型アーキテクチャでシミュレートできるが、これらのパラダイムに内在する並列実行可能性は無視されている。また、各々のパラダイムの並列実行については多くの研究があるが、それらに共通に適用できる並列実行機構についてはほとんど関心が払われていない。以上の議論は文献 [1] をもとにしている。

1.2 研究の目的

そこで本研究では各々のパラダイムに共通に適用できる並列実行機構を提案するために、その前段階として、1 つのパラダイムに着目し、そのパラダイムに内在する並列実行可能性を探求する。着目するパラダイムは論理型とし、特に並列論理型言語に焦点を充てる。並列論理型言語は並列性の自然な記述を可能とする。この言語は従来の手続き型言語に比べて複数計算間の待ち合わせを直接意識することなく並行性の記述が可能になるので、問題に内在する並列性を最大限に引き出すことができる。一方、並列論理型言語はその言語特有の性質から実行効率は逐次型アーキテクチャマシンではあまり良くない。並列論理型言語の効率良い実行を妨げる要因としては主に 2 つある。1 つはユニフィケーションと呼ばれるパターンマッチングの繰返しである。ユニフィケーションを効率良く実

行させるにはデータ型を出来る限り早い段階で決定する必要がある。この対策としては1命令で多方向への分岐が可能なタグ付き分岐命令 JTD が提案されている [12]。もう1つは頻繁に生じる実行コンテキストの切替えである。並列論理型言語用マシンのこれまでの研究としては、並列マシンの性能の基本となる要素プロセッサ自体の高速化よりもむしろネットワークの構成やプロセッサ間通信の方式など、「並列」に重点をおいたものであった。一方で、並列論理型言語特有のコンテキスト・スイッチが予想以上に高頻度であるとともに、この処理の「重さ」が性能にかなりの悪影響を及ぼしていることが指摘されている [2]。これはデータ依存関係による小さな粒度のプロセス（スレッド）間の同期を取るために発生する。従って、「軽い」コンテキスト・スイッチを実現する機構を備えた要素プロセッサのアーキテクチャの検討が必要である考えられる。

1.3 本論文の構成

以下、本論文では次のような構成に従って、並列論理型言語の実行に適したマルチスレッド型プロセッサアーキテクチャについて論じる。

まず第二章では、プロセッサのアーキテクチャを論じるための基礎として、論理型言語がどのようなものであり、またどのように処理されるかを述べる。

次に第三章では、シミュレーションによる実験を行なう前段階として、プログラム中のどのスレッドがある時点に実行されるかを決定する方法を確定する。そのために、この章では、マルチスレッドプログラミングが可能な典型的な言語として Java に着目し、Java 仮想マシンが、ある時点で実行されるべきスレッドをどのように決定するかという点を確認する。

第四章では、本論文の主題である提案したアーキテクチャに関して述べる。

また第五章では、提案したアーキテクチャに関する有効性を動作シミュレーションする。

第六章では、実際の評価データに基づいてその有効性を明らかにする。

最後に第七章では、本研究の結言を述べるとともに、今後の課題について概観する。

第2章　論理型言語とその処理方式

2.1　論理型言語の概要

本研究では特定のプログラミング言語（または言語群）に向いたプロセッサのアーキテクチャを論ずる。そのためには、対象となる言語の持つ性質や特徴を知らなければならない。特に専用プロセッサの最大の特徴である高速処理の実現のためには、プログラムがどのように処理され、またどのような操作が頻繁に実行されるのかを、具体的に明らかにする必要がある。

そこで本章では、論理型言語の仕様や特徴、更には具体的な処理の方法について代表的な論理型言語である Prolog を対象に論ずる。Prolog は 1971 年に Alan Colmerauer により考案され、その後 Robert Kowalski によってプログラミング言語として位置付けられた論理型言語である [3]。Prolog の最大の特徴は、一階述語論理に基づくホーン節によってプログラムが表現されること、すなわち「論理」に基礎を置いていることである。従って、プログラムの数学的な取扱い、例えばプログラムの検証、合成、変換などの問題に対する相性が良い。

一方別の観点からは、Prolog を強力な記号処理言語と捉えることもできる。Prolog はこの分野での先駆的な存在である LISP と同様に、リスト構造やより複雑な構造体を取り扱う機能を持つ。その上、ユニフィケーションという強力な操作によって、不完全なデータ構造の利用や、複雑な構造体間のパターン・マッチングが可能である。

以下本章では、まず Prolog のプログラミング言語としての構造を概説した後、その特徴のユニフィケーションの処理について述べる。本研究は論理型言語ではなく並列論理型言語に着目するのでバックトラックの処理に関する説明は省略し、論理型言語の説明の後、続いて並列論理型言語の概要について述べる。

2.1.1　言語の構造

Prolog のプログラムは、事実 (fact)、規則 (rule)、及び質問 (query) の三つの要素から構成されていると捉えることができる [4]。

```
example1.

    father(忠盛, 清盛).
    father(忠盛, 経盛).
```

```
father(忠盛, 教盛).
```

は全て「事実」 [5] であり、「忠盛は清盛の父である」などを表現していると解釈することができる。また、

```
example2.

    brother(X,Y):-father(F,X),father(F,Y).
```

は規則であり、「F が X の父であり、かつ F が Y の父でもあるならば、X と Y は兄弟である」という論理的な推論規則を示す。ここで X,Y,F は変数であり、この「規則」に関してはその値はどのようなものであっても良い。

なお、「事実」と「規則」は総称してクローズと呼ばれる。「規則」の左辺 (':-' の左側) をヘッド、右辺をボディと言う。ボディを構成する要素 (上の例では father(F,X),father(F,Y)) はゴールと呼ばれ、これらは AND の関係にある。更に、名前や引数の数が同じヘッドを持つクローズの集合は述語と呼ばれ、一つの述語に含まれる事実や規則は OR の関係にある。

次に、これらの事実と規則を前提にしたとき、「清盛と経盛は兄弟であるか」という命題は、以下の三段論法によって真であることが証明できる。

1. 忠盛は清盛の父であり、かつ忠盛は経盛の父である。
2. F が X の父であり、かつ F が Y の父でもあるならば、X と Y は兄弟である。
3. ゆえに清盛と経盛は兄弟である。

一方、Prolog の処理系に対して；

```
example3.

   :-brother(清盛, 経盛).
```

という「質問」を行なうと、前述の三段論法を逆方向にたどる形の後向き推論によって、以下に示すような実行過程を経て質問が真であることが導き出せる。

```
example4.

****execution of brother(清盛, 経盛)****
(1) :-brother(清盛, 経盛).
(1)'  ****seach****
       ゴール brother(清盛, 経盛) とユニファイ可能なヘッドを持つクローズを探す

(2) brother( X , Y ):-father(F,X),father(F,Y).
(2)'    上記のクローズが見つかった。ゴールとヘッドをユニファイする
```

```
(2)'  ****unify(brother(清盛, 経盛),brother( X , Y ))****

(3) brother(清盛, 経盛):-father(F, 清盛),father(F, 経盛).
(3)'  ****seach****
(3)'   ゴール father(F, 清盛) とユニファイ可能なヘッドを持つクローズを探す
(3)'   クローズ father(忠盛, 清盛) が見つかった。ゴールとヘッドをユニファイする
(3)'  ****unify(father(F, 清盛),father(忠盛, 清盛))****

(4) brother(清盛, 経盛):-father(忠盛, 清盛),father(F, 経盛).
(4)'  ****seach****
(4)'   ゴール father(F, 経盛) とユニファイ可能なヘッドを持つクローズを探す
(4)'   クローズ father(忠盛, 経盛) が見つかった。ゴールとヘッドをユニファイする
(4)'  ****unify(father(F, 経盛),father(忠盛, 経盛))****

(5) brother(清盛, 経盛):-father(忠盛, 清盛),father(忠盛, 経盛).

****program**********************************
     father(忠盛, 清盛).
     father(忠盛, 経盛).
     father(忠盛, 教盛).
     brother(X,Y):-father(F,X),father(F,Y).
*********************************************
```

以上のような実行過程を、手続き型言語のそれと比較すると、

1. クローズ ≈ サブルーチン
2. ゴール ≈ サブルーチン呼出
3. ユニフィケーション ≈ 引数の受渡し

のように考えられる。なお、ユニフィケーションは、手続き型言語の引数受渡しとは異なり、双方向の代入と比較を兼ね備えている。よって Prolog は「ユニフィケーションと言う特殊な引数受渡し機構を持った、サブルーチン呼出しだけからなる言語」と言うことになる。

2.1.2 ユニフィケーション

基本的なデータ表現

Prolog の最も基本的なデータは、アトムと整数である。アトムは通常小文字で始まる英数字の列で表現される。例えば；

```
example5.

   father    mother   brother   sister   family
```

は全てアトムである。

さて、アトムとアトムのユニフィケーションの規則は

```
if 同じ名前のアトム
  then ユニフィケーション成功
else
       ユニフィケーション失敗
```

というものである。従って、ユニフィケーションを行なう側にとって、アトムの具体的な名前である文字列は意味がなく、個々のアトムを識別するための「番号」付けがなされていれば良い。そこで、多くの場合、アトムの内部表現は；

```
example6.

   father    -> 1
   mother    -> 2
   brother   -> 3
   sister    -> 4
   family    -> 5
```

のように、適当な整数が用いられる。

一方、整数の自然な内部表現は整数自身であるが、例えば「整数 '3'」と「番号 '3' が割り振られたアトム」とは区別する必要がある。この区別のために用いられるのがタグである。一般に Prolog のデータの内部表現は図 2.1 に示すように、データの「型」を示すタグと, 数値やアトムの番号などを示す「値」の部分に分かれている。

変数の表現

Prolog の変数の有効範囲はクローズの内部であり、手続き型言語の局所変数と同様にローカルなスコープを持っている。また、クローズが「呼び出された」時点では、変数は「何も入っていない」状態となっている。この「何も入っていない」状態、すなわち未定

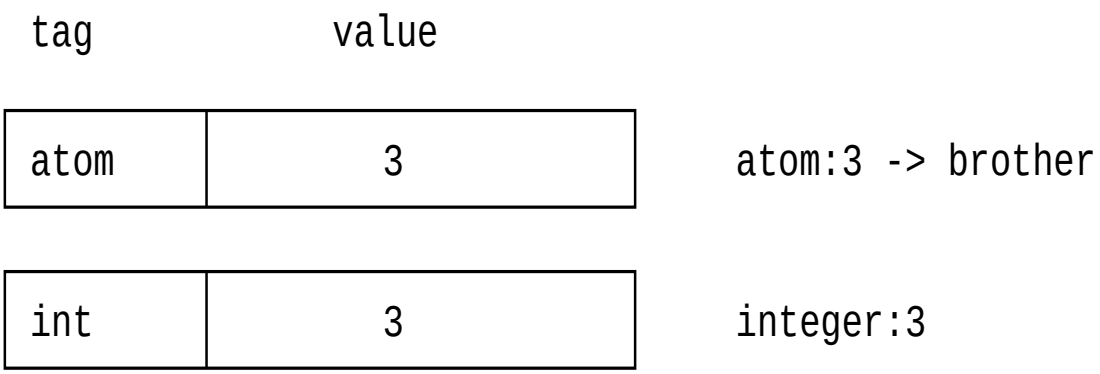

図 2.1: アトムと整数の内部表現

義状態が Prolog の変数の大きな特徴である。Prolog の変数は未定義状態の場合のみ、任意の値をユニフィケーションによって代入することができ、一旦代入が行なわれると別の値に変更することはできない。

```
example7.
(1) brother( X , Y ):-father(F,X),father(F,Y).
(2) brother(清盛, 経盛):-father(F, 清盛),father(F, 経盛).
(3) brother(清盛, 経盛):-father(忠盛, 清盛),father(忠盛, 経盛).
```

以上の議論から、Prolog の変数を表現するためには未定義状態であることを示す方法が必要である。そこで図 2.2 に示すように、タグが undef であるようなデータが未定義状態の変数であるとする場合が多い。そして、ユニフィケーションのとき、一方のタグが undef で、もう一方のタグが atom であれば、undef である未定義変数にタグと値の両方が代入される。また、未定義変数の「値」の部分は何であっても良いが、他の処理との関係で自分自身を指すポインタを入れておくと便利であることが多い。さて一方が未定義変数であ

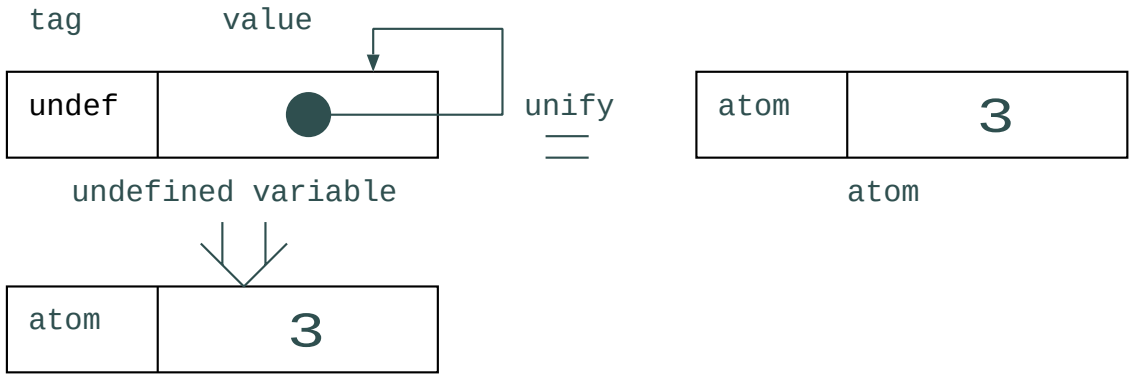

図 2.2: 未定義変数の内部表現

る場合のユニフィケーションでは代入が行なわれる。しかし、両方が未定義変数の場合には単純な代入を行なうことはできない。この場合、二つの変数を「同じもの」にする処理が必要となる。この「同じものにする」処理は、Reference Pointer という特殊なポインタを用いて行なう。図 2.3 に示すように、未定義変数どうしのユニフィケーションでは、一方の変数に他方を指すポインタを代入し、タグに Reference Pointer であることを示す ref をセットする。一方、Reference Pointer に対するユニフィケーションは、そのポインタが指すデータが「同じもの」であることを実現する。従ってユニフィケーションのとき

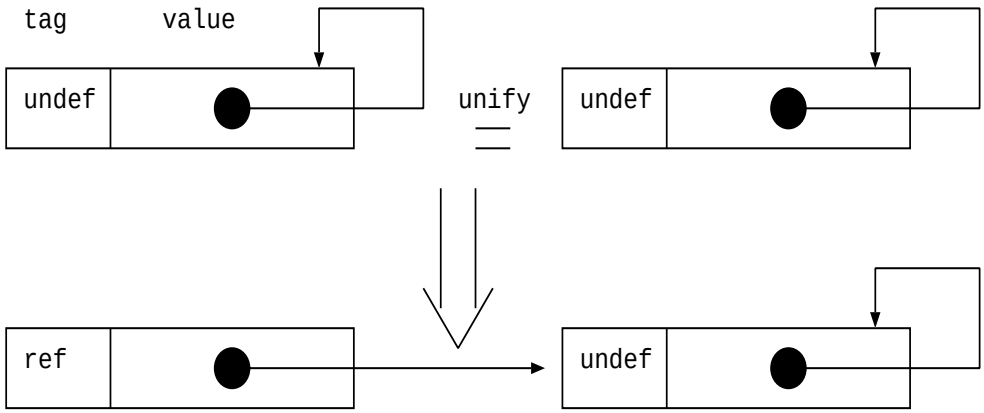

図 2.3: 未定義変数どうしのユニフィケーション

には、Reference Pointer であるか否かを判断し、そうであればそれが指しているデータを読む、という操作が必要となる。この操作はデレファレンスと呼ばれ、Prolog 処理において重要な操作の一つである。

基本的なユニフィケーション

以上のことから、アトム、整数、及び未定義変数（局所変数）に関する、二つのデータ X と Y のユニフィケーション操作は以下のようになる。

1. X をデレファレンスしその結果を X' とする。
2. Y をデレファレンスしその結果を Y' とする。
3. 表 1 に従った操作を行なう。

表1：基本的なユニフィケーション操作

tag(X')	tag(Y')		
	undef	atom	int
undef	Y' ← ref to X' if X' < Y' X' ← ref to Y' if X' > Y'	X' ← Y'	X' ← Y'
atom	Y' ← X'	success if X' = Y' fail otherwise	fail
int	Y' ← X'	fail	success if X' = Y' fail otherwise

ただし、X ’< Y ’はX ’がY ’よりもスタックのボトム側にあることを意味する。

2.2 並列論理型言語の概要

プログラミング言語である並列論理型言語 GHC(Guarded Horn Clauses) は、Prolog と同じ一階述語論理に基づいた論理型プログラミング言語で、第五世代コンピュータプロジェクトの一環として上田ら [21] の提案により、新世代コンピュータ技術開発機構 (ICOT) で開発された。GHC が Prolog と異なる最大の特徴としては、並列プログラミングを基本としている点である [6] 。本研究では対象とする並列論理型言語は GHC にしているが本章では実際に処理系 [7] が配布されているプログラム言語 KL1 を基に概説する。KL1 は並列論理型言語 GHC に基づいて並列動作することを前提に設計されたプログラミング言語である。プログラムはガード付きホーン節の集合として構成されている。

図 2.4 に示すように H はヘッド、G_n はガード、B_m はボディを表す。ヘッド H、ボディ B

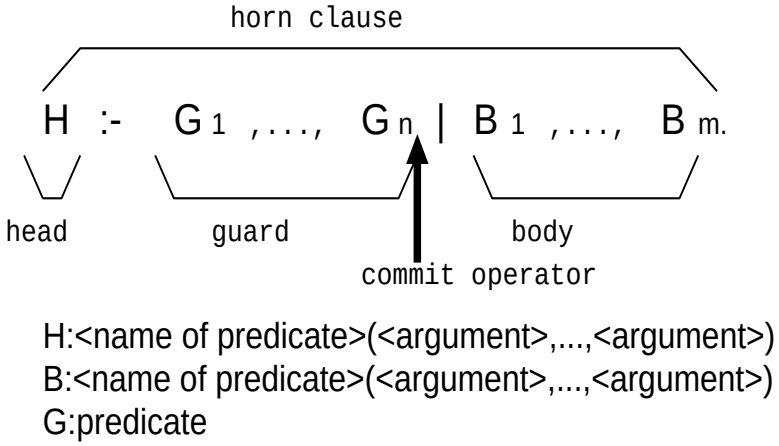

図 2.4: 代表的なホーン節 (GHC)

は述語名と引数で表されている。述語名、引数、ガード、ボディ部については下記に示す。

述語名 節で定義を与える述語の名前

引数 述語の引数と節の中で使う変数名の対応をつけるための仮引数の並び

ガード 節の適用条件を示す部分でカンマで区切ったガードゴールを全て満たしたとき適用条件を満たしたものとする

ボディ部 節が選ばれ適用条件を満たしたときには、ボディ部全てを実行する。内容は active unification、他の述語、自分自身を呼び出すゴールが書ける

2.2.1 基本的な実行機構

ユーザが定義した述語ゴールの実行において、ある初期ゴール（集合）が与えられると次のように実行される。（ゴール：計算を起動させる文）

1. 初期ゴール集合を実行すべきゴールの集合（ゴールプール）に入れる。
2. ゴールプールの中から任意のゴールＰを取り出し、ゴールＰの述語名と引数個数が一致するヘッドｈを持つ節をプログラムから探す。ここでそのような節が節Ｃであったとする。
3. 次に、節Ｃのガード検査が行なわれ、適用条件が満たされるとゴールＰを節Ｃのボディ B に書き換え（リダクション（簡約化））、ゴールプールにボディ B にあるボディゴールを戻す。
4. 節Ｃのガード検査が行なわれ、適用条件が満たされない（条件を試せない）ときはゴールプールに差し戻し、ゴールプールにある違うゴールＰ’を取り出す（ゴールＰの中断）。
5. 2〜3 の動作をゴールプールに書き換えるゴールが無くなるまで繰り返す。

このように計算を行なう。なお、基本的な実行機構のイメージを図 2.5 に示す。

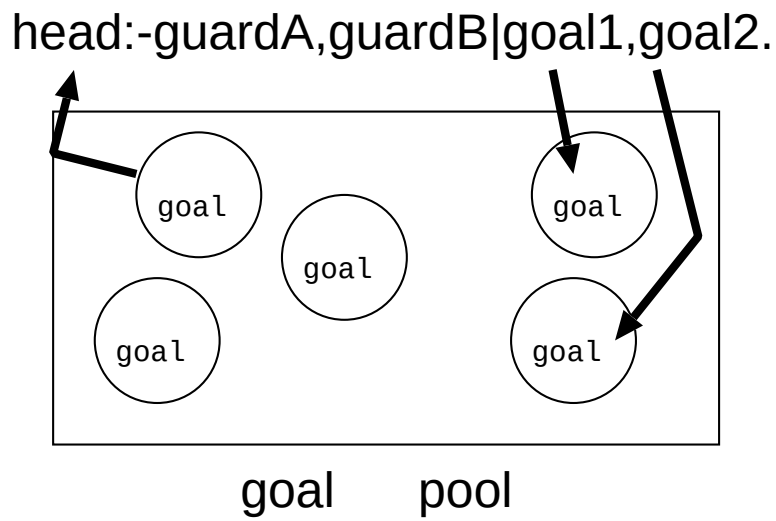

図 2.5: 基本的な実行機構のイメージ

2.2.2 通信、同期の機構

例えば次のようなプログラムを実行しようとする。not 述語は inverter を意味する。

```
example8.
```

```
**********not.kl1**********
            main :- true |                          %1
            not(X,Y),                          %2
            not(1,X),                          %3
            io:outstream([print(Y),nl]).   %4
            not(In, Out) :- In = 0 | Out = 1.  %5
            not(In, Out) :- In = 1 | Out = 0.  %6
************************************************
```

- %1、%5、%6 は OR 関係
- %2、%3、%4 は AND 関係 · · · 並列実行可能

以下で上記のプログラムの実行過程を追ってみる。

1. まず、main 関数が goal pool に入れられる。
2. 次にゴールプールには main ゴールしかないので、main ゴールが選ばれる。
3. プログラム中の main 述語ではガード部が true だから無条件に
   not(1,X),not(X,Y),io:outstream([print(Y),nl])
   に書換えられてゴールプールに入れられる。この 3 つの書換えられた各々のゴールは並列実行可能である。
4. not(X,Y) が取り出されたとする。
5. プログラムでのヘッドは%5,%6 の not(In,Out) で合致する。しかし、ガード部分で %5,%6 とも条件を試せない。従ってゴール not(X,Y) は中断される。その結果、ゴール not(X,Y) はゴールプールに戻され、実行コンテキストが切替えられる。
6. 次にゴール not(1,X) が選ばれるとプログラム not(In,Out) の%6 で条件が合う。従って X=0 とアクティブユニフィケーション（能動的に変数 X が 1 に単一化）される。追加されるゴールは存在しない。
7. ゴール io:outstream([print(Y),nl]) %4 が選ばれても Y が具体化されていない。従って中断されて、実行コンテキストが切替えられる。
8. ゴール not(X,Y) が選ばれると 6. で X=0 とユニフィケーションされていた。従ってプログラム not(In,Out) %5 で条件を満たす。その結果 Y=1 とユニフィケーションする。追加されるゴールは存在しない。
9. 残っているゴールは io:outstream([print(Y),nl]) だけである。これは組み込み述語である。Y が 1 と具体化されている。従って 1 を出力し、ゴールプールには何も存在しなくなる。
10. ゴールプールには何も存在しなくなったので計算を終了する。

以上のことから

- ゴール間での通信では共有変数への参照で行なう。

■ 同期はガードで必要な値を検査すると自動的に行なわれる。

が分かる。

またKL1の変数はいったん値が決まると別の値に変わることはなく同じ値を保持し続ける。一方Prologは深さ優先でかつ左方優先で探索が進むので無限ループに陥る可能性がある。KL1では非決定的に一つを選択した後、非可逆的に解を進めていく。そのため、KL1では無限ループに陥ることはない。

2.2.3 扱えるデータ

■ 内部構造を持たず、その値自身に意味がある

記号アトム 純粋な識別子

数値アトム 整数

■ 構造を持ったデータ

ファンクタ ファンクタ型

コンス リストへ発展できる

- 要素に構造を持ったデータでも良く、要素の値が未定のデータ構造でも良い

KL1の代表的な処理系KLICでは、以上の基本的なデータ型は型宣言しないで使える。それ以外の構造データはgeneric data objectで実現される [8]。

2.2.4 特徴

手続き型言語とKL1とを比べるとKL1では変数がその値を変えることは原則的にない。変数の値は具体化されていない、すなわち未定義状態であるか、ある値であるかのどちらかである。そのため変数の値が計算の進むたびに変わる手続き型言語と比べるとKL1は複数の選択肢の中から1つの解を導くことで十分な場合に対して有効であり、そのことも並列処理に適していることの一つとして考えられる。

KL1は書換えられるゴールが複数個あるとき、どのゴールを書換えるかは決定していない。そのためプロセッサの台数や実行可能なスレッド数や通信のタイミングによって得られる解が異なる。計算を自由な順番で割り当てられるので効率良く計算できる。しかし、実行ごとに計算が異なるため、デバッグでは困難になる場合がある。

論理型言語Prologと比べるとKL1は共有変数の値の具体的状況により同期機構を実現しているため並列処理に向いているといえる。

以上は文献 [9] [10] をもとにしている。

第3章　アーキテクチャの設計

3.1　目的

以上、論理型言語の処理、特に並列論理型言語の処理を確認した。本章では、言語特有の処理における問題点に着目し、アーキテクチャの立場からその改善案を提案する。

3.2　設計方針

ハードウェアの役割は、” 処理全体において比較的単純であるにも関わらずその頻度が多いために性能を抑えている要因” を解決することであり、これを、提案するアーキテクチャの設計方針とする。

3.3　言語の性質

第二章でも説明したように、

論理型言語 Prolog の処理は、

1. Unification(以下、ユニフィケーションと呼ぶ) と呼ばれるパターンマッチングの繰返し
2. それに失敗したときに起こる Backtracking(以下、バックトラッキングと呼ぶ) と呼ばれる状態の復旧

に集約される [12] [13] [14] [15] [16] [17] 。

並列論理型言語では、

1. AND 関係にあるゴールは並列に実行される
2. ゴール間の同期は共有変数のユニフィケーション、具体的には未定義変数の代入により行なわれる
3. 全てのクローズは「コミット」と呼ばれるカットに類似したオペレータを持ち、一つのクローズだけが決定的に選択される、すなわち、OR ノードの下の枝を全解探索するバックトラッキングを行なう代わりに入力引数のテストによって一つの枝を非可逆

的に選択する

という特徴を有する [2] [18] [19] [20] [21]。

従って、並列論理型言語では

1. バックトラッキングは発生しないためその点は考慮する必要はない
2. データ依存関係によって小さい粒度のプロセス (またはスレッド) 間の同期を取るため実行コンテキストが頻繁に切り替わる。

そのため並列論理型言語では

1. ユニフィケーション
2. 頻繁に生じる実行コンテキストの切替え

を検討する必要がある。

ユニフィケーションを効率良く行なうためには、データ型を出来る限り早い段階で決定する必要がある。しかしながら、論理型言語や並列論理型言語などのような記号処理言語の場合は、データ型が決定されるのは実行時であるので、正当性のチェックや操作手順の選択といったデータ型の判断は実行時に行なわなければならない [2]。これを汎用計算機上で行なう場合、AND 演算、比較演算、条件分岐など、いくつかの命令を実行する必要がある。これらを下記で示す。

1. 一般にプログラミング言語ではデータのアクセスや操作の具体的内容は対象となるデータの型によって異なる。
2. 実行時にデータ型を判別するにはデータの内部表現は以下のようにする手法が一般的である。
   (a) データの「値」 ...数値やアドレス。
   (b) タグ ...データの型を示す。
3. データ型の判断：タグの抽出とその値に基づく処理手順の決定であり、以下の手順を伴う。
   (a) タグの抽出 ...AND 演算
   (b) 判別した値の判断 ...比較演算
   (c) 判別した値に基づく処理のへの分岐 ...条件分岐

しかもデータ型の判断は、データの参照、演算、更新といったごく基本的な操作に付随しているため、手続き型言語に比べて処理速度が格段に低下する。これを解決するために、本研究では 1 命令で多方向への分岐が可能なタグ付き分岐命令 (JTD:Jump Tag Dispatch、以下、JTD と呼ぶ) を導入する [12]。これはデータの内部表現において値の他にタグを付加し、JTD によって 2 つのオペランド・レジスタのタグ部 2 ビットを連結した 4 ビットによってこの命令に続いて格納されている表を引き、そこに置かれたディスパッチング・アドレスにジャンプする。これによって 16 方向の分岐が実現され、ユニフィケーション

ルーチン等のプログラミングにおいて有効である。また、オペランド・レジスタの片方にゼロを代入することにより 4 方向分岐を実現することができ、WAM の Indexing 命令等をプログラムするのに活用できる。この命令の概要を図 3.1 に示す。

Opcode	op1	op2	Meaning
JTD	Rs1	Rs2	PC ← M [PC + Dispatch(Rs1,Rs2)]

op1 : operand 1
op2 : operand 2

図 3.1: JTD 命令

なお、JTD は分岐命令であるため、通常のパイプラインプロセッサではパイプラインを乱す。

また、頻繁なコンテキストスイッチに関してはマルチスレッド型プロセッサとすることにより隠蔽されると考えられる。

3.4 マルチスレッド処理

例えば、代表的なパイプライン処理、パイプライン処理の問題点のうちの一つの分岐ハザード、及びマルチスレッド処理は以下の図 3.2-3.4 に示すようになる。

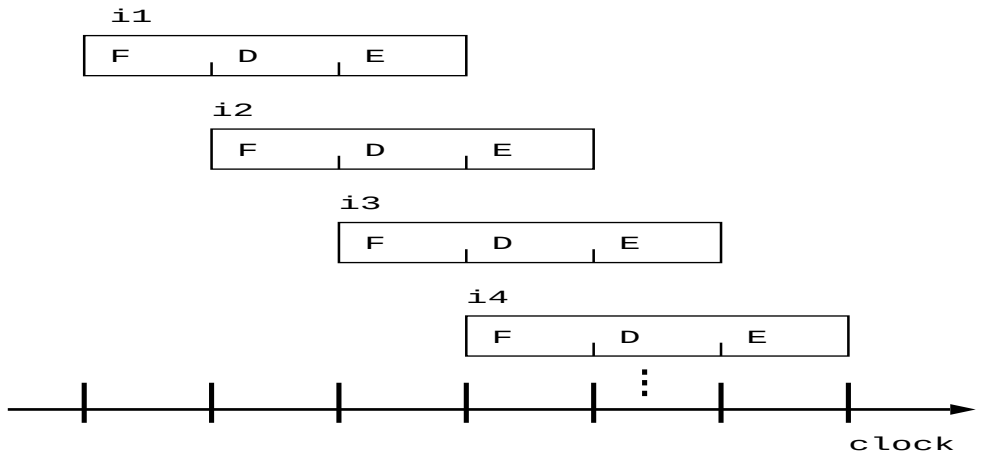

図 3.2: パイプライン処理

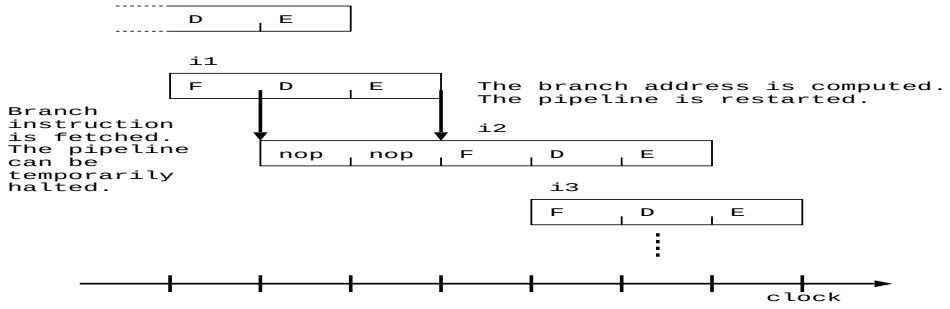

図 3.3: 分岐命令によるパイプラインバブルの発生 (分岐による遅延を低減させるには一般的に遅延分岐や分岐予測が用いられる。)

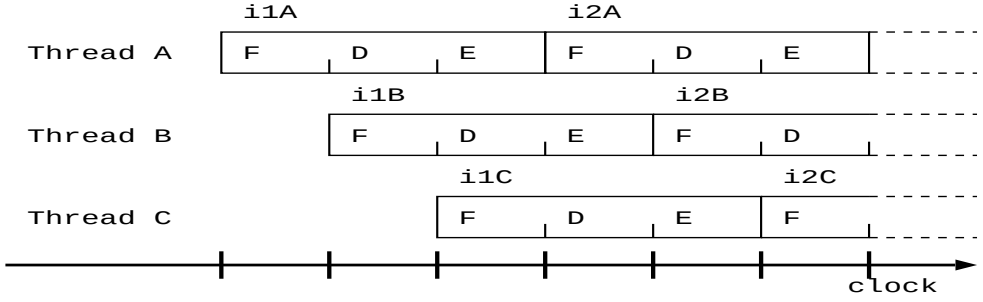

図 3.4: 複数スレッドのパイプライン処理による実行経過 (3 つのパイプラインステージに分割し、3 命令をオーバーラップして実行できるパイプラインプロセッサに、A、B、C の 3 つのスレッドの命令を順次投入し、実行。パイプラインに投入する命令を複数のスレッドの命令を順次投入してパイプラインバブルの発生を回避する手法)

3.5 マルチスレッド型アーキテクチャ

実際、伊藤らによって、関数型プログラムの実行に適したマルチスレッド型アーキテクチャが提案されている [22] 。このプロセッサは各スレッドごとにハードウェア資源を有し、次に実行されるスレッドを取り出すスレッド選択ユニットを持つ [22] [30] 。従って、データハザード、分岐ハザード、構造ハザードは避けられている。このプロセッサはキャッシュミスを扱う制御ユニットによりストールの生じないパイプラインとなっている。このパイプラインはスーパーパイプラインで構成されており、パイプラインステージと同数のスレッドを処理でき、このスレッドはプログラムカウンタ、レジスタファイル、及び各種レジスタを有する [25] [26] [27] [28] [29] 。なお、このプロセッサのハードウェア構成は以下の図 3.5 のようになっている。

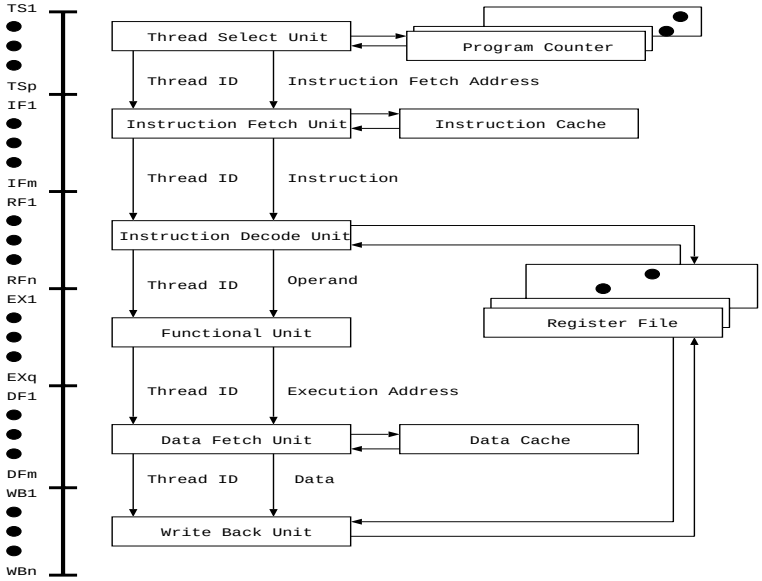

図 3.5: ハードウェア構成

このスレッドの切替え方針はサイクル毎となっている [31] 。この方針はクロック毎の切替えを認め、命令がロードされるかどうかとは無関係である。つまり、クロック毎という条件で異なるスレッドからのコンテキストを切替える。次々と続いていくスレッドは独立したものであり、パイプラインの実行に有効である。コンテキストの切替えはパイプラインの依存性を隠すことができる。複数スレッドのコンテキストをハードウェアで保持するのでコンテキストスイッチの時間コストはゼロになる。それゆえに、切替えによって性能の優位性が得られる [32] 。このプロセッサの主な特徴は 4 つある。第一に、このプロセッサはステージ数の数のスレッドで満たされるようにするための十分なスレッドがある限り、十分に利用される。そのため、スーパーパイプラインのステージ数を満たす十分なスレッドが必要なだけである。一度スーパーパイプラインのステージが満たされれば、このプロセッサはピーク時の性能で実行され続け、追加的にスレッドを加えても高速化さ

れない。第二に、このプロセッサは単一のプロセッサ環境で多くのスレッドをサポートする。第三に、このアーキテクチャモデルはパイプライン、及びクロック毎の多くの命令を発行する専用のユニットで構成される。第四に、このプロセッサはパイプラインステージと同数のスレッドを発行することによりパイプラインハザードを隠蔽できる。一方で、このプロセッサは 17 段の命令パイプラインを有しているが、関数型プログラムを実行させることによる実際の効果を証明するには至らなかった。なお、マルチスレッド処理を導入すればパイプライン・ハザードが発生しないためパイプラインの効率向上を図ることが可能となる [22] [23] 。

3.6 並列論理型言語とマルチスレッド処理

さらに並列論理型言語ではシステム内に存在する多数のゴールは比較的小さいプロセス (スレッド) に相当し、並列に処理可能であるため非常に多数のスレッドをパイプラインに投入出来る。従ってパイプラインに投入可能なスレッド数が少ないことによるマルチスレッド処理の効率低下が発生することもない。その上、上記で述べた JTD が原因によるパイプライン・ハザードもこのプロセッサではあればハザードが発生しないため問題はない。

以上の事から、JTD 及びマルチスレッド処理を組み合わせれば、実行サイクル数は大幅に短縮されると同時にハザード隠蔽効果による CPI 値の削減が可能となり、並列論理型言語の実行に適したアーキテクチャになると考えられる。

そこで、いままで見てきたように、データ依存による細粒度のプロセス（スレッド）間の同期による GHC の重いコンテキストの切替えを隠蔽し、さらに、並列実行可能性を最大限に生かすために、軽いコンテキストの切替えが可能なマルチスレッドプロセッサに着目し、並列論理型言語の実行に適したマルチスレッドパイプラインプロセッサを提案する。このアーキテクチャは並列論理型言語を効率良く実行させることが可能であり、並列性を内在する各々の言語に有益であり、並列実行可能性を最大限生かすことができることを目標とする。この提案を証明するために 4 つのことを行なう。最初、単純な構成のマルチスレッドパイプラインプロセッサを設計する。次に、実際に並列論理型言語の実行をシミュレートする。さらに、並列論理型言語の並列実行可能性を最大限生かすことによる性能の改善度を確認する。最後に提案したプロセッサが並列性を内在する各々の言語に適用可能な共通の並列実行機構を提供できることを確認する。我々はプロセッサを設計し、シミュレートするために、中村、小栗らによって独自に開発されたハードウェア設計システムである高位論理合成システム PARTHENON [24] を用いる。

CPU は、小さな手順の連続という形で個々の命令を実行する。それらの手順は次のようなものである。

1. メモリから命令レジスタに次の命令をフェッチする。
2. 次の命令を指すようにプログラムカウンタを書換える。

3. フェッチしたばかりの命令のタイプを判定する。
4. 命令がメモリ内のワードを使う場合、その位置を判定する。
5. 必要なら、メモリ内のワードデータを CPU レジスタにフェッチする。
6. 命令を実行する。
7. 手順 1 に戻り、次の命令の実行を開始する。

この手順の連続は、フェッチ-デコード- 実行:fetch-decode-execute サイクルと呼ばれることが多い。これは、すべてのコンピュータの処理の中核部分である [33] 。従って、最も基本的な命令サイクルは 3 つの段階:fetch,decode,execute からなる。これらの命令の段階は命令パイプラインによって実行される。従って、最も基本的なパイプラインのステージ数は 3 段からなるパイプラインプロセッサであり、また、マルチスレッド処理によるパイプライン・ハザード及び同期処理等の隠蔽効果を知るには最低 3 段からなるマルチスレッドパイプラインプロセッサが必要である。さらに、並列論理型言語のユニフィケーションを効率良く処理するためには上記で見てきたように JTD 命令が必要である。以上の事から、プロセッサとしては命令セットに JTD 命令を追加した 3 段からなるパイプラインプロセッサと命令セットに JTD 命令を追加した 3 段からなるマルチスレッドパイプラインプロセッサの 2 種類を作成して、シミュレーションを行なう。

命令セットに JTD 命令を追加した 3 段からなるパイプラインプロセッサは構造が最も基本的かつ単純であり、基本的なパイプラインプロセッサの特性をシミュレートして実測するために作成する。命令セットに JTD 命令を追加した 3 段からなるマルチスレッドパイプラインプロセッサはマルチスレッド処理による効果をシミュレートして実測するために作成する。命令セットに JTD 命令を追加した 4 段以上のパイプラインプロセッサ及び同様の構造のマルチスレッドパイプラインプロセッサは、命令セットに JTD 命令を追加した 3 段からなるパイプラインプロセッサ及び同様の構造のマルチスレッドパイプラインプロセッサの特性から理論的に拡張できるものとして作成しない。設計後、次の章では作成したプロセッサを用いて基礎データを収集する。JTD 命令の効果による特性を把握するために JTD 命令がある場合とない場合の CPI 値及びテストプログラムの実行に要するクロック・サイクル数の測定を行なう。実行可能なスレッド数を増加させていったときの性能の改善度に関する特性を把握するために、スレッド数を増加させていったときの CPI 値及びテストプログラムの実行に要するクロック・サイクル数の測定を行なう。ある種の並行プログラムに必要な同期処理の隠蔽効果に関する特性を把握するために、同期処理を伴うテストプログラムをシミュレートさせたときの CPI 値及びテストプログラムの実行に要するクロック・サイクル数の測定を行なう。並列論理型言語以外で並列性を内在するコードを実行させたときの特性を把握するために、そのコードをシミュレートさせたときの CPI 値及びテストプログラムの実行に要するクロック・サイクル数の測定を行なう。これらの特性により、第 5 章で、並列論理型言語の実行に適したプロセッサアーキテクチャ及び並列性を内在する全ての言語に共通して適用できる並列実行機構について論じる。

3.7 パイプライン方式

前の節で設計方針を説明した。この節では設計方針に従って命令パイプラインを有するプロセッサを設計する。仕様は以下の通りとする。

1. 命令語長は 34 ビット
   (a) タグ 2 ビット
   (b) データ 32 ビット
2. 命令メモリとデータメモリのバスは分割。命令フェッチと結果のメモリへの書き戻し同時可能
3. レジスタは
   (a) 34 ビット汎用レジスタ 8 本
   (b) プログラムカウンタ
   (c) フラグ

ハードウェア構成要素を表 2 に示し、このプロセッサで実行できる命令を下記表 3 に示す。表 2 の Rn は R0,···,R7、imm16 は 16 ビット即値を表す。また、パイプライン処理を行なう場合、パイプラインハザードが発生する可能性がある。そこで、通常のパイプラインプロセッサではハザードを回避する機構を実装する。このプロセッサでは、コントロールハザードに対してはフェッチ動作を止める代わりに、何も作業を行なわない NOP 命令を挿入した。データハザードに対してはシミュレーション時に NOP 命令を直接挿入することによりデータハザードを回避することにした。なお、このプロセッサの命令セットは jr Rn に関しては文献 [48]、jtd Rn,Rm に関しては文献 [12]、その他に関しては文献 [41] を参考にしている。PARTHENON で用いる言語 SFL でのパイプラインプロセッサの記述は付録 B.1 に示す。

表 2:ハードウェア構成要素

構成要素	処理内容など
I-Cache	命令キャッシュ
Fetch Unit	フェッチ機構
Decode Unit	デコード機構
機能ユニット	ALU 演算
D-Cache	データキャッシュ
Register File	レジスタファイル
PC, FLAG	PC、各種フラグ

表 3：パイプラインプロセッサの命令セット

命令の種類	ニーモニック	動作
特殊命令	nop	no operation
データ転送命令	mov Rn, Rm	Rn ← Rm
	mov Rn, TN	Rn ← ThreadNumber
	mov Rn, imm16	Rn ← imm16
	mov Rn, imm16	Rn < 15:0 > ← imm16
	mov Rn, imm16	Rn < 31:16 > ← imm16
	mov (Rn), Rm	M(Rn) ← Rm
	mov (imm16), Rm	M(imm16) ← Rm
	mov Rn, (Rm)	Rn ← M(Rm)
	mov Rn, (imm16)	Rn ← M(imm16)
論理演算命令	or Rn, Rm	Rn ← Rn | Rm
	and Rn, Rm	Rn ← Rn & Rm
算術演算命令	inc Rn	Rn ← Rn + 1
	dec Rn	Rn ← Rn − 1
	add Rn, Rm	Rn ← Rn + Rm
	sub Rn, Rm	Rn ← Rn − Rm
	cmp Rn, Rm	Rn − Rm
シフト命令	shr Rn	Rn ← Rn ÷ 2
	shl Rn	Rn ← Rn × 2
ビット命令とバイト命令	test Rn, Rm	Rn & Rm
制御転送命令	jmp imm16	pc ← imm16
	jmp Z,imm16	if Z,pc ← imm16
	jmp NZ,imm16	if not Z,pc ← imm16
	jmp C,imm16	if C,pc ← imm16
	jmp NC,imm16	if not C,pc ← imm16
	jtd Rn,Rm	pc ← M[pc + Dispatch(Rn,Rm)]
	jr Rn	pc ← Rn

また、ハードウェア構成を図 3.6 に示す。図中の点線で囲まれた部分が NOP 命令挿入処理機構となる。

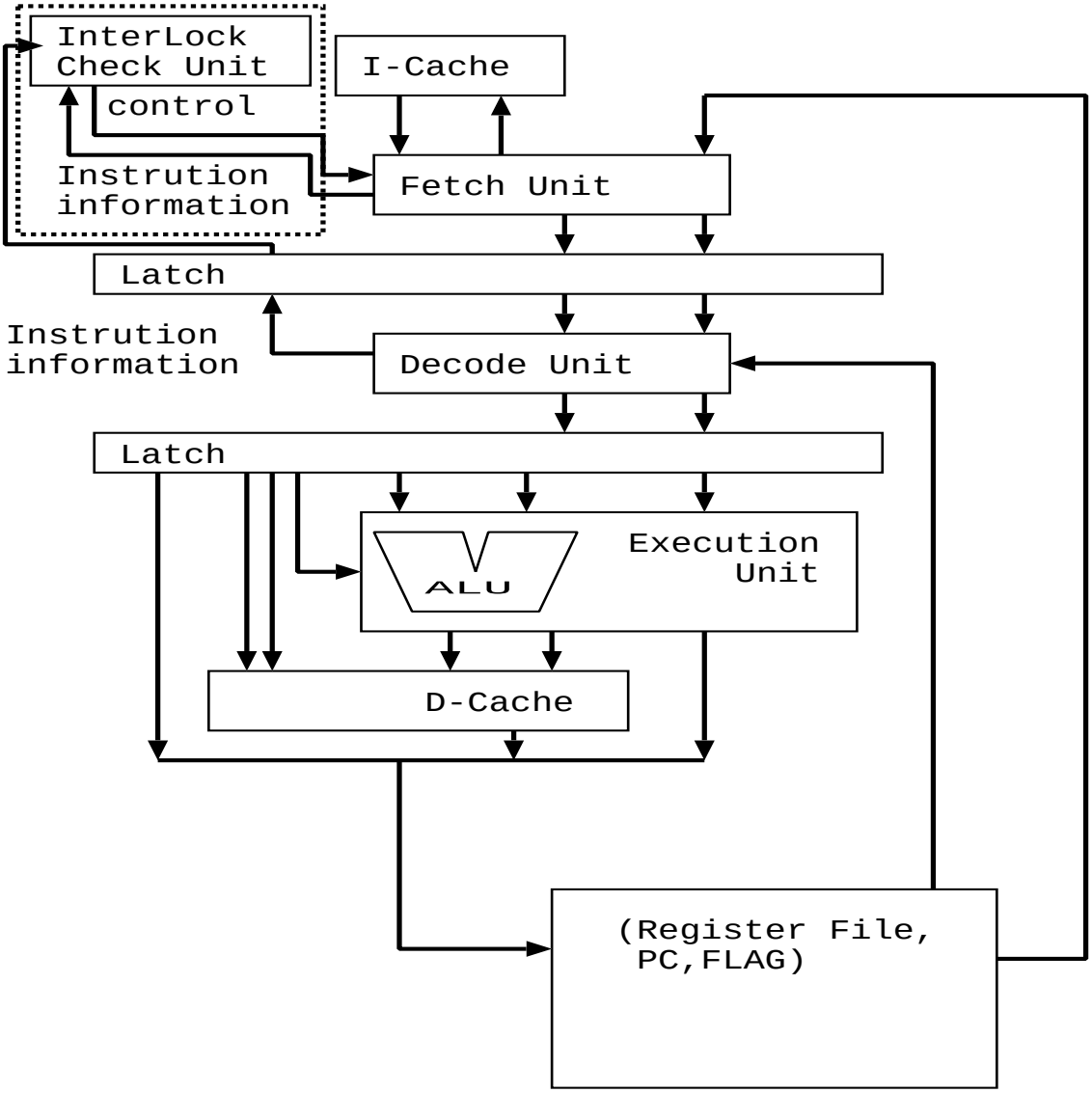

図 3.6: ハードウェア構成

3.8 マルチスレッド方式

前の節と同様に、この節では設計方針に従って命令パイプラインを有するマルチスレッドパイプラインプロセッサを設計する。仕様は以下の通りとする。

1. 命令語長は 34 ビット
   (a) タグ 2 ビット
   (b) データ 32 ビット
2. 命令メモリとデータメモリのバスは分割。命令フェッチと結果のメモリへの書き戻し同時可能
3. レジスタは
   (a) 34 ビット汎用レジスタ 8 本
   (b) プログラムカウンタ
   (c) フラグ

ハードウェア構成要素を表 4 に示す。また、このプロセッサで実行できる命令は表 3 及び下記表 5 とする。表 5 の Rn は R0,⋯,R7、imm16 は 16 ビット即値を表す。また、命令実行の処理の全体を 3 つのパイプラインステージに分割したので、3 つの命令をオーバーラップして実行することができる。このオーバーラップして実行される 3 つの命令を、それぞれ別のスレッドから読み込み、順次パイプラインに投入する。依存関係のない、3 つのスレッドをパイプライン処理するために、プログラムカウンタ (PC)、レジスタファイル、フラグを 3 セット用意する。また、3 つのスレッドをプログラムからコントロールできるようにするために表 5 に示す命令を実装する。opt という命令は、open thread の略で、新しいスレッドを開始させるための命令である。新しいスレッドの処理が開始されると、スレッドが使用中である事を示すフラグをセットさせ、プログラムから他のスレッドの使用状況を調べることができるようにする。これと対をなす命令として、clt という命令を用意した。これは close thread の略であり、スレッドで目的の処理を終了させるための命令である。この命令は、スレッドの使用中を示すフラグをクリアさせることにより命令フェッチの停止及びスレッドの使用終了通知を可能とさせている。また、このフラグの状態による分岐命令を追加した。以上の事から、3 つの命令を使用することにより、3 つ以上のスレッドをプログラムにより管理させ、処理の終了したスレッドに次々と新しいスレッドを投入させる。なお、このプロセッサのマルチスレッド用の命令セットは文献 [23] を参考にしている。マルチスレッドパイプラインプロセッサの SFL による記述は付録 B.2 に示す。

表 4:ハードウェア構成要素

構成要素	処理内容など
Thread Selecter	スレッドを選択
I-Cache	命令キャッシュ
Fetch Unit	フェッチ機構
Decode Unit	デコード機構
機能ユニット	ALU 演算
D-Cache	データキャッシュ
Register File	スレッド数必要
PC, FLAG	スレッド数必要

表 5：マルチスレッド処理に必要な命令セット

命令の種類	ニーモニック	動作
制御転送命令	jmp T1,imm16	if T1=1,pc ← imm16
	jmp T2,imm16	if T1=2,pc ← imm16
	jmp T3,imm16	if T1=3,pc ← imm16
スレッド開始命令	opt1 imm16	Open Thread 1,T1=1
	opt2 imm16	Open Thread 2,T2=2
	opt3 imm16	Open Thread 3,T3=3
スレッド終了命令	clt1	Close Thread 1,T1=0
	clt2	Close Thread 2,T2=0
	clt3	Close Thread 3,T3=0

また、ハードウェア構成を図 3.7 に示す。図中の点線で囲まれた部分がマルチスレッド処理用の機構である。

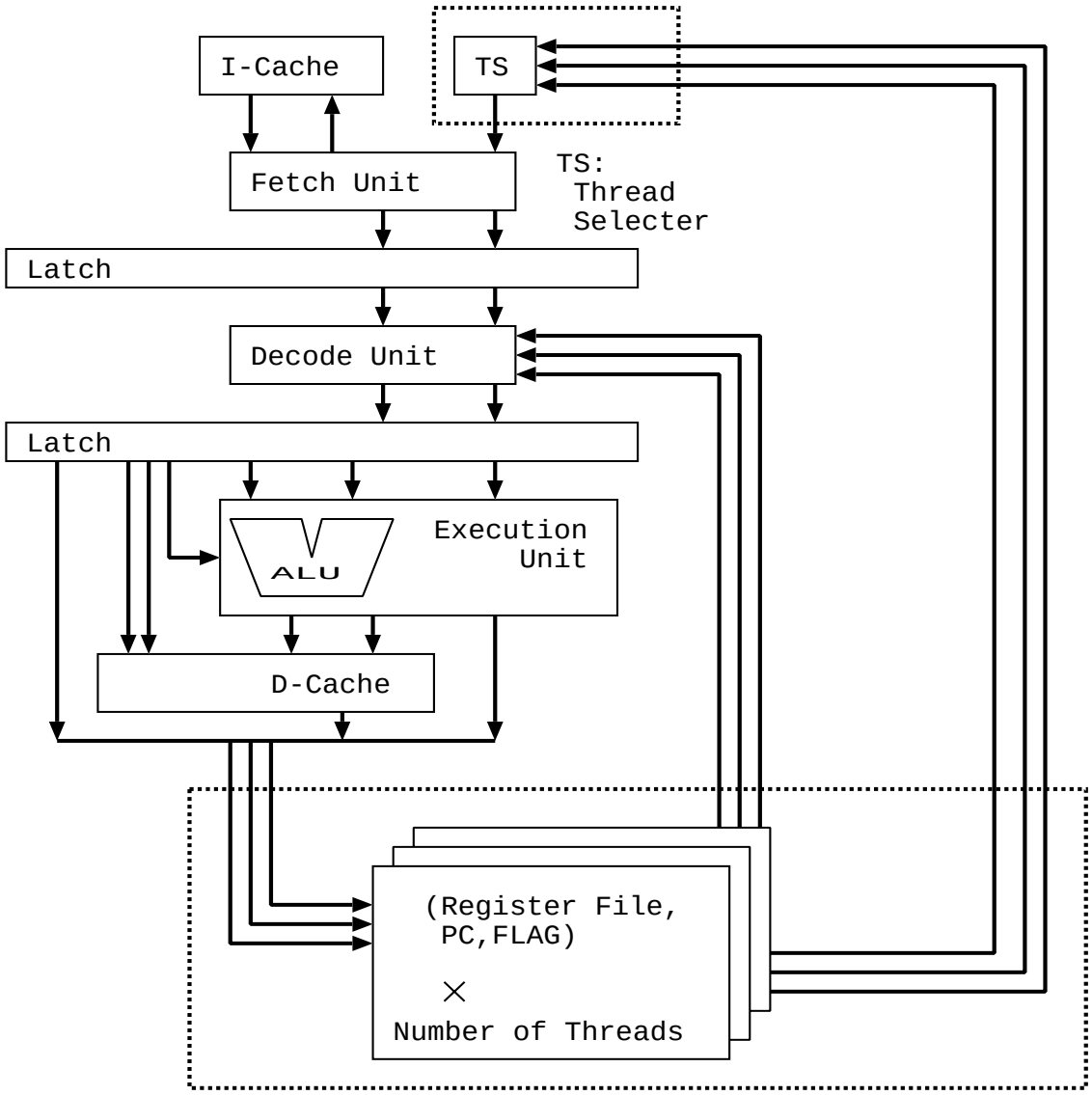

図 3.7: ハードウェア構成

3.9 スレッドスケジューリング

3.9.1 目的

本章で提案するプロセッサアーキテクチャの有効性を証明するために次章でシミュレーションによる実験を行なう。シミュレーションプログラムは GHC をハンドコンパイルしたコードとする。従って、まず第一に、マルチスレッドプログラム用のハンドコンパイルしたコードに対して、プログラム中のどのスレッドがある時点に実行されるかを決定する方法を確定しなければならない。そのために、マルチスレッドプログラミングが可能な典型的な言語として Java に着目し、Java 仮想マシンが、ある時点で実行されるべきスレッドをどのように決定するかという点を確認する。また、具体的な実行過程を付録に示す。

3.9.2 オペレーティングシステムによる違い

同一優先度のスレッドのスケジューリングに影響を与えるオペレーティングシステム固有のスケジューリングイベントがある [11] 。発生するタイミングによって、次に示す 3 つのタイプの Java スケジューリングイベントが考えられる。

1. 実行中のスレッドが実行可能状態でなくなるとき

実行中のスレッドがブロックしたり終了すると、Java 仮想マシンは他のスレッドを選んで実行中のスレッドとする。

2. 実行中のスレッドよりも優先度の高いスレッドが実行可能状態になったとき

これらのイベントは Java スケジューラのプリエンプティブモデルの中心となる。

3. 任意のタイマーが終了したとき

Java 仮想マシンは任意の時点でスケジューリングイベントを作成する場合がある。

3.9.3 本研究の実験でのスケジューリング

以上から本研究の実験でのスケジューリングを決定する。対象となるのは、ハードウェアでマルチスレッド処理の機構が組み込まれていない命令パイプラインを有するプロセッサである。これに対してはシミュレーションを容易にするためにタイプ 1 のスケジューリング方式を採用する。つまり、実行中のスレッドが終了し、実行可能状態でなくなるときに他のスレッドを選んで実行中のスレッドとする。

第4章　動作シミュレーション

4.1　目的及び内容

我々は提案したアーキテクチャが並列論理型言語の実行及び並列性を内在する各々の言語に有効であることを示すためにハンドコンパイルしたコードによるシミュレーションを4つ行なう。

1. タグ付き分岐命令のある場合とない場合のテストプログラムによるプロセッサの性能比較。
2. コードのスレッド数を増加させてスレッド数増加によるプロセッサの性能の飽和。
3. 同期処理を伴うコードの性能比較。
4. 並列処理が可能で特定の言語に依存しないテストプログラムによる性能比較を行なう。この特定の言語に依存しないプログラムとしては行列演算を用いる。

なお、本シミュレーションでは、マルチスレッド処理やJTD命令の効果を比較するため、通常のパイプラインプロセッサを用いた場合において、分岐遅延やフォワーディング等のバブルの発生を防止する方策は一切とらないこととする。

4.2　並列論理型言語を実行させた場合の測定

共通事項：

下記に示すテストプログラム終了条件に基づき、SECONDSスクリプトを実行させて、プログラムの実行に要した総クロック数を求める。

なお、ここでいうSECONDSとは、PARTHENONで用いるハードウェア記述言語SFL(Structured Function description Language)で記述された動作の流れを直接解釈して実行する、会話型のSFL動作シミュレータであり、SFL記述が、設計者の意図に一致しているかどうかを確認するために用いられている。

1. 条件：
   (a) パイプラインプロセッサの場合：
      i. 最後の命令まで到達した。
   (b) マルチスレッドプロセッサの場合：
      i. 全てのスレッドがcloseされた。

4.2.1 データ型判定を補助する命令及びマルチスレッド機構の効果

実験方法

ここではデータ型判定を補助する命令、すなわち JTD 命令の効果を把握する。そのための例題として必要なことは、データ型判定を伴う処理ということである。データ型判定を伴うには第 2 章で示したようにユニフィケーション処理が必要である。ユニフィケーション処理としては第 2 章で示した not 述語を用いた処理を使用する。また、ここではマルチスレッド機構の効果も把握する。そのための例題として必要なことは前章で示したようにパイプライン段数分の互いに独立したスレッドをパイプラインに投入できるようにすることである。パイプライン段数は 3 段としたので 3 つの互いに独立したスレッドをパイプラインに投入できれば良い。従って例題としては次に示す 2 つのことを含んだ処理とする。1 つは not 述語を用いた処理、もう 1 つは 3 つの互いに独立したスレッドをパイプラインに投入する処理である。並列論理型言語では第 2 章で説明したゴールがスレッドに相当しており、第 2 章で示したように AND 関係にもなっているボディ部の各々のゴールが 1 つ 1 つのスレッドとなる。以上の事から以下に示すプログラムを例題として用いる。なお、以降の並列論理型言語のシミュレーションにおいても主要な処理を同一にする。それは、主要な処理を共通にすることにより、JTD 命令の有無、スレッド数増加に伴う性能飽和及び同期処理の各々の実験に対する固有の結果を抽出させ、3 つの実験結果を統合的に見た総合的な評価を行なうためである。主要な処理は not 述語を用いる。

実験に用いたプログラム

```
:-module main.
main:-true|not(1,X),not(1,Y),not(1,Z).
not(In,Out):-In=0|Out=1.
not(In,Out):-In=1|Out=0.
```

1. コンパイルコードについて
   (a) WAM コード [36] を参考にして、並列論理型言語のコードをハンドコンパイルしてアセンブリコードにする。(以後のシミュレーションも同様にする。なお、アセンブリコードの一部を付録 C に示す。)
   (b) JTD 命令を使用しない場合は第 4 章で述べたようにデータ型の判定に AND 演算、比較演算、条件分岐を用いる。
   (c) プログラムの実行手順は第 2 章で述べたことを基本とする。(以後のシミュレーションも同様にする。)
   (d) パイプラインプロセッサのスレッドの実行手順は第 3 章で決定した通りとする。(以後のシミュレーションも同様にする。)

このプログラムは図 4.1 のように処理されていくこととする。

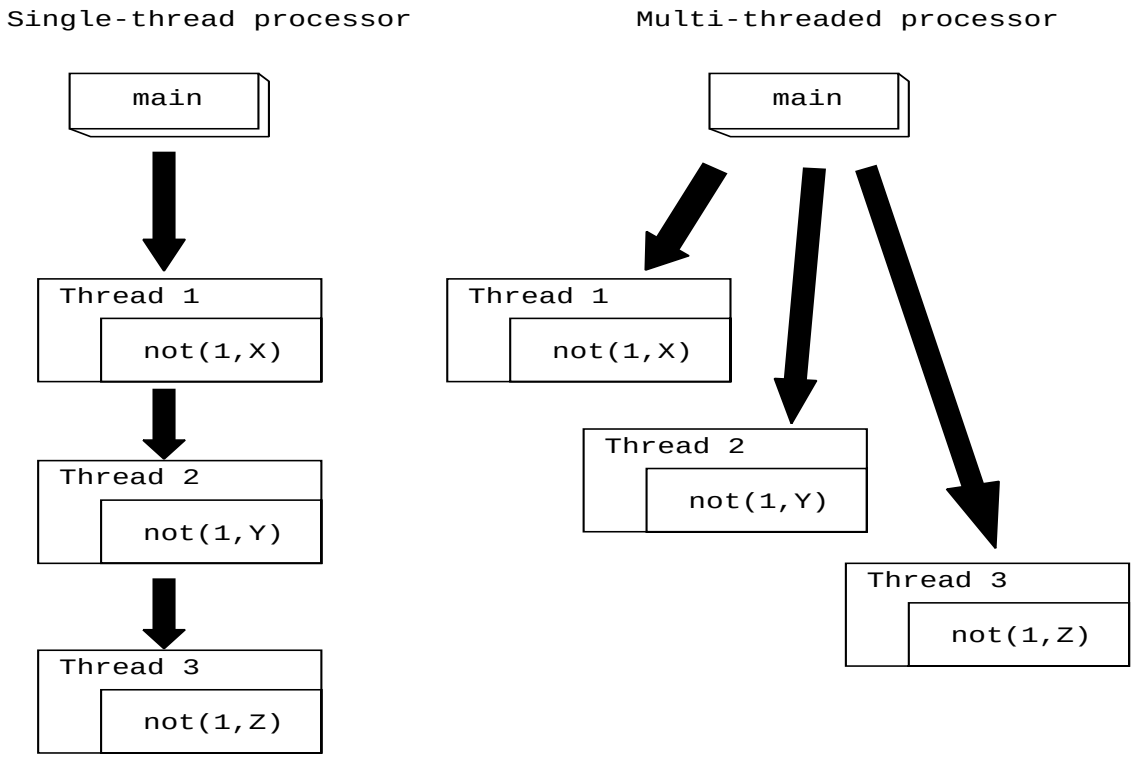
Single-thread processor
main
Thread 1
not(1,X)
Thread 2
not(1,Y)
Thread 3
not(1,Z)
Multi-threaded processor
main
Thread 1
not(1,X)
Thread 2
not(1,Y)
Thread 3
not(1,Z)

図 4.1: 実行の流れ

実験結果

図 4.2 はクロック・サイクル数による JTD 及びマルチスレッド機構の効果の比較を表す。図 4.3 はクロック・サイクル数削減率を基準とした速度向上比による JTD 及びマルチスレッド機構の効果の比較を表す。図 4.4 は CPI 値による JTD 及びマルチスレッド機構の効果の比較を表す。図 4.5 は CPI 値を基準としたパイプライン性能向上比による JTD 及びマルチスレッド機構の効果の比較を表す。なお、図 4.2、4.4 において図の中央から左側は JTD 命令を使用しない場合、右側は JTD 命令を使用した場合を示す。表 6 は各プロセッサでのテストプログラムの実行の所要クロック数及び CPI の結果を示す。なお、P は通常のパイプラインプロセッサ、M はマルチスレッドパイプラインプロセッサ、NoJTD は JTD 命令未使用、JTD は JTD 命令使用を各々示すことにし、以降の議論においても同様とする。表 7 は NoJTD と P を基準としたときの性能向上比の結果を示す。

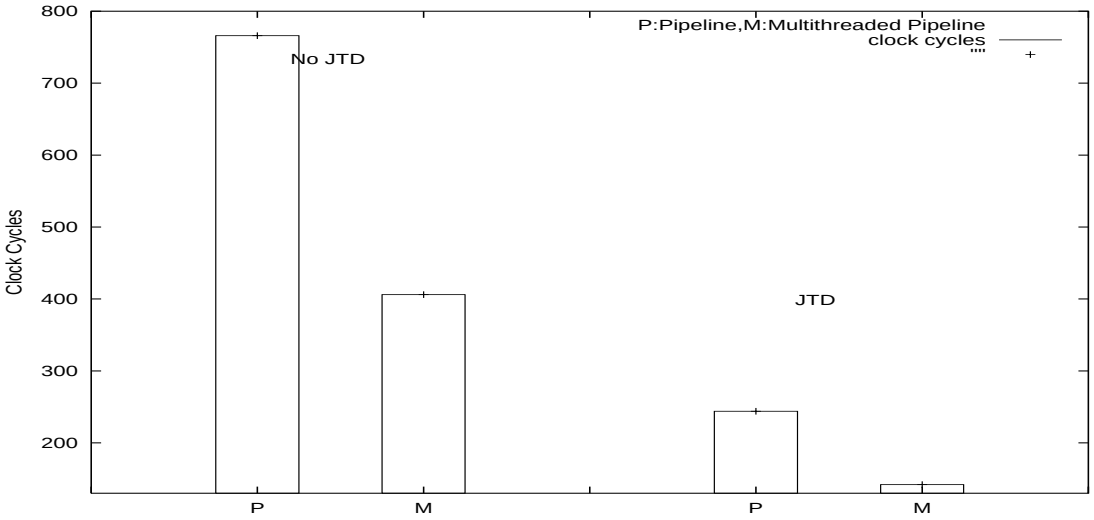

図 4.2: クロック・サイクル数による JTD 及びマルチスレッド機構の効果の比較

表 6:JTD 及びマルチスレッド機構の効果

方式	クロック数	CPI
NoJTD,P	766	1.9
NoJTD,M	406	1.0
JTD,P	244	1.8
JTD,M	142	1.1

表 7:JTD 及びマルチスレッド機構の効果 (性能向上比)

方式	性能比	
	クロック数	CPI
NoJTD,P	1.0	1.0
NoJTD,M	1.9	1.9
JTD,P	3.1	1.1
JTD,M	5.4	1.8

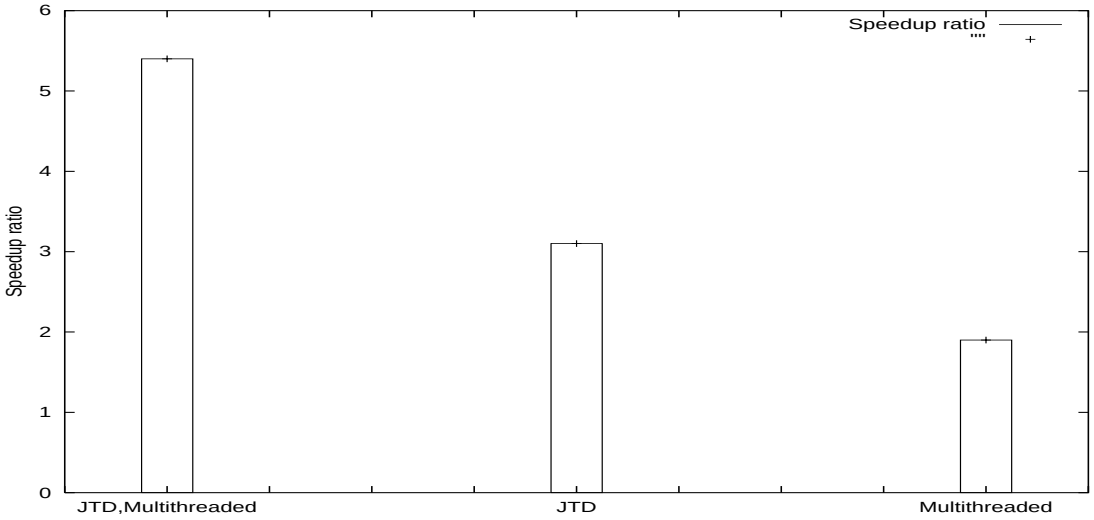

図 4.3: クロック・サイクル数削減率を基準とした速度向上比による JTD 及びマルチスレッド機構の効果の比較

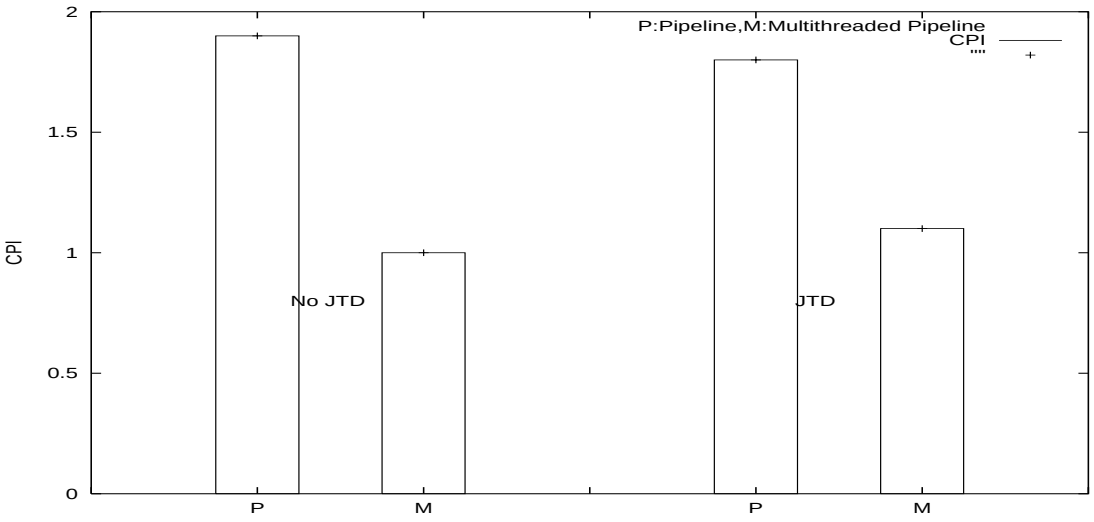

図 4.4: CPI 値による JTD 及びマルチスレッド機構の効果の比較

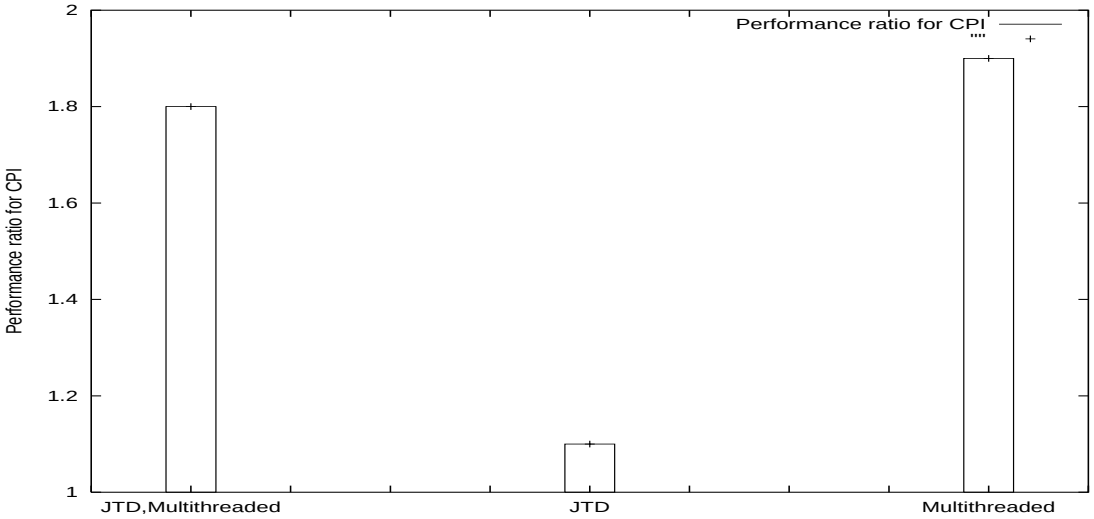

図 4.5: CPI 値を基準としたパイプライン性能向上比による JTD 及びマルチスレッド機構の効果の比較

4.2.2 スレッド数を増加させたときの性能の飽和

実験方法

ここでは実行可能なスレッド数を増加させていったときのプロセッサの特性を把握する。そのための例題としてはスレッド数の増加分の変化に対するプロセッサの特性を確認していくために同じ大きさの実行可能なスレッド数を増加させていったときの特性を把握する。プログラムを実行させるための最小のスレッド数は 1 つであり、パイプライン段数が 3 段のマルチスレッドパイプラインプロセッサに対して最高性能を得るにはスレッド数は 3 つ必要である。従って実行可能なスレッド数を増加させていったときに対するマルチスレッドパイプラインプロセッサの最高性能を得るまでの変化を把握するには、スレッド数を 1 から 3 まで変化させていったときの特性を確認する必要がある。また、この変化が周期的に繰り返されていくものであるかを確認し、かつ、パイプライン段数以上のスレッド数を処理させたときの特性を把握するには、スレッド数は 1 から 6 までの場合を確認する必要がある。7 個以上のスレッドの実行に対しては、1 から 6 個までの場合の結果から理論的に拡張できるものとして確認しない。以上の事から以下に示すプログラムを例題として用いる。なお、スレッドスケジューリングは通常のパイプラインプロセッサの場合は次のように行なう。実行中のスレッドが終了し、実行可能状態でなくなるときに他のスレッドを選んで実行中のスレッドとする。マルチスレッドパイプラインプロセッサの場合は次のように行なう。ハードウェアで用意しているレジスタファイルや各種フラグは 3 つなので、同時に時分割多重で並列実行可能なスレッド数は 3 つまでである。従って、実行するスレッド数が 3 つの場合は 3 つをスレッドに投入し、各スレッドはクロックサイクル毎に切替えて処理する。実行するスレッド数が 1 つの場合は 1 つをスレッドに投入し、空

いた残りの 2 つのスレッド処理機構には何も投入しないことにし、投入されたスレッドはパイプライン処理をしないことにする。実行するスレッド数が 2 つの場合は 2 つをスレッドに投入し、空いた残りの 1 つのスレッド処理機構には何も投入しない。実行するスレッド数が 4 つ以上の場合は 3 つのスレッドまでを時分割多重で並列実行し、残りのスレッドは実行中のスレッドの中で終了し、実行可能状態でなくなったスレッドに対して、この残りのスレッドを順次投入する。

実験に用いたプログラム

```
:-module main.
main:-true|  (1).
not(In,Out):-In=0|Out=1.
not(In,Out):-In=1|Out=0.
```

なお、(1) は下記の表 8 の通りとする。

表 8：コンテキスト数による (1) のコード

コンテキスト数	ソースコード
1 個の場合	not(1,X)
2 個の場合	not(1,X),not(1,Y)
3 個の場合	not(1,X),not(1,Y),not(1,Z)
4 個の場合	not(1,X),not(1,Y),not(1,Z),not(1,U)
5 個の場合	not(1,X),not(1,Y),not(1,Z),not(1,U),not(1,V)
6 個の場合	not(1,X),not(1,Y),not(1,Z),not(1,U),not(1,V),not(1,W)

このプログラムは図 4.6 のように処理されていくこととする。

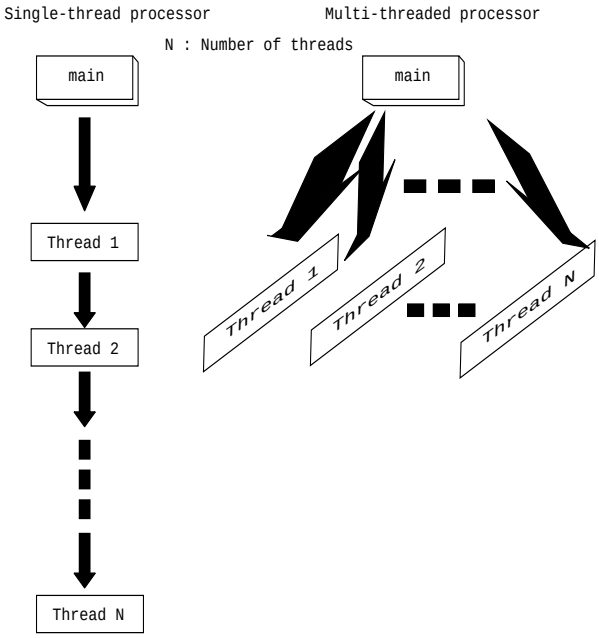

図 4.6: 実行の流れ

実験結果　図 4.7 はコンテキスト数を増加させたときにおけるマルチスレッド機構の効果によるクロック・サイクル数の比較を示す。図 4.8 はコンテキスト数を増加させたときにおけるマルチスレッド機構の効果による CPI 値の比較を示す。表 9 はコンテキスト数を増加させたときのマルチスレッド機構の効果の結果を示す。

表 9：コンテキスト数を増加させたときのプログラムの実行に要したクロック数及び CPI 値の結果

	方式	コンテキスト数					
		1	2	3	4	5	6
クロックサイクル数	P	78	164	244	324	404	484
	M	130	139	142	275	277	282
CPI 値	P	1.8	1.8	1.8	1.8	1.8	1.8
	M	2.9	1.5	1.1	1.5	1.2	1.0

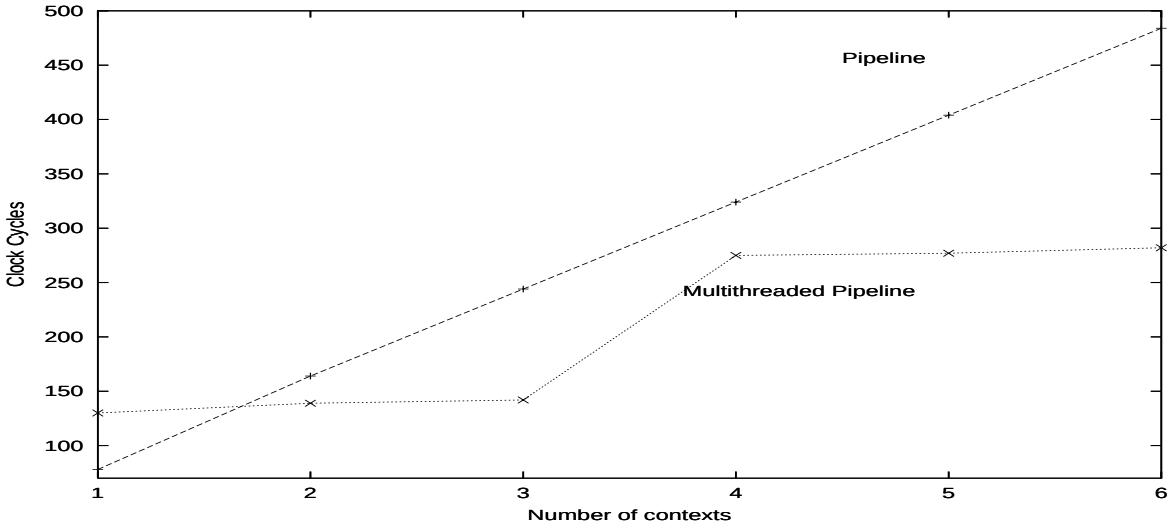

図 4.7: コンテキスト数を増加させたときにおけるマルチスレッド機構の効果によるクロック・サイクル数の比較

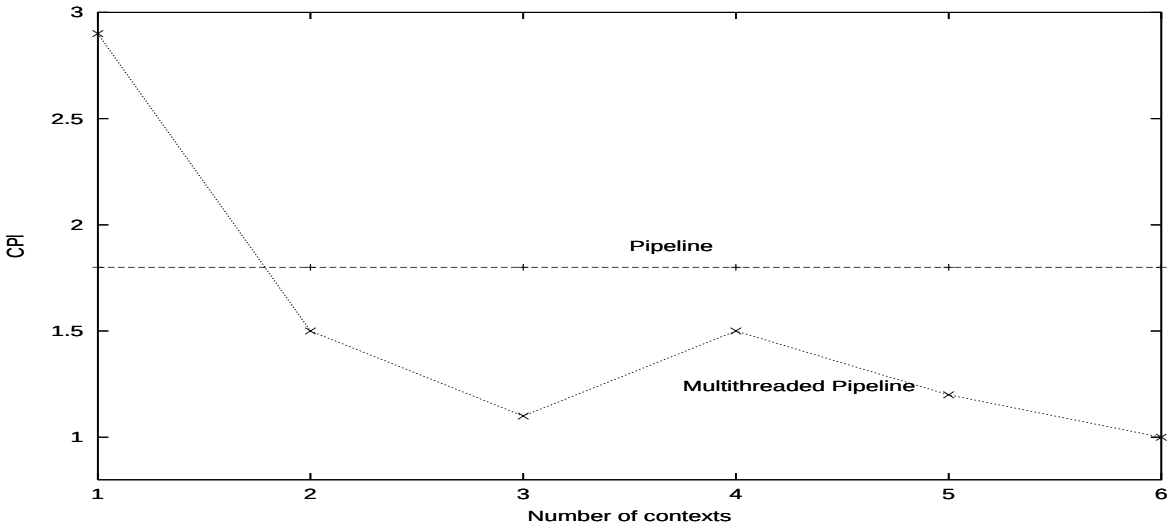

図 4.8: コンテキスト数を増加させたときにおけるマルチスレッド機構の効果による CPI 値の比較

4.2.3 方式による同期処理の性能差

実験方法

ここでは同期処理を伴うプログラムに対するプロセッサの特性を把握する。そのための例題としては、少なくとも 2 つのゴール間で同期・通信を行なう特性を確認しなければならない。また、マルチスレッド処理による同期処理の性能の改善策は同期処理のオーバヘッドの隠蔽である。従って、例題としては 2 つのゴール間で同期・通信を行うものとして、かつ、それらの同期処理に伴うコンテキスト・スイッチの処理を隠蔽化できるだけの並行して処理可能なスレッド数があるものとする。以上の事から以下に示すプログラムを用いる。

実験に用いたプログラム

```
:-module main.
main:-true|not(X,Y),not(0,X),not(1,Z),not(1,U).
not(In,Out):-In=0|Out=1.
not(In,Out):-In=1|Out=0.
```

ここで X は共有変数とする。従って、not(X,Y) と not(0,X) の X は同じ変数を参照し、not(0,X) の X が具体化されるまで、not(X,Y) は中断し続ける。

このプログラムは図 4.9 のように処理されていくこととする。

実験結果 図 4.10 は同期処理を伴う場合のクロック・サイクル数によるマルチスレッド機構の効果の比較を示す。図 4.11 は同期処理を伴う場合の CPI 値によるマルチスレッド機構の効果の比較を示す。表 10 は各プロセッサでのテストプログラムの実行のための所要クロック数、CPI 値及び P の実行結果を基準とした性能向上比を示す。

表 10:クロック・サイクル数及び CPI 値の結果及び性能比

方式	クロック数	CPI 値	性能向上比	
			クロック数	CPI 値
P	807	1.9	1.0	1.0
M	542	1.2	1.5	1.5

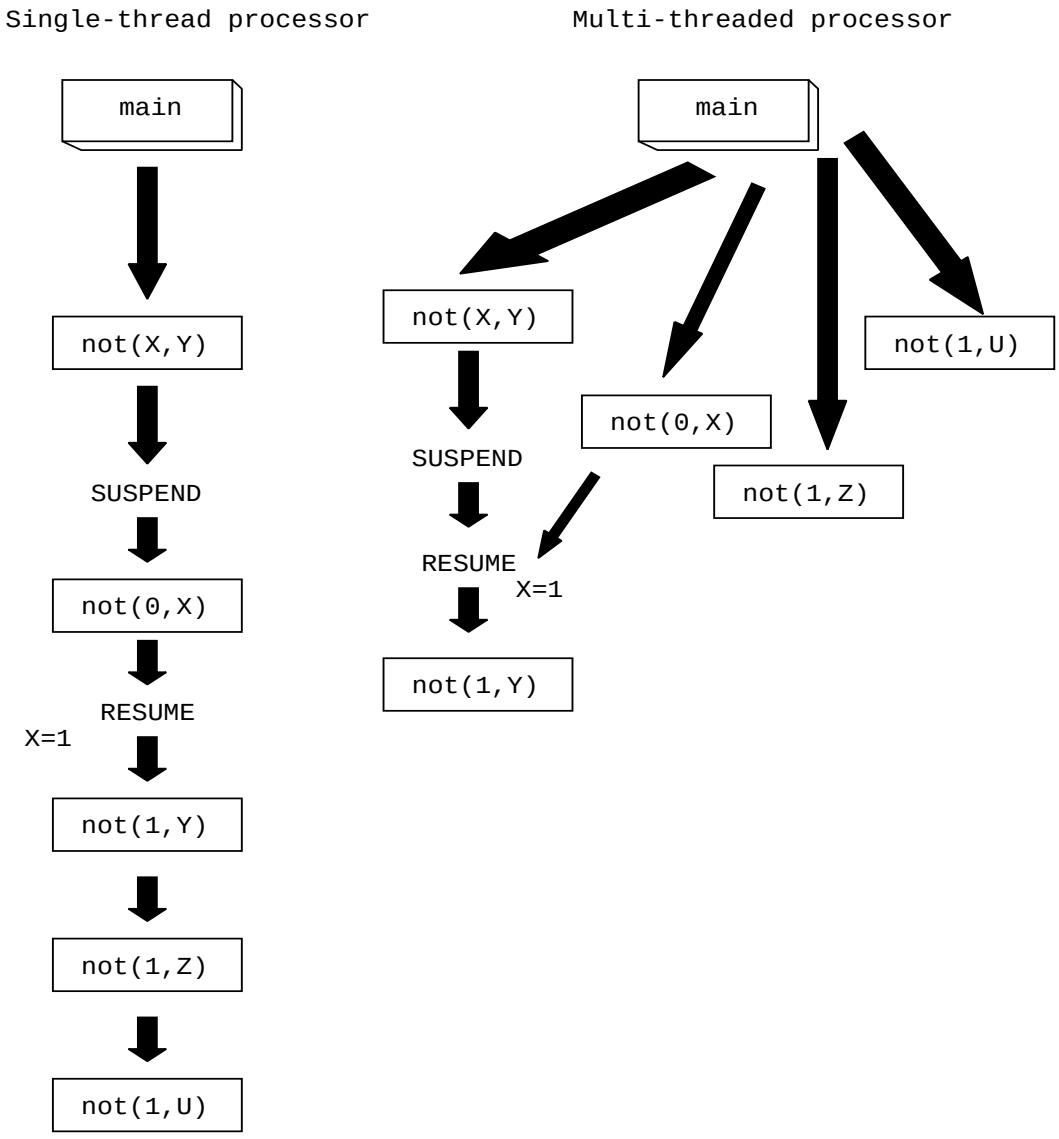
Single-thread processor
Multi-threaded processor
main
not(X,Y)
SUSPEND
not(0,X)
RESUME
X=1
not(1,Y)
not(1,Z)
not(1,U)
main
not(X,Y)
not(0,X)
not(1,U)
not(1,Z)
SUSPEND
RESUME
X=1
not(1,Y)

図 4.9: 実行の流れ

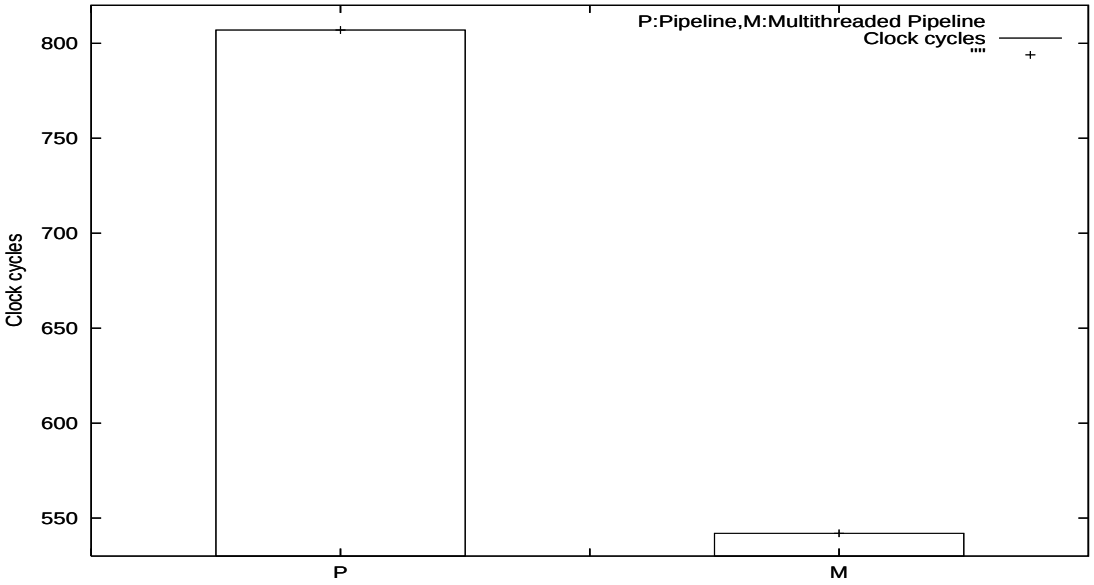

図 4.10: 同期処理を伴う場合のクロック・サイクル数によるマルチスレッド機構の効果の比較

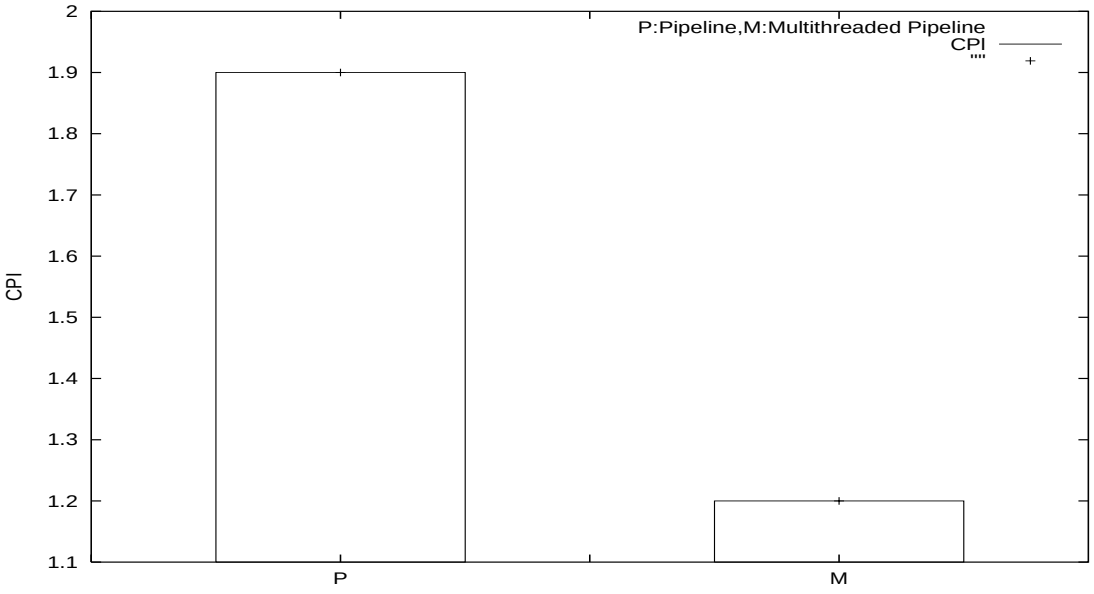

図 4.11: 同期処理を伴う場合の CPI 値によるマルチスレッド機構の効果の比較

4.3 並列論理型言語以外で並列性を有するコードを実行させた場合の測定

4.3.1 実験方法

並列に処理を行なうことが可能なテストプログラムとしてここでは行列積の計算を行なうプログラムを作成した。この実験では、並列論理型言語のような言語特有の処理を考慮したプログラムとはしない。対象とする言語は命令型言語としてハンドコンパイルしたコードを対象とする。

行列積の計算は、行列 a[I][K] と b[K][J] の積を c[I][J] とすると、c の各要素は図 4.12 のように計算することが出来る。図からも明らかなように c の各要素の計算は独立しているので、並列処理が可能となる。

$$c[i][j] = \sum_{k=0}^{K-1} a[i][k] \times b[k][j]$$

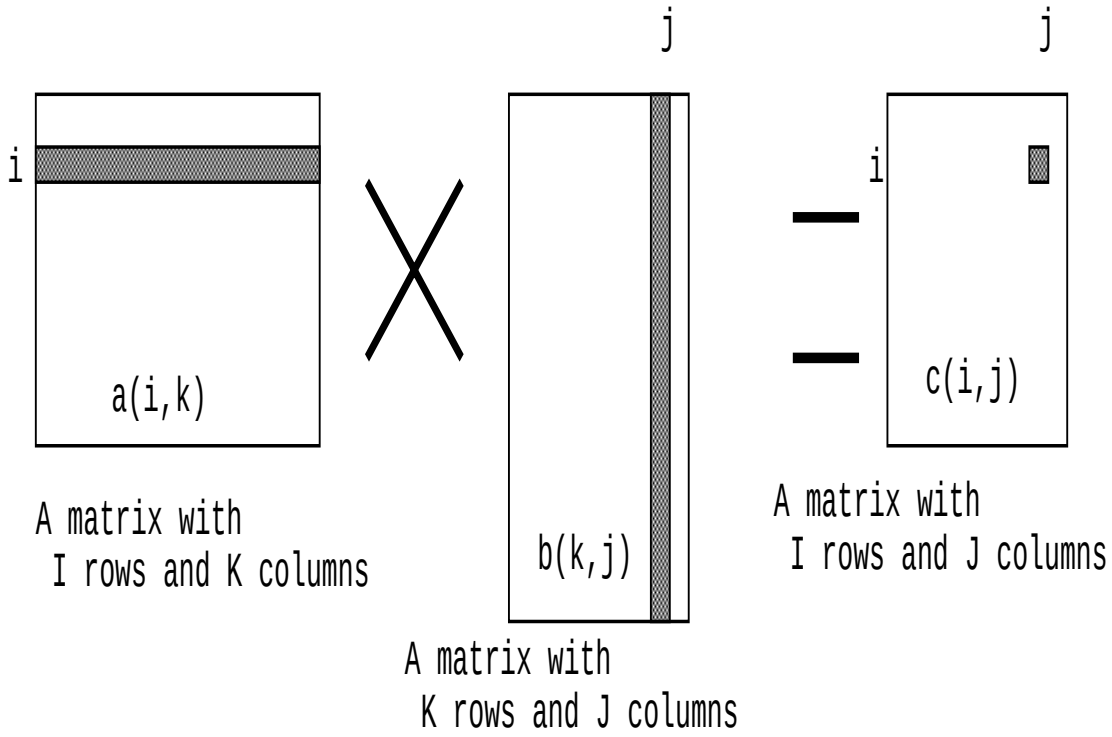

図 4.12: 行列積の計算

いまAを$m \times n$、Bを$n \times p$とすればCは$m \times p$の行列となる。このときCの(i,j)要素C_{ij}は

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} (i = 1, \cdots, m; j = 1, \cdots, p)$$

と表せる。つまり行列Cの計算には3重のループを使用することがわかる。ループをどのように回せば速いのかは処理系によって変わる。ここではまず全ての組み合わせ3!の組み合わせ、6通りをC言語で実装し、その中で最も速いものをシミュレーションのアルゴリズムに適用する。手続き的に考えると以下のことを実行する。

表11：ループを回す手順(その1)

型	ijk	jik	kij
実行手順	for i=1 to m for j=1 to p for k=1 to n $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$ end end end	for j=1 to p for i=1 to m for k=1 to n $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$ end end end	for k=1 to n for i=1 to m for j=1 to p $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$ end end end

表12：ループを回す手順(その2)

型	kji	ikj	jki
実行手順	for k=1 to n for j=1 to p for i=1 to m $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$ end end end	for i=1 to m for k=1 to n for j=1 to p $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$ end end end	for j=1 to p for k=1 to n for i=1 to m $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$ end end end

ijk型、jik型は手計算でやるときの内積型である。内積型は次のように演算すると主記憶とのやりとり回数が減少して性能が改善される。

表 13：ループを回す手順 (その 3)

型	ijk'	jik'
実行手順	for i=1 to m for j=1 to p s=0 for k=1 to n $s = s + a_{ik} \times b_{kj}$ end $c_{ij} = s$ end end	for j=1 to p for k=1 to n s=0 for i=1 to m $s = s + a_{ik} \times b_{kj}$ end $c_{ij} = s$ end end

右辺の c_{ij} や a_{ij}、b_{ij} はそれぞれメモリ領域に読みとりに行き、左辺の c_{ij} はメモリ領域に書き込みに行く。一方、書換えた ijk' 型や jik' 型は変数 s を新たに作り c_{ij} の値が決定するまで c_{ij} を読み取りにも書き込みにも行かない。よって、一般的には計算量が減少すると考えられる。

しかし、処理系によって効果は大きく異なる。Pascal では行方向のメモリ割り当てをするので kij 型、ikj 型が良く、Fortran コンパイラでは列方向のメモリ割り当てをするので kji 型、jki 型が良い [51]。また、メモリ割り当て方によってキャッシュ効率も変わる。連続した番地に割り当てられている場合はアクセス効率は良いが、そうでない場合はアクセス効率は良くない。従って連続した番地に割り当てられていない場合はキャッシュの効果は望めない [50] 。そのため、行方向にメモリ割り当てしている場合は、列方向のアクセスに対してはキャッシュ・ミスが頻発し、列方向にメモリ割り当てしている場合は、行方向のアクセスに対してはキャッシュ・ミスが頻発する。

以下で C 言語による処理時間を計測して求めた実行時間の相対値を示す。

1. 計測条件
   (a) gcc コンパイラの最適化レベルは全て -O2
   (b) vmstat で CPU の空き比率が 90%を超えている
   (c) ijk 型、jik 型はそれぞれ改良型の ijk' 型、jik' 型で行なう
2. 計測及びデータ処理方法
   (a) time で 10 回ずつ計測
   (b) 計測したデータの平均値を相対値にする

図 4.13 より本研究での行列演算による動作シミュレーションではループを回す手順は ijk' 型と決定する。

次に、決定したアルゴリズムに基づいてシミュレーションを行なう。このとき、上記で述べたキャッシュ・ミスは今までと同様に考慮に入れないものとする。ここでは C 言語と

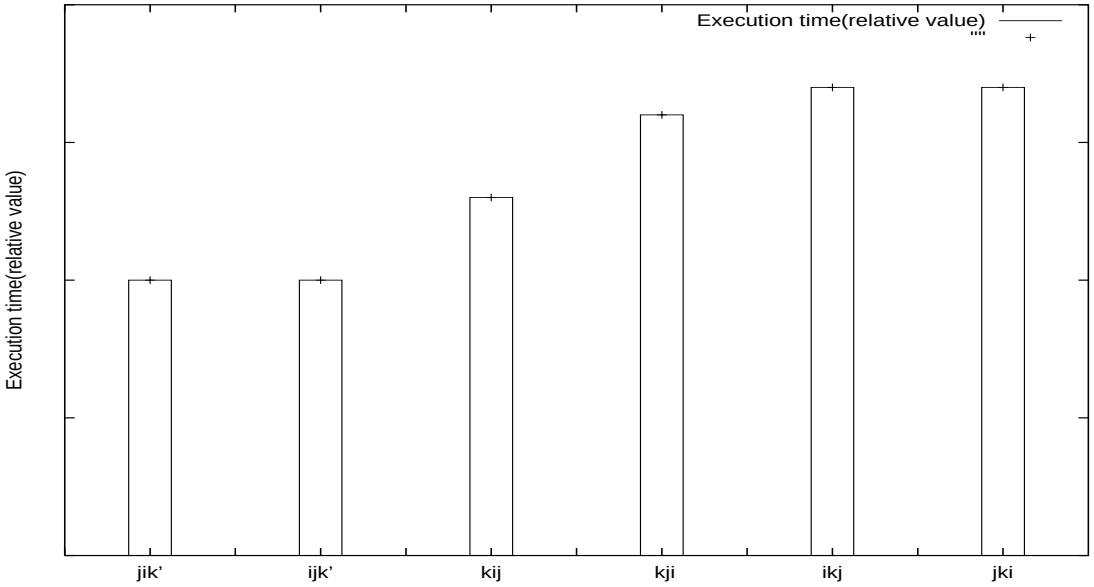

図 4.13: ループの順番別の整数型 100×100 行列の積演算の実行時間 (相対値)

同様に行方向優先でメモリ上に割り当てるものとする。なお、本研究で提案したプロセッサは簡易プロセッサであり、掛け算命令は実装していないが、掛け算についてはシフト命令で代用する。

掛け算の処理は以下のように行なう。

まず、次の式のように展開する。

$$A \times B = A \times (b_{31} \times 2^{31} + b_{30} \times 2^{30} + \cdots + b_0 \times 2^0) = A \times b_{31} \times 2^{31} + \cdots + A \times b_0 \times 2^0$$

A は掛けられる数、B は掛ける数、b_n は掛ける数の n ビット目を示す。この式は下位ビットから調べていく方法と上位ビットから調べていく方法の 2 種類ある。それぞれ次のような特徴を持つ。

1. 下位ビットから調べる方法
   (a) 計算する数によってはループの数を減らす事が可能となる。
2. 上位ビットから調べる方法
   (a) 計算する数にかかわらず同じループ回数を必要とする。

本研究の動作シミュレーションにおいては掛け算はシフト命令で代用する。また、計算量を同じにするために上位ビットから調べていく方法を用いることにする。並列処理を行なうための分割は、i のループを m 個に分割することにより、m 個の依存関係の存在しないプログラムとする。それをマルチスレッドプロセッサではマルチスレッド処理を行ない、シングルスレッドプロセッサでは逐次処理を行なう。実行の流れを図 4.14 で示す。

なお、シミュレーションではA=$\begin{pmatrix} 4 & 4 & 4 \\ 4 & 4 & 4 \\ 4 & 4 & 4 \end{pmatrix}$、B=$\begin{pmatrix} 4 & 4 & 4 \\ 4 & 4 & 4 \\ 4 & 4 & 4 \end{pmatrix}$のときのA×Bを求める演算にする。従ってm=3となり、3個の依存関係の存在しないプログラムを処理することとなる。

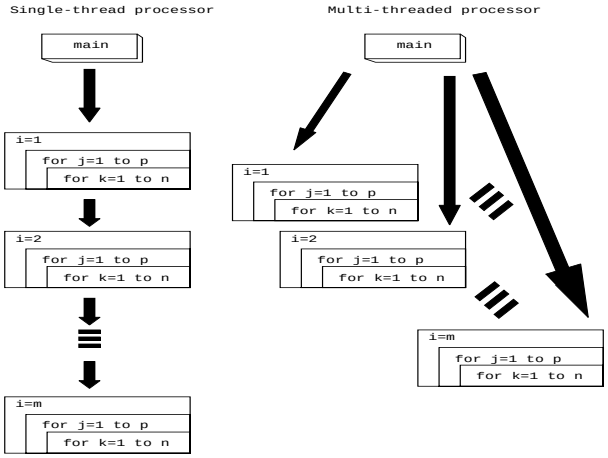

図 4.14: 実行の流れ

4.3.2 実験結果

図4.15は行列積で並列処理を行なった場合のクロック・サイクル数によるマルチスレッド機構の効果の比較を示し、図4.16は行列積で並列処理を行なった場合のCPI値によるマルチスレッド機構の効果の比較を示す。表14は各プロセッサでのテストプログラムの実行に要したクロック・サイクル数、CPI値及びP の実験結果を基準とした性能向上比を示す。

表14：実行結果及び性能比

方式	クロック数	CPI値	性能向上比	
			クロック数	CPI値
P	32816	1.7	1.0	1.0
M	19191	1.0	1.7	1.7

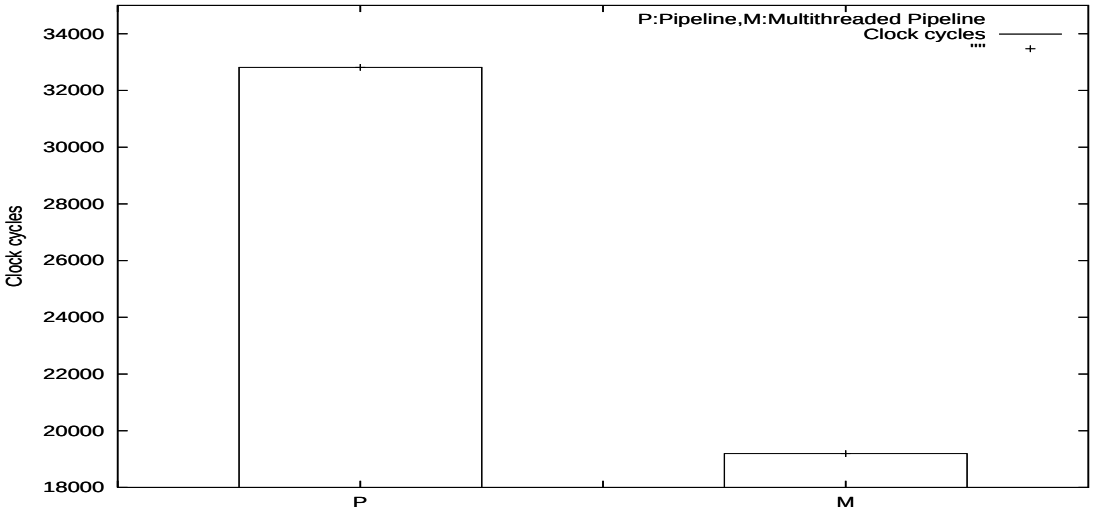

図 4.15: 行列積で並列処理を行なった場合のクロック・サイクル数によるマルチスレッド機構の効果の比較

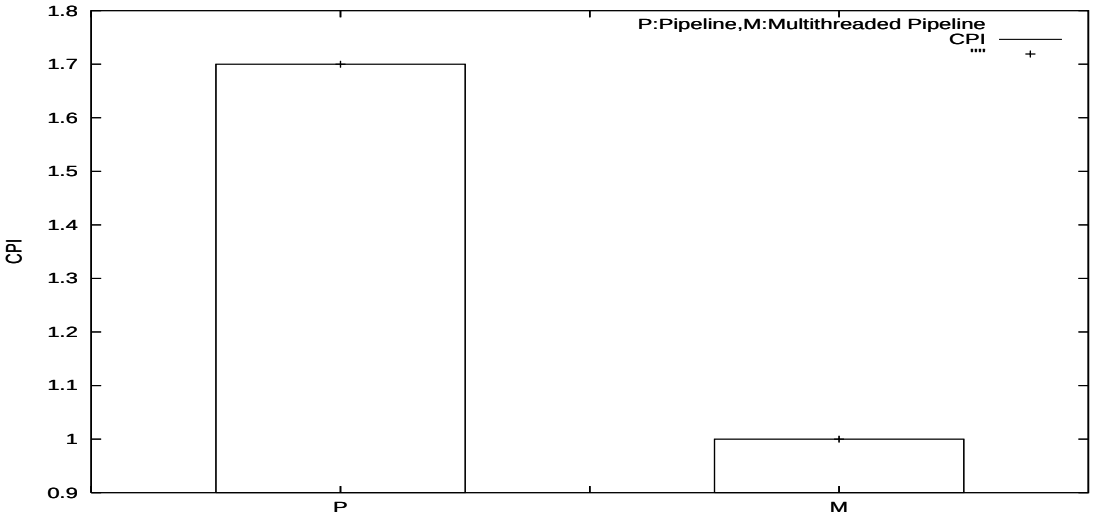

図 4.16: 行列積で並列処理を行なった場合の CPI 値によるマルチスレッド機構の効果の比較

第5章　考察

5.1　並列論理型言語を実行させた場合の測定

通常のパイプラインプロセッサで並列論理型言語を実行させた場合、動的なデータ型の判定のためにAND演算、比較演算、条件分岐が頻出し、処理効率が低下する。データ型の判定にJTD命令を使用した場合、プログラムの実行に要するクロック・サイクル数の削減は可能だが分岐ハザードによってパイプラインは乱されることになる。ソースコードに並列実行可能性が存在したとしても、逐次処理を基本としており並列実行による処理効率の向上は望めない。さらに並列論理型言語特有の同期処理は頻繁に起こりうるので同期処理に伴うコンテキスト切替えのための処理に多くを占めるようになり主要な処理の実行は頻繁に妨げられる。

また、伊藤ら [22] によってマルチスレッド型プロセッサ・アーキテクチャと関数型プログラムの特徴を組み合わせることにより並列処理のための高性能なマルチスレッド型プロセッサ・アーキテクチャが提案され、ゲートレベルでの論理合成を行なうことにより性能見積りが行なわれた。その結果、プロセスルールとして0.1μmの技術を仮定すると1GHzで動作可能となることが確認された。

また、今回作成した命令セットにJTD命令を追加したマルチスレッド型パイプラインプロセッサにおいては以下のような結果となった。

データ型判定を補助する命令であるJTD命令及びマルチスレッド処理機構の効果を通常のパイプラインプロセッサと比較した結果を図4.2-4.5に示す。図4.2はプログラムの実行に要したクロック・サイクル数に基づいた比較の結果であり、図4.3はプログラムの実行に要したクロック・サイクル数の削減率に基づいた比較の結果であり、図4.4はCPI値に基づいた比較の結果であり、図4.5はCPI値に基づいたパイプラインの性能向上比による結果である。

この結果から明らかなようにJTD命令ではCPI値の性能向上が約1.1倍となっておりパイプラインの性能向上はほとんどなされないがJTD命令によってクロック・サイクル数は約70%削減されて約3.1倍性能向上するので総合性能は約3.1倍改善される。一方、マルチスレッド処理機構によってCPI値は約1.0となり、性能比では約1.9倍パイプラインの性能が向上しており、クロック・サイクル数は約47%削減されて約1.9倍性能向上するので総合性能は約1.9倍改善される。JTD命令とマルチスレッド処理機構の両方を用いることによって、CPI値は約1.1となり、性能比では約1.8倍パイプラインの性能が向上しており、クロック・サイクル数は約81%削減されて約5.4倍性能向上するので総合性能

は約 5.4 倍改善される。

よって、JTD 命令によって多方向分岐のための命令数は削減され、マルチスレッド処理によって JTD 命令の分岐ハザードやその他のパイプライン・ハザードの隠蔽化及びテストプログラム内に存在する複数スレッドの並列実行が行なわれたことが考えられる。以上の結果より、JTD 命令及びマルチスレッド処理機構が並列論理型言語の効率実行化につながる可能性が確認された。これらのことから次のことが考えられる。

1. JTD 命令によってユニフィケーションに必要な命令を削減させ、総クロック数を約 3 分の 1 に削減させた。
2. マルチスレッド処理によってパイプライン・ハザードを隠蔽させ、CPI 値は約 2 分の 1 に削減されて、CPI 値は約 1.0 になり、パイプラインの性能を向上させた。
3. JTD 命令及びマルチスレッド処理機構の導入により、全体として約 5.4 倍性能が向上される。
4. クロック数で比較すると JTD 命令で約 1.9 倍、マルチスレッド処理で約 3.1 倍性能向上されるにもかかわらず、両方を用いた場合の総合的な性能向上は約 5.4 倍で、$1.9 \times 3.1 \approx 5.9$ 倍とならない。

全体的な性能向上が約 5.9 倍とならない理由を次に示す。

まず、上記 1.、2. について具体的にどれくらいの効果があったかを統計を取り確認をした。その図を図 5.1、5.2、表 15、16 に示す。

なお、図 5.1、5.2 において図の中央から左側は JTD 命令を使用しない場合、右側は JTD 命令を使用した場合を示す。

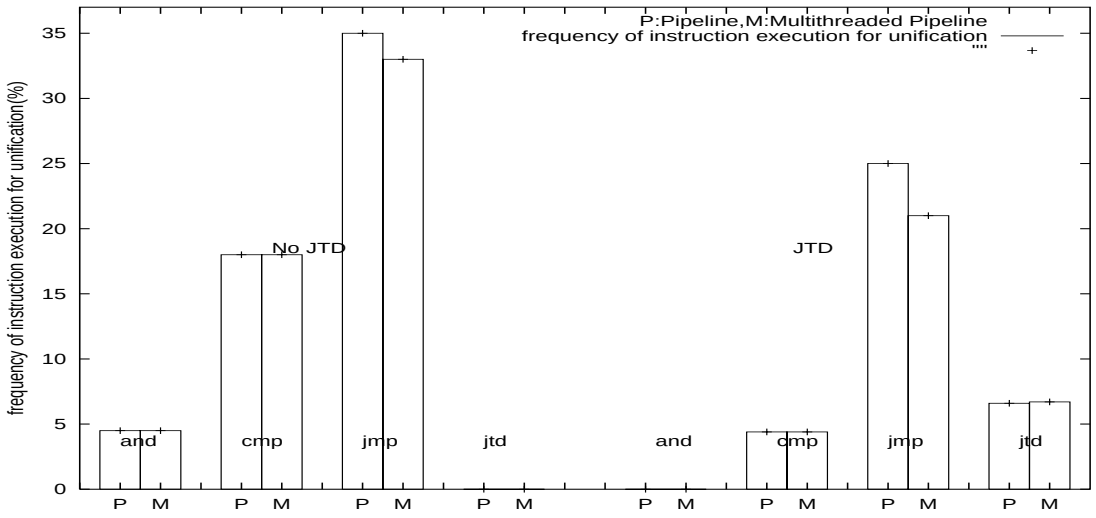

図 5.1: ユニフィケーションに必要な命令の実行頻度による JTD 及びマルチスレッド機構の効果の比較

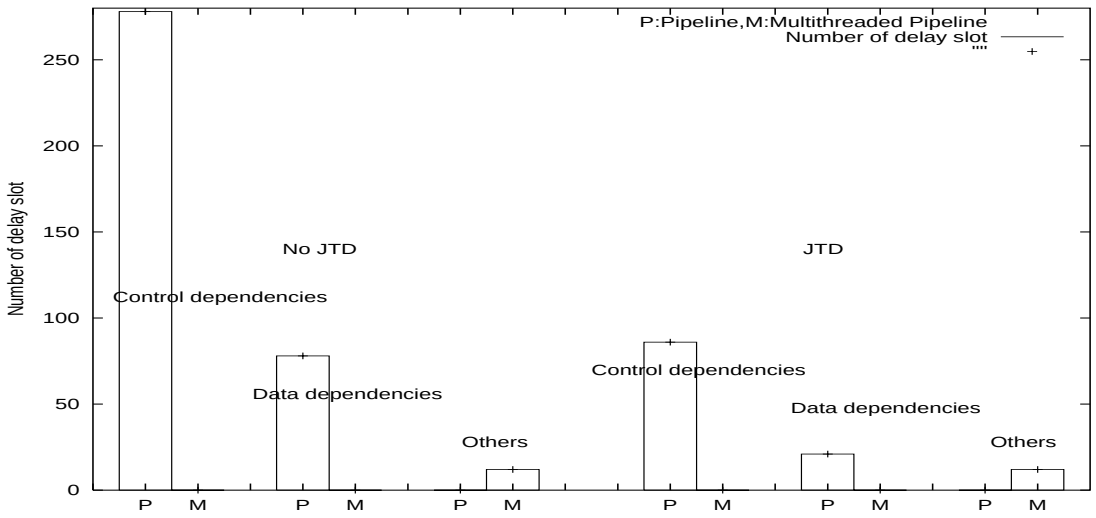

図 5.2: パイプラインバブルによる JTD 及びマルチスレッド機構の効果の比較

表 15 ： ユニフィケーションに必要な命令の実行頻度

命令の種類	方式	実行頻度 (%)
and	NoJTD,P	4.2
	NoJTD,M	4.0
cmp	NoJTD,P	20
	NoJTD,M	20
jmp	NoJTD,P	33
	NoJTD,M	32
jtd	NoJTD,P	0
	NoJTD,M	0
and	JTD,P	0
	JTD,M	0
cmp	JTD,P	12
	JTD,M	13
jmp	JTD,P	21
	JTD,M	20
jtd	JTD,P	5.5
	JTD,M	4.8

表 16 ： パイプラインバブル

パイプラインバブル発生原因	方式	パイプラインバブル発生数
分岐命令	NoJTD,P	280
	NoJTD,M	0
データ依存	NoJTD,P	99
	NoJTD,M	0
その他	NoJTD,P	0
	NoJTD,M	12
分岐命令	JTD,P	88
	JTD,M	0
データ依存	JTD,P	24
	JTD,M	0
その他	JTD,P	0
	JTD,M	12

図 5.1 はユニフィケーションに必要な命令の実行頻度による JTD 命令及びマルチスレッド処理機構の効果を比較した結果であり、図 5.2 はパイプラインバブルによる JTD 命令及びマルチスレッド処理機構の効果を比較した結果である。

この結果から、JTD 命令によって AND 演算は約 4.2%、比較演算は約 8%、条件分岐は約 12%削減されている。一方、マルチスレッド処理によってパイプラインバブルの発生は

一部を除いて存在していない。この一部に関しては、以下のことが原因と考えられる。スレッド毎に分配している部分の実行はパイプライン処理されていない。それぞれのスレッドの処理は同時には終了しない。ただし、上記 2 点については実行されるスレッドの大きさが十分にあれば無視できるほどの投入数になると考えられる。

また、通常命令も JTD 命令も同様に分岐ハザードが生じるが、JTD 命令の導入により分岐命令の出現頻度が減ると同時に、データ型を決定させるタグ判定のための AND、CMP 命令も減っている。このことは結果的に CPI 値を求めるときの分母の命令数が、クロック・サイクル数が減る以上に減少したためであると考えられる。その結果、CPI 値で比較した場合のマルチスレッド導入の効果も JTD 命令の有無で約 1.9 倍から約 1.8 倍に低下している。実際に表 17、18 のような削減効果となった。

表 17：削減方策が JTD 命令の場合の削減比率

	方式	
	M	P
クロック数	2.9	3.1
実行命令数	3.0	2.9

表 18：削減方策がマルチスレッドの場合の削減比率

	方式	
	NoJTD	JTD
クロック数	1.9	1.7
実行命令数	1.0	1.0

表 17、18 から明らかなように JTD 命令とマルチスレッド処理の両方を用いるとクロック数が削減される比率よりも実行命令数が削減される比率が高くなり、CPI 値を求めるときの分母が他の場合と比較して小さくなっている。さらに実験結果から、NoJTD,M と JTD,M では JTD,M の方がクロック数は少ないが CPI 値が高く、両方使用した方がパイプラインの性能改善度が低い。これはマルチスレッドパイプラインプロセッサの性能改善度はパイプラインハザード数に比例して向上するためである。すなわちパイプラインハザード数が多いほど、マルチスレッドによる性能改善度が高い。

NoJTD,M のマルチスレッド効果は NoJTD,P の性能改善度であり、JTD,M のマルチスレッド効果は JTD,P の性能改善度である。実際に比較すると

表 19:各プロセッサでのパイプラインハザード

方式	コントロールハザード数	データハザード数	合計のハザード数
NoJTD,P	278	78	356
JTD,P	86	21	107

表 19 からも明らかなようにパイプラインハザード数が少ないほどマルチスレッド処理の効果は少なくなるので、NoJTD,M に対して JTD,M の方が CPI 値の性能改善度が低くなっ

ている (CPI 値でみると約 9%性能改善度が低下する)。従って特定の言語向きの命令を加えることでパイプラインハザードも減るがハザードが減るとマルチスレッドの効果も下がることになる。しかし、クロック数の大幅削減が可能となる。すなわち、高能率命令の導入はクロック数を改善すると同時に命令数も減少させるので CPI 値が単純には改善しないが、総クロック数は確実に減少するので総合性能は必ず改善されると考えられる。これらのことから、パイプラインハザードを減らす処理機構を追加するとマルチスレッドによる性能改善度がやや低くなるが、クロック数は大幅に削減でき、総合性能は改善されることがいえる。

次にスレッド数を増加させたときの性能の飽和に関して比較した結果を図 4.7、4.8 に示す。図 4.7 はコンテキスト数を増加させたときにおけるマルチスレッド機構の効果をクロック・サイクル数に基づいて比較した結果であり、図 4.8 はそれを CPI 値に基づいて比較した結果である。

この結果から、通常のパイプラインプロセッサではコンテキスト数の増加に関わらず CPI 値はほぼ均一であり、プログラムの実行に要するクロック・サイクル数は一定の割合で増加していく。一方、マルチスレッドパイプラインプロセッサでは CPI 値は並列実行可能なコンテキストが存在しない場合は通常のパイプラインプロセッサよりも性能が改善されず、存在する場合は性能は改善され、パイプラインの段数の整数倍毎に CPI 値は約 1.0 になる。

ただしこのことはスレッドスケジューリングと関連している。以降のスレッド数を増加させていったときの実験結果の考察においても実験手順で述べたスレッドスケジューリングに依存している。

プログラムの実行に要するクロック・サイクル数はパイプライン段数が 1-3 間では平均して約 137 であり、パイプライン段数が 4-6 間では平均して約 278 であり、パイプライン段数 $\times N'(N' = 1, 2, \cdots) + 1$ 毎にクロック数は大幅に増大する。

よって、$3 \times$ N(N:整数) 毎に性能向上が飽和、つまりパイプライン段数 $\times$ N 毎に性能は飽和する。また、独立して、かつ、並行に処理できるスレッドが 2 つ以上存在すれば、マルチスレッド処理を利用した方がパイプラインの効率は良いことが考えられる。以上の結果より、2 つ以上の並列実行可能なコンテキストが存在した場合及びパイプラインステージが全てパイプラインの段数と同数のスレッドで満たされた場合ではマルチスレッド処理がパイプラインの高性能化につながる可能性が確認された。さらに、パイプライン段数 $\times N'(N' = 1, 2, \cdots) + 1$ 毎にクロック数が大幅に増していることからパイプラインの段数をさらに細分化し、細分化した分の数だけのお互いに独立したスレッドをマルチスレッドパイプラインに投入できれば、CPI 値は約 1.0 のまま、細分化した分だけプログラムの実行に要するクロック・サイクル数はほぼ一定の値となることにより、クロック・サイクル数の大幅な増大を防ぐことが可能となるだけでなく、パイプライン動作のスループット効果の大幅な向上につながる可能性が考えられる。さらに、細分化したステージの間隔をクロック周期と定義し直せば、動作周波数は、細分化する前のステージ数を Before、細分化した後のステージ数を After とすると、(After $\div$ Before) 倍向上する。これらのことを概念的に図 5.3 で示す。図 5.3 はパイプライン段数が 3 段及び 9 段の場合における図であ

り、いずれもマルチスレッド処理を前提としたときの図である。この図は実行可能なコンテキスト数を増加させたときのプログラムの実行に要するクロック・サイクル数の変化を示す。なお、パイプライン段数が 3 段の場合は実測値に基づくが、9 段の場合は実験結果の図 4.7、4.8 に基づく推測値である。

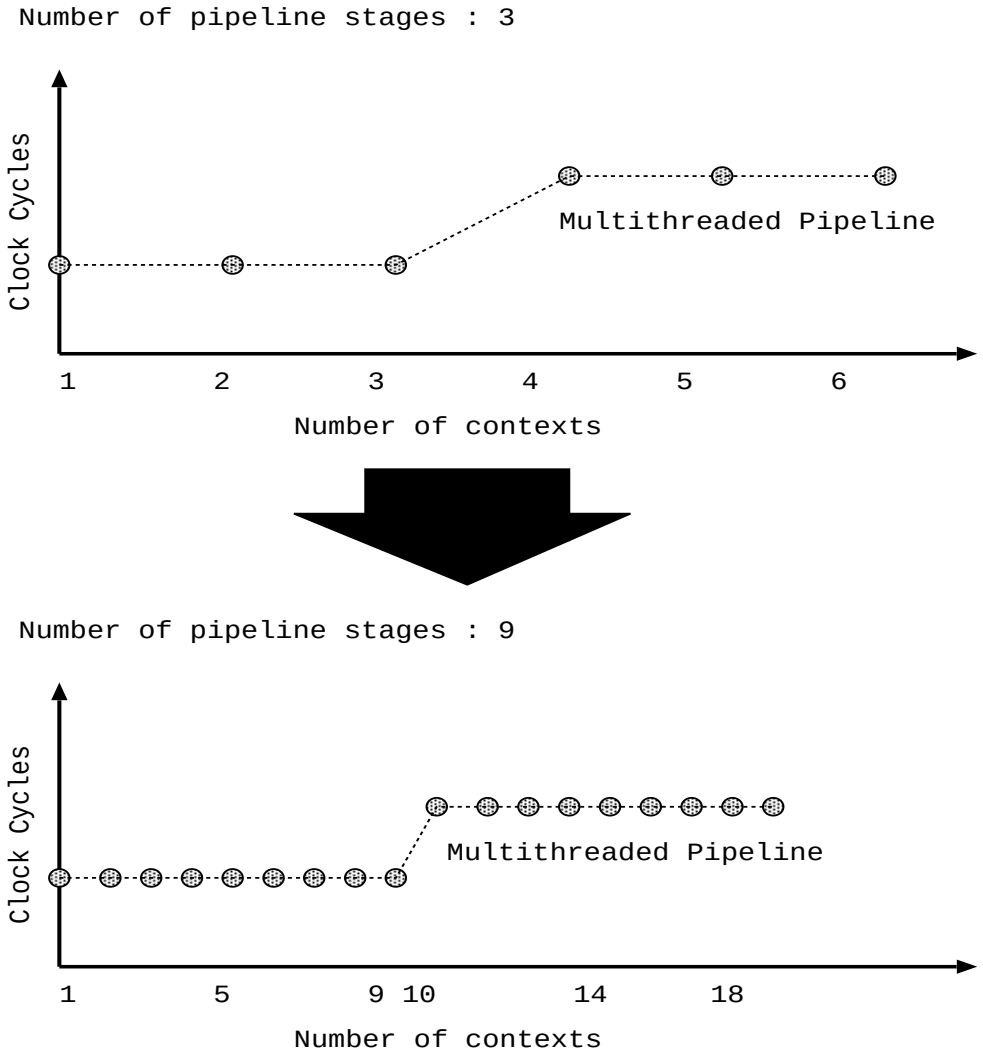

図 5.3: パイプラインの細分化

続いて、同期処理を伴う場合の通常のパイプラインプロセッサとマルチスレッドパイプラインプロセッサを比較した場合を図 4.10、4.11 に示す。図 4.10 はクロック・サイクル数を比較した結果であり、図 4.11 は CPI 値を比較した結果である。

この結果から同期処理による実行コンテキスト切替えが生じたとき、通常のパイプラインプロセッサではコンテキスト切替えに伴う処理を行なうことにより主要な処理がその分だけ遅れることになる。一方、マルチスレッドプロセッサでは、同期処理による実行コンテキスト切替えが生じた場合でも、サイクル毎にスレッドを切替えているため、主要な処理と同期処理を時分割多重で並列処理している。従って、同期処理に伴う実行コンテキスト切替えのオーバヘッドは隠蔽化されている。性能比で比較するとマルチスレッドパイプラインプロセッサは通常のパイプラインプロセッサに対して CPI 値は約 1.5 倍性能が向上し、プログラムの実行に要したクロック数は約 1.5 倍性能が向上している。

よって同期処理においてはクロック数が約 1.5 倍改善されるので総合性能は約 1.5 倍向上される。この結果からもマルチスレッド処理機構は同期処理にも有効であることが確認できる。

5.2 並列論理型言語以外で並列性を有するコードを実行させた場合の測定

並列論理型言語を効率良く実行させるには、ユニフィケーションに伴う動的なデータ型の判定を出来る限り早く確定させ、同期処理に伴う頻繁な実行コンテキスト切替えのオーバヘッドを隠蔽する必要がある。JTD 命令及びマルチスレッドパイプラインプロセッサは並列論理型言語の効率良い実行のために有効に働いた。この 2 つの効果は「動的型判定を必要とする言語」かつ「並列性を内在する言語」において有効であると考えられる。一方、他言語で並列性を内在する言語や処理に対して有効な処理はハードウェアの見地からは並列動作による効果であると考えられる。従って、並列性を内在する言語や処理に対してはその並列性を生かした動作を行なうことが可能な機構で、その処理を実行させることによって性能改善を把握できれば、その並列性を生かした動作を行なうことが可能な機構が並列性を内在する言語や処理に対して有効であることが確認できる。

先に示したように行列積で並列処理を行なった場合の特性を図 4.15、4.16 に示す。いずれの図からも明らかなようにマルチスレッド処理になると性能が大幅に向上していることが確認できる。

CPI 値で比較した性能は約 1.7 倍向上し、プログラムの実行に要したクロック・サイクル数は約 1.7 倍向上している。従って、クロック数が約 1.7 倍改善されるので約 1.7 倍総合性能が向上される。よって、この結果からはマルチスレッド処理機構は並列処理において有効であることが確認された。

以上の議論から考えられることを表 20 にまとめた。

表 20：実験結果より考えられること

問題点	解決策
unification 時のデータ型判定に伴う and,cmp,jmp 命令多数発生	JTD 命令の導入
JTD 命令の分岐ハザード	マルチスレッド処理機構で隠蔽
並列論理型言語の実行で並列にできるにもかかわらず逐次実行することによる性能低下	マルチスレッド処理機構で並列実行
同期するためのコンテキストスイッチによる性能低下	マルチスレッド処理機構で隠蔽

表 20 から言えることは、並列論理型言語特有の問題はマルチスレッド処理機構で解決されるということである。マルチスレッド処理機構を有するパイプラインプロセッサ、つまり、マルチスレッドパイプラインプロセッサはパイプラインハザードの問題を解消する。従って、並列論理型言語特有の問題は、結局、ハザードの問題を解消することと考えられる。パイプラインハザードの問題はあらゆるパラダイムに共通の問題である。従って、マルチスレッドパイプラインプロセッサはあらゆるパラダイムに有効であると考えられる。ただし、マルチスレッドパイプラインでは上でみてきたように 2 つ以上の並列性を取り出せることが性能向上の前提条件である。従って並列実行可能性を有するあらゆるパラダイムにマルチスレッドパイプラインプロセッサは有効であると考えられる。

また、本研究で提案したアーキテクチャは動的型判定に有効な JTD 命令をパイプラインハザードなしで効果的に活用できる。このことから、提案したアーキテクチャは並列論理型言語のみならず、「並列実行可能性を有する言語」でかつ、「動的型判定を伴う言語」に対しても有効であると考えられる。

第6章　結言

以上の議論から、2つのことを結論とする。第一に、JTD 命令を有するマルチスレッドパイプラインプロセッサは並列論理型言語の実行性能を大幅に向上させることが可能である。第二に、このプロセッサは並列性を内在する各々の言語に対して大いに有益な共通の並列実行機構を供給する。また、本研究の最終の目標は並列性を内在するあらゆる言語に共通に適用できる並列実行機構の提供である。その観点から今後の課題としては以下のことが考えられる。

1. 並列論理型言語は細粒度の並列処理である。これに対し、粗粒度の並列処理、例えば、JAVA のマルチスレッドプログラミングにおいても本研究で提案したアーキテクチャの有効性を確認する必要がある。
2. 並列論理型言語と同様に、「並列実行可能性」及び「動的型判定」を伴う言語は他に、例えば、Multilisp [34] が考えられる。この言語においても本研究で提案したアーキテクチャの有効性を確認する必要がある。
3. 今回のシミュレーションではキャッシュ・ミス、メモリアクセス・ミスは考慮に入れなかった。従って、それらを考慮に入れたシミュレーションを行なうことによる定量的な評価が必要である。定性的にはマルチスレッド型パイプラインプロセッサはキャッシュ・ミスを起こしたスレッド以外は処理を続行できるため、ひとつのキャッシュ・ミスによるスループット損失が、単一スレッド型と比べて 1/n(n は同時に処理されるスレッド数) で済む。
4. アーキテクチャそのものの性能向上を図る。具体的には、本研究の簡易型パイプラインからパイプライン段数を性能向上限界まで段数をあげていきその効果を追求する。
5. さらに、今回のようにハンドコンパイルではなくコンパイラの動作をソフトウェアに委ねられるようにコンパイラの開発を行ない、さまざまなテストプログラムにおいてその有効性を実証していく必要がある。

謝辞

本研究を進めるに当り御指導して下さいました日比野靖教授に心より感謝致します。また、タグ付き分岐命令を使用する際の注意点をご指摘を戴きました堀口進教授、マルチスレッドの問題点を指摘して戴きました田中清史助教授、性能比較の注意点をご指摘戴きました篠田陽一教授、宮崎純助手に深く感謝致します。

また、研究を進めるに当り、励まして下さった日比野研に心から御礼申し上げます。最後に、PARTHENON の使用を快諾して下さいました日本電信電話株式会社に感謝致します。

参考文献

[1] 日比野靖, 高級言語計算機, 電子情報通信学会誌,Vol.78,No.11,pp.1164-1170,1995.

[2] 中島浩, ICOT Technical Report:TR-670, ICOT,1991.

[3] Robert Kowalski, LOGIC FOR PROBLEM SOLVING, Elsevier North Holland, Inc., 1979. 邦訳: 論理による問題の解法, 浦昭二 監修, 山田眞一, 菊池光昭, 桑野龍夫 訳, 培風館, 1992.

[4] William F. Clocksin and Christopher S. Mellish, Programming in Prolog, Springer-Verlag, 1994.

[5] 井上光貞, 笠原一男, 児玉幸多, 改定版 新詳説 日本史, 山川出版社, 1994.

[6] 瀧和男 編集, 第五世代コンピュータの並列処理, bit 別冊, 共立出版株式会社,1993.

[7] http://www.klic.org/software/klic/index.en.html

[8] 近山隆, 藤瀬哲郎, 関田大吾, KLIC ユーザーズマニュアル, 1997.

[9] 財団法人 新世代コンピュータ技術開発機構, KLIC 講習会テキスト KL1 言語編, http://www,klic.org/software/klic/inside/master.html,1995.

[10] 松宮志麻, KLIC 処理系における並列多次元配列の実装と評価, 早稲田大学理工学部, 卒業論文,2000.

[11] Scott Oaks and Henry Wong, JAVA Threads, O'Reilly and Associates,Inc.,1997 邦訳：戸松豊和, 西村利浩, JAVA スレッドプログラミング, 株式会社オーム社,1997.

[12] 瀬尾和男, 横田隆史, Prolog 指向 RISC プロセッサ"Pegasus", 情報処理学会研究報告 (計算機アーキテクチャ),86-CA-63-5,1986.

[13] 金田悠紀夫, Prolog マシン, 森北出版株式会社,1992.

[14] 後藤滋樹, Prolog 入門, サイエンス社,1996.

[15] 田中英彦, 論理型言語指向の推論マシンの位置付けと開発の現状, 情報処理, 特集,1991.

[16] 横田実, 論理型言語の逐次実行処理方式, 情報処理, 特集,1991.

[17] 金田悠紀夫, 松田秀雄, 逐次型推論マシンのアーキテクチャ, 情報処理, 特集,1991.

[18] 市吉伸行, 論理型言語の並列処理方式, 情報処理, 特集,1991.

[19] 後藤厚宏, 並列型推論マシンのアーキテクチャ, 情報処理, 特集,1991.

[20] 柴山潔, 並列記号処理, コロナ社,1991.

[21] 淵一博 監修, 古川康一, 溝口文雄 共編, 並列論理型言語 GHC とその応用, 共立出版株式会社,1987.

[22] 伊藤英治, 関数型プログラムの実行に適したマルチスレッド型プロセッサ・アーキテクチャに関する研究 北陸先端科学技術大学院大学修士論文,1997.

[23] 中島史敬, 論理合成システム PARTHENON によるマルチスレッド並列処理プロセッサの設計, 名古屋工業大学卒業論文,1995.

[24] 小栗清, 名古屋彰, 野村亮, 雪下充輝, はじめての PARTHENON, CQ 出版社,1994.

[25] 相原孝一, マルチスレッド型プロセッサ向きのキャッシュ機構のパイプライン化に関する研究, 北陸先端科学技術大学院大学修士論文,1997.

[26] 永田真也, 可変機構を備えたマルチスレッド型プロセッサに関する研究, 北陸先端科学技術大学院大学修士論文,2000.

[27] 井上陽介, マルチスレッド型プロセッサに適したメモリアーキテクチャに関する研究, 北陸先端科学技術大学院大学修士論文,2000.

[28] 杉本朗子, マルチスレッド型ウルトラパイプライン・プロセッサの FPGA を用いた実装による検証, 北陸先端科学技術大学院大学修士論文,1999.

[29] 池田吉郎, ウェーブパイプラインを用いたマルチスレッド型プロセッサアーキテクチャに関する研究, 北陸先端科学技術大学院大学修士論文,1999.

[30] R. H. Halstead and T. Fujita, MASA:A Multithreaded Processor Architecture for Parallel Symbolic Computing, in Proc. 15th. Annual Intl. Symp. on Comp. Arch., 1988,p443.

[31] R. Guu Prasadh and Chuan-lin Wu, A Benchmark Evaluation of a Multi-threaded RISC Processor Architecture, Proc. of the 20th Int'l Conf. on Parallel Processing,Vol.1,pp.84-91,1991.

[32] K.Hwang, ADVANCED COMPUTER ARCHITECTURE, McGraw-Hill,Inc.,1993.

[33] Andrew S. Tanenbaum, STRUCTURED COMPUTER ORGANIZATION, Prentice Hall,Inc.,1999. 邦訳：長尾高弘, 構造化コンピュータ構成, 株式会社ロングテール,2000.

[34] P.M.Kogge, THE ARCHITECTURE OF SYMBOLIC COMPUTERS, McGraw-Hill,Inc.,1991.

[35] 富田眞治, 第 2 版コンピュータアーキテクチャ, 丸善株式会社,2000.

[36] Hassan, Warren's Abstract Machine:A TUTORIAL RECONSTRUCTION, MIT PRESS,1999.

[37] John P.Hayes, Computer Architecture and Organization, McGraw-Hill Companies,Inc.,1998.

[38] David A. Patterson and John L. Hennessy, Computer architecture:A quantitative approach, Morgan Kaufmann Publishers,Inc.,1990. 邦訳：富田眞治, 村上和彰, 新実治男, コンピュータ・アーキテクチャ：設計・実現・評価の定量的アプローチ, 日経 BP 社,1999.

[39] John L. Hennessy and David A. Patterson, COMPUTER ORGANIZATION & DESIGN:THE HARDWARE/SOFTWARE INTERFACE Second Edition, Morgan Kaufmann Publishers,Inc.,1998. 邦訳：成田光彰, コンピュータの構成と設計 第 2 版 [上][下]:ハードウェアとソフトウェアのインターフェース, 日経 BP 社,1999.

[40] インテル・アーキテクチャ・ソフトウェア・ディベロッパーズ・マニュアル 上巻：基本アーキテクチャ, 資料番号 243190J,Intel Corp.,1999.

[41] インテル・アーキテクチャ・ソフトウェア・ディベロッパーズ・マニュアル 中巻：命令セット・リファレンス, 資料番号 243191J,Intel Corp.,1999.

[42] インテル・アーキテクチャ・ソフトウェア・ディベロッパーズ・マニュアル 下巻：システム・プログラミング・ガイド, 資料番号 243192J,Intel Corp.,1999.

[43] 神保進一, 最新マイクロプロセサテクノロジ, 日経 BP 社,2000.

[44] Ben Catanzaro, Multiprocessor System Architecture, Prentice Hall,Inc.,1995. 邦訳：榊正憲, SPARC マルチプロセッサアーキテクチャ, 株式会社アスキー,1995.

[45] Barry Wilkinson, Computer Architecture:Design and Performance Second Edition, Prentice Hall Europe,1996. 邦訳：高橋義造監訳, 渡辺尚, 小林真也, 長谷川誠訳, 計算機設計技法 第 2 版マルチプロセッサシステム論, 株式会社トッパン,1998.

[46] 小泉修, 図解でわかる PC アーキテクチャのすべて, 株式会社日本実業出版社,2000.

[47] 藤原秀雄, コンピュータの設計とテスト, 工学図書株式会社,1990.

[48] Gerry Kane, mips RISC Architecture, Prentice-Hall,Inc.,1988. 邦訳：前川守, mips RISC アーキテクチャ R2000/R3000, 共立出版株式会社,1992.

[49] 奥川峻史, コンピュータ・アーキテクチャと RISC, 共立出版株式会社,1992.

[50] 中田育男, コンパイラの構成と最適化, 朝倉書店,1999.

[51] 小国力編著, 行列計算ソフトウェア WS, スーパーコン, 並列計算, 丸善株式会社,1997.