計算機アーキテクチャ 116－5
（1996．1．26）

オンチップマルチプロセッシングアーキテクチャ
— ロック機構 —

岩田健一　佐野健　最所圭三　福田晃
奈良先端科学技術大学院大学
{keniti-i, takesi-s, sai, fukuda}@is.aist-nara.ac.jp

集積回路技術の発展を背景に，チップの集積度は年々増大している。この傾向は今後も続くと思われる。我々は，このような集積度の増加をにらみ，1チップ上に複数プロセッサ(PE)を集積してプロセッサ間の通信遅延を削減することにより，細/中粒度並列処理を実現する「オンチップマルチプロセッシングアーキテクチャ(OCMP)」を提案している。OCMPでは並列性の記述にfork-joinを用いていたが，アプリケーションをシミュレータ上で実行した結果を解析することにより，負荷のバランスがとれない場合があることがわかった。また，その原因の一つが，同期に用いているロック機構であることがわかったため，我々はロック機構の改善を試みた。その結果，ベンチマークの実行速度が最大24%向上した。

On-Chip-Multiprocessing Architecture
— Locking Mechanism —

Ken'ichi IWATA , Takeshi SANO , Keizo SAISHO , and Akira FUKUDA
Nara Institute of Science and Technology

By growing of integrated circuit's technology, the number of transistors in one IC chip is increasing year by year. The growth enables on chip multiple processor which includes multiple processor elements in one IC chip. We have proposed "On-Chip-Multiprocessing Architecture (OCMP)" which provides effective execution environment for fine/medium grain parallel execution. OCMP provides the instruction level fork-join mechanism in order to decrease overhead for task management. Through the simulation experiments, we find the weak point of OCMP's fork-join mechanism. There are imbalance of load between processor elements. By analyzing the result of simulation experiments, the locking mechanism for synchronization of tasks is one of cause of imbalance. Thus, we try to improve the locking mechanism. By the trial, benchmark's execution time is 24% shorter than before in the best case.

1 はじめに

集積回路の集積度は年々増大している。この傾向は将来的にも加速度的に続くと予想されている。この集積度の増大を背景にして，1つのチップ上に複数のプロセッサ (PE) を集積するアーキテクチャがいくつか提案されている [1][2][3][4]。

これまでの並列処理システムは，スーパースカラや VLIW などの命令レベルの粒度で並列処理を行なうものと，より大きい粒度の処理を目的として複数の独立したプロセッサを用いるマルチプロセッサシステムが主なものである。

1チップ上に複数の PE を収納することにより，低遅延で高速なプロセッサ間通信を実現できるため，マルチプロセッサシステムよりはるかに小さい粒度の並列処理を効率良く実行することが可能となる。もちろん，1チップ上に収納できるプロセッサ数が限られるため，中規模程度の並列処理に限定される。また，1つのチップに収められるため，クロックレベルでの PE 間の同期も可能となり，命令レベルでの並列処理も可能となる。

我々は，1つのチップ上に複数のプロセッサ (PE) を集積するアーキテクチャである「オンチップマルチプロセッシングアーキテクチャ(OCMP)」を提案している [5]。我々のアーキテクチャは，簡便で高速なプロセッサ割り当てを命令レベルで提供することにより，チップ上の低遅延通信を十分に発揮することができる。この結果，OCMP では細/中粒度の並列処理を効率良く行なうことができる。

OCMP では，プロセッサ割当て命令として fork (exec) 命令，同期命令として join, unlock 命令を用意している。fork 命令は他のプロセッサに処理を依頼するが，空きプロセッサがない場合には自分で処理をする。OCMP の設計当初は，タスクが割り当てられたプロセッサには処理が依頼されないことと，処理の粒度が小さいことから自動的にプロセッサ間の負荷バランスが取れるものと考えていた。

ところが，アプリケーションをシミュレータ上で実行した結果，アプリケーションによってはプロセッサ間の負荷バランスがとれない場合があることを発見した。シミュレーション結果を解析することにより，その原因がプロセッサ間の同期待ちにスピンロックを使用しているためであることがわかった。この問題点を解決するため，ロック機構にサスペンドロック機構を追加することにした。本稿では OCMP について概要の説明および問題点の指摘をし，サスペンドロック機構を付加した OCMP の性能を評価する。

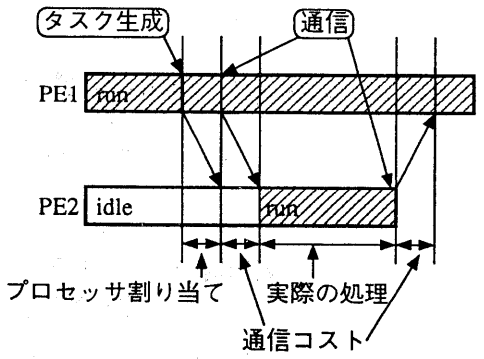

図 1: 処理のオーバーヘッド

2 OCMP

2.1 OCMP の設計方針

OCMP の目標は細/中粒度並列処理である。粒度が小さい場合，問題となるのは処理のオーバーヘッドである (図 1)。我々はこのオーバーヘッドを軽減するため，タスクを PE に割り当てる処理を機械命令で提供することにした。従来，PE へのタスクの割当て処理は，スケジューラを用いて行ってきた。スケジューラはプロセッサの利用効率を上げる一方で，それ自身のオーバーヘッドが大きいという問題がある。我々は，タスクが細/中粒度であることに着目し，スケジューラによるスケージューリングという重い処理を行なわなくてもそれほど待ちは生じないのではないかと考え，発生した仕事を単純に PE に割り当てるという操作に置き換えることによってスケジューリングのオーバーヘッドを削減することを目標とした。

2.2 構成

OCMP は，1チップ上に複数の PE と，共有キャッシュ(Shared Cache) を持つ (図 2)。各 PE はメモリ空間を共有する。また，各 PE は独立したデータキャッシュと，命令キャッシュを内蔵する。

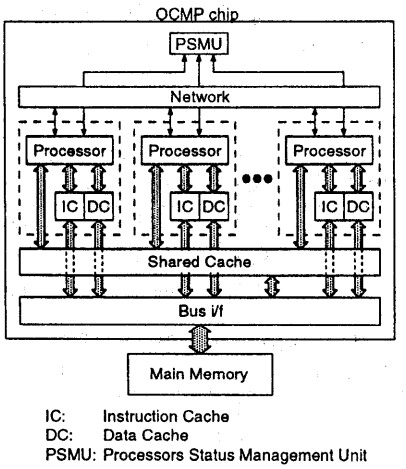

図 2: OCMP のブロック図

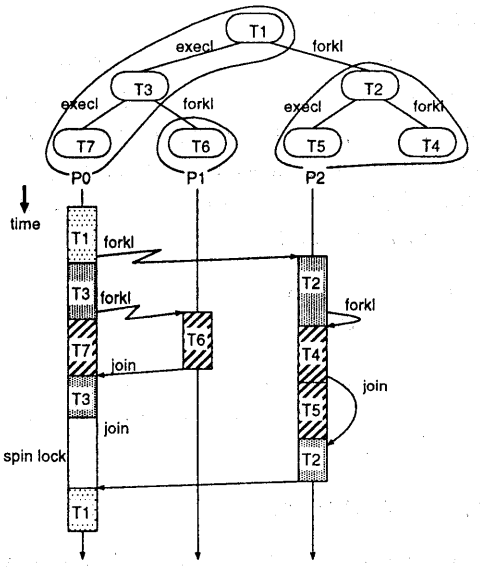

図 3: fork-join 処理

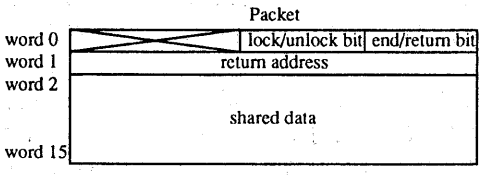

図 4: パケット

OCMP では図3に示すような fork-join によって並列性を記述し，これらの命令はハードウェアでサポートすることにより，並列処理の際のオーバーヘッドを軽減している。また，同期命令として unlock, join 命令を持つ。これらの命令の働きは次節で説明する。

2.3 fork-join 処理

2.3.1 タスク割当て

OCMP では，並列性の記述は fork-join を命令レベルで提供している。fork(exec) 命令は，新たな処理の流れを生成するもので，以下の形式になっている。

fork(exec) ra rb
(ra,rb はともにレジスタで，ra はパケットの先頭アドレス，rb は処理の飛び先番地を指している)

呼出側 (親) は fork 命令に先立ち，共有キャッシュ上にパケット (図 4) と呼ばれる領域を確保する。ここには，被呼出側 (子) が必要とするデータ (引数) および，戻り先番地，lock/unlock bit，return/end bit などを設定する。lock/unlock bit は，親と子の同期処理 (join 命令), return/end bit は子の終了処理 (unlock 命令) で用いる。その後，以下のように fork 命令を実行する。

1. 空き PE がある場合

   パケットの return/end bit を end に設定して確保した PE に子のタスクを割り当てる。

2. 空き PE がない場合

   親への戻り番地をセットし，return/end bit を return にセットして自分に子のタスクを割り当てる。

exec 命令は，必ず自分自身にタスクが割り当てられることを除けば，fork 命令と同様である。

OCMP では，関数を並列処理の単位として，あるタスクの中で 2 つ以上のタスクを生成し，それらを fork 命令と exec 命令により，空いているプロセッサおよび自分自身に割り当てることにより並列処理を行う。

2.3.2 子の終了処理

子は，割り当てられたタスクの先頭番地から処理を進めるが，そのタスクは必ず unlock という命令で終了する。unlock 命令は，呼び出された時のパケットの lock/unlock bit を unlock にしたあと，end/return bit の状態によって PE を解放するか，または親の処理に戻るかの処理を行う。

1. end/return bit が end のとき
   PE を解放する。
2. end/return bit が return のとき
   パケットの return address に戻る (親の処理に戻る)。

2.3.3 親と子の同期処理

親は子にタスクを割り当てた後，何らかの処理を実行したのちに子の終了を待ってデータを受けとらなくてはならない。子からのデータは子のパケットを介して受け渡されるが，このデータの読みだしには join 命令を用いる。join 命令は，パケットのアドレスと，読みだすべきデータのオフセットを引数としてとる。パケットの lock bit をチェックして lock ならばビジーウェイトによるスピンロックに入り lock が解除されるまで待つ。lock されていないならばデータを読みだして，指定したレジスタに持ってくる。

3 従来の OCMP の評価および問題点

我々は，従来の OCMP の性能を評価するため，シミュレーターを作成し，いくつかのアプリケーションを実行した。その結果，スピンロックによる待ちが大きいという問題点があることがわかった。

3.1 結果

ベンチマークにはクイックソート (qsort) を用いた。OCMP の PE 数が 4，8，16 のそれぞれの場合について，100，1,000，10,000 要素のデータをクイックソートでソーティングした。表 1–3 にその結果を示す。なお，run は何らかのタスクを実行している状態，idle はタスクが割り当てられておらず PE がアイドルの状態，spinlock は親が子をスピンロックにより同期待ちしている状態である。

表 1: qsort100 の PE の状態の割合

PE 数	run(%)	idle(%)	spinlock(%)
4	49.6	26.7	23.7
8	32.7	47.9	19.5
16	19.2	69.5	11.3

表 2: qsort1000 の PE の状態の割合

PE 数	run(%)	idle(%)	spinlock(%)
4	53.5	17.7	28.7
8	38.9	35.4	25.7
16	26.0	55.2	18.8

表 3: qsort10000 の PE の状態の割合

PE 数	run(%)	idle(%)	spinlock(%)
4	55.4	15.5	29.1
8	40.7	31.6	27.8
16	28.0	50.2	21.8

これらの表から，スピンロックが無視できないぐらい大きいことがわかる。

3.2 fork-join 処理における問題点

OCMP での fork-join 機構では，処理を二つに分けて自分自身と子に割り当てるが，子の処理が親の処理より短くなるように割り当てないと，親が子より先に終了する。この結果，親には子との同期待ちが生じる。同期待ちの期間は，親はビジーウェイトによるスピンロック状態になり，無駄にプロセッサを占有してしまう。

このような場合でも我々は，タスクの粒度が十分小さいことから，同期待ちはそれほど大きくならないと考えてきた。しかし，上記のシミュレーションの結果，同期待ちが予想以上に大きいことがわかった。

qsort ではタスクの大きさはリストに比例しているため，ソフトウェアで対応することも可能だが，一般的にはタスクの大きさを把握することは困難である。

次節では，この問題点を改善する機構を提案し，改善後の性能を評価する。

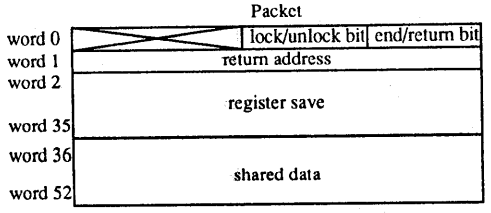

図 5: 改良後のパケット

4 サスペンドロック機構

3節で，ビジーウェイトによるスピンロックはタスクが細/中粒度である場合でも多くの時間 PE を占有してしまうことがわかった。この問題を解決するため，スピンロックの代わりにサスペンドロック機構を用いることを提案する。

4.1 概要

これまで，親のタスクが子のタスクより早く終った場合，親はビジーウェイトによるスピンロックにはいり，この終了を待つようになっていた。このスピンロックの代わりに，サスペンドロックを用いる。親は，子がまだ終了していないことを発見すると，プロセスを退避し，PE を解放する。子は，自分の終了時に親がすでに終了いる場合は，親のプロセスを復帰し，親の処理を続ける。この方法は，プロセス退避などによるオーバーヘッドが大きくなるが，スピンロックによる無駄な PE の占有がなくなる。

この機構を実現するため，パケットに，親の終了/未終了ビット (suspend/end bit) と親のプロセスの退避スペースを加える (図 5)。

4.2 タスク割当て処理

タスク割当ては従来と同様である。

4.3 子の終了処理

子は従来と同様に unlock 命令で終了するが，lock bit を unlock した後，親が終了しているかどうかチェックし，終了していれば親のプロセスを復帰して親のタスクを再開する。まだ終了していなければ，従来と同様に lock bit をリセットして終了する。

4.4 親と子の同期処理

従来と同様に join 命令で同期をとって返り値を読み出すが，その際に子がまだ終了していなければプロセスを退避し，子のパケットの 親の終了ビットをセットして終了する。子がすでに終了していれば，従来と同様に処理する。

4.5 評価

我々は，OCMP におけるサスペンドロックの効果を評価するために，3節のシミュレータにサスペンドロック機構を加え，同様のベンチマークを行った。

サスペンドロックにおいてはプロセスの退避と復帰作業が必要となるため，スピンロックの期間が短い場合，サスペンドロックに置き換えても改善されない。場合によっては性能が落ちることもある。表4は，3.2節で示したベンチマークにおけるスピンロックの長さの分布である。

表 4: qsort1000 の PE 数 8 の場合

spinlock サイクル数	個数 (%)	サイクル数合計 (%)
1 -10	5.9	0.03
11 -100	28.2	1.4
101 -1000	49.4	14.6
1001-10000	14.0	35.6
10001-	2.5	48.3

この表からわかるように，サイクル数が長いものの個数は少ないが，それが全スピンロックサイクルに占める割合は大きい。また，1回のスピンロックにおける平均サイクル数は 1142 サイクルとなっており，スピンロックをサスペンドロックに変更しても十分効果があることを示している。

ここでは，シミュレータの都合上プロセス退避のオーバーヘッドはないものとしてシミュレーションを行なった。実際にもレジスタ退避/復帰のオーバーヘッドが主なもので，数~10 数サイクルであると考えており，これは平均スピンロックサイクル数より十分小さい値であるので，オーバーヘッドはないという仮定の元であっても現実とかけはなれた結果は出ないものと考えている。

実際にクイックソートを実行した結果を表 5-7 に示す。

表 5: qsort100 の PE の状態の割合

PE 数	run(%)	idle(%)	spinlock(%)
4	59.1	40.9	0.0
8	36.9	63.1	0.0
16	19.5	80.5	0.0

表 6: qsort1000 の PE の状態の割合

PE 数	run(%)	idle(%)	spinlock(%)
4	69.9	30.1	0.0
8	48.9	51.1	0.0
16	29.5	70.5	0.0

表 7: qsort10000 の PE の状態の割合

PE 数	run(%)	idle(%)	spinlock(%)
4	73.2	26.8	0.0
8	53.4	46.6	0.0
16	33.4	66.6	0.0

また，3.1節の結果と比較して，のべ実行サイクル数および各状態の増減を表 8–10 に示す。run サイクル数の変化はなく，スピンロックのサイクル数はほぼ 0 になっているので割愛した。

表 8: サイクル数の増減 (qsort100)

PE 数	のべ clock(%)	idle(%)
4	-16.0	35.3
8	-11.2	18.5
16	-1.2	14.8

表 9: サイクル数の増減 (qsort1000)

PE 数	のべ clock(%)	idle(%)
4	-23.3	36.4
8	-20.1	17.9
16	-11.7	13.6

表 10: サイクル数の増減 (qsort10000)

PE 数	のべ clock(%)	idle(%)
4	-24.4	35.8
8	-23.4	16.1
16	-14.7	13.2

要素数が多い場合には最大 24%の実行速度の改善が見られた。これは，スピンロックによって無駄に占有されていた PE が活用されたことを示している。しかし，スピンロックの割合が少ない場合の実行速度の改善はほとんど見られなかった (表 8 PE=16 のとき)。これは，アイドルプロセッサ数が十分であるため，スピンロックしている PE を解放しても割り当てられるタスクが少ないためである。しかしながら，スピンロックで占有されていたサイクルがアイドル状態になるため，アイドルの割合はどの場合にも増加する。これはマルチプログラミング環境において処理の多重度を大きくできることを示している。

5 まとめ

本稿では OCMP の動作を説明し，タスクの偏りによるスピンロックサイクル数の増大という問題点を指摘した。また，それに対する改善策として，サスペンドロックを導入し，シミュレータでその有効性を確認した。具体的には，スピンロックによる PE の占有がほぼなくなり，その PE が実行待ちのタスクを実行することによって，実行時間が最大 24%改善された。また，実行待ちのタスクがない場合でも，それがアイドル状態になる。そのため，アイドルのサイクル数は最大 36%増加した。

今後の課題として，パイプライン実行やキャッシュ，サスペンドロック時のオーバーヘッドを含めたより詳細なシミュレーションを行なう予定である。

参考文献

[1] P.P.Gelsinger et al.,"Microproessor circa 2000", IEEE Spectrum,Vol.26,No.10,pp.43-47,1989.

[2] G.S.Sohi et al."Multiscalar Processors",Proc. of the 22nd Annual International Symposium on Computer Architechture,pp.414-425,1995.

[3] 高橋真史, 高野裕之, 鈴木清吾, 田胡治之 "オンチップマルチプロセッサのアーキテクチャの検討", 信学技報, CPSY95-3,1995.

[4] 岩下茂信, 宮嶋浩志, 村上和彰 "次々世代汎用マイクロプロセッサ・アーキテクチャ PPRAM の概要" 情処研報, 95-ARC-113,pp.1-8,1995.

[5] 田沼仁, 田中哲也, 岩田健一, 福田晃 "オンチップ・マルチプロセッシング・アーキテクチャ(OCMP)の提案" 信学技報, CPSY,May 1995.