マルチコア・マルチスレッド CPU における浮動小数点演算命令のコア間委託実行機構の提案

高木, 一弘 / TAKAKI, Kazuhiro

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編 / 法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
11
(開始ページ / Start Page)
1
(終了ページ / End Page)
6
(発行年 / Year)
2016-03-24
(URL)
https://doi.org/10.15002/00012881

マルチコア・マルチスレッドCPUにおける浮動小数点演算命令のコア間委託実行機構の提案
An Architecture towards Sharing FPU across Cores for the Design of Multithreading Multicore CPUs

高木 一弘 *
Kazuhiro Takaki
法政大学情報科学研究科情報科学専攻
Email: kazuhiro.takaki.r2@stu.hosei.ac.jp

***Abstract*—The multithreading and multicore techniques are widely adopted in the design of the modern high-performance CPUs. Multithreading technique allows multiple threads to share the functional units (FUs) within a core for the better utilization of the FUs. Thus there will be confliction on the use of some FUs, the floating-point unit (FPU) for instance. In such a case, some floating-point instructions will be suspended until the FPU is available for use. Multicore technique implements a small-scale multiprocessor on a chip. A thread that runs on one core cannot use the FUs of other cores. This results in poor utilization of the FPU in some cores if the threads running on those cores do not contain floating-point instructions at all, although in other cores, the threads are straggling to complete for the FPU. Different from the traditional multiprocessors that are implemented with multiple CPU chips, because the multicore CPUs implement multiprocessors on the same chip, it becomes possible to let the threads in a core group share all the FPUs in the group. When a conflict on the use of FPU occurs, some floating-point operations can be redirected to the cores of the same group in which the FPUs are in idle state, so that the overall performance of the multicore CPU will be improved. This paper investigates such a group architecture and gives the performance improvement of the proposed architecture to that of the traditional multicore architecture. Our experimental results show that, on average for the floating-point benchmarks, 4.25%, 7.34%, and 7.45% performance improvements can be achieved by redirecting the floating-point operations to other cores within the group with the group sizes of two, four, and eight, respectively, under the conditions of instruction redirecting overhead is assumed to be zero.**

I. はじめに

過去数十年間にわたってムーアの法則にしたがったCPUの性能向上はチップ上のトランジスタ数の向上, クロック周波数の向上, また命令レベル並列性の追求などにより実現されて来た. 一方でさらなるクロック周波数向上の追求には過大な電力が必要となり, 一般的なPCやサーバーに搭載する限界を突破してしまうという点, 現実的な命令レベル並列性の追求には限界が見えてきたという点などを背景として命令レベル並列性と比べより粒度の粗い並列性を利用して性能向上を図る手段としてマルチスレッド技術やマルチコア技術が注目されてきた. マルチスレッド技術においてはクロックごとにスレッドを替えるFine-Grainマルチスレッディングやレイテンシの長い命令を発行した場合にスレッドを替えるCorse-Grainマルチスレッディング, 複数のスレッドを同時に実行する同時マルチスレッディング [1] などが提案されてきた. 一方でマルチコア構成においてはシンプルなコアを多数搭載するもの [2] や, 高性能なコアを少数搭載するもの [3] などが提案されてきた. 近年においてはマルチコア, マルチスレッディング構成をとるCPUが搭載されたデバイスが非常に多くなっており, 一般的なコンピュータに加え, ノートパソコンやタブレット, スマートフォンなど多岐に渡るまでとなっている.

マルチコア構成をとるCPUは複数のコアによって構成されており, 各コア自体がマルチスレッディングCPUと同様の挙動を行う. マルチスレッディングCPUの特徴の一つとして, コア上で動作する全てのスレッドが各種の機能ユニット (FU), すなわちALU, 分岐ユニット, ロードストアユニット, 浮動小数点演算ユニット等を共有しているという点がある. マルチスレッディング技術は複数のスレッドが発行する命令列を, これらFUの使用率を向上させるようスケジューリングして実行することで性能の向上を果たす機能を持つ. 一方で複数のスレッドが同時に同じFUを使用する命令を複数発行した際にハードウェア資源の競合が発生するため, マルチスレッディングによる性能向上にも制限が発生する. すなわち, 実装されたFUの数やパイプラインよりも同一サイクル内で発行されたそれらを利用する命令の数が多い場合, 全ての命令を一度にパイプラインに投入することは不可能となる. したがって実装されたFU, パイプラインの数を超えた後続の命令についてはパイプラインが投入可能な状態になるまで待機する必要があり, ストールが発生する.

この問題に対する最も簡単な解決方法は, 実装するFUの数を増やすことである. しかし, この方法は演算性能の向上を実現する一方で, コストの増加という問題を発生させるだけでなく, 追加で実装されたFUが常に最大限活用されるわけではないという, 使用率の低下という問題も孕んでいる. マルチコアアーキテクチャにおいては, コアに対するコストの増加はすなわち実装するコアの数だけCPUへのコスト増加を招き, コア数が多い場合には大きな問題となり得る. 特にスマートフォンやノートパソコンなどのモバイル型の端末においては, より少ないコスト・消費電力でより良いパフォーマンスを保証する必要があるためこの問題はさらに重大なものとなる. 昨今ではメニーコアアーキテクチャも積極的に研究が行われており, モバイル端末同様に一つのコアへのコスト増加が大きな問題となり得る. したがってコアに対するコストの増加は慎重に検討されるべきである.

本論文においてはこの問題に対して, 各コアに実装されたFUをコア間で相互に利用できるようにし, 命令の授受を行うことにより衝突を回避するアーキテクチャの提案を行う.

* Supervisor: Prof. Yamin Li

このような機構を CPU の構成に組み込むことで, FU の実装数を増やすことなく、各コアで使用できる FU の数を増やすことができるため、性能向上を見込むことが可能となる. 同時に, 従来複数個実装していたコストの大きい FU の数を削減し, 提案機構を組み込むことで同程度のパフォーマンスをより少ないコストで実現するという運用も考えられる. それらに加え, 使用率の低い FU に対してタスクを実行させることにより, リソースの利用率の向上にも寄与することが可能である.

II. 資源競合と命令のコア間委託実行

一般的なマルチコア, マルチコア CPU において,FU はコア内で複数のスレッドによって共有がなされており, コア間での共有はされていない. このような構成はマルチコア CPU の設計を容易にする一方で, 機能ユニットの使用率の低下を招く. 通常それぞれのコアは異なるタスクを同時に実行する. このような場合あるコアにおいて, ある FU の競合が頻繁に発生している一方で, 別のあるコアにおいては当該 FU の使用率が低いという状況が存在する.

従来はそれぞれのコア内におけるこのような事態について着目し, マルチスレッディング技術によって解決が成されてきた. 一方でマルチコア環境においてもその構成を利用する事で同様の考えを適応させることができると考えられる. すなわち, 本来のマルチスレッディングによる実行では待機状態に陥る命令を他コアの計算資源を利用して計算することで, 本来当該クロックでは実行が不可能であった命令が実行可能となる. したがって, あるコアが他のコアの計算資源を一時的に借りることで, マルチコア, マルチスレッド構成の CPU の使用率を向上させることが可能となる.

図 1 はハードウェア資源の競合が発生した状況下において, コアから別のコアへ命令の委託を行った状況を示したものである. コア A では浮動小数点演算を中心としたタスクが実行されている一方で, 別のコア B では整数演算を中心としたタスクが実行されているような状況を想定する. このような場合コア B における FPU は多くの時間において有効に使用されていない状態となる. したがってコア A においてリザベーションステーションで待ち状態となっている命令をとりだし, コア B の資源を割り当てることで待ち状態を解消する.

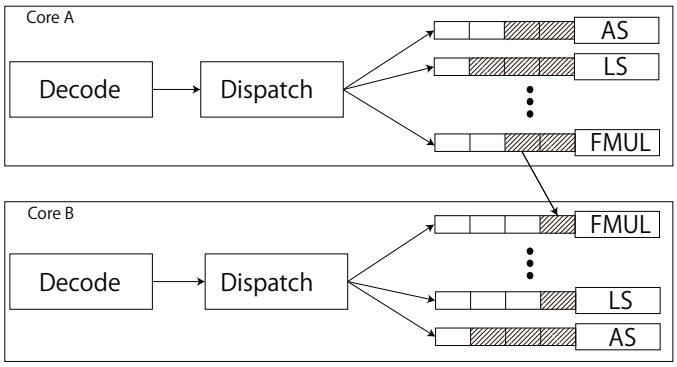

図 1. 命令の委託実行による衝突回避

III. Issue レイテンシと Operation レイテンシ

本節では衝突に関する問題について, 我々が取り扱う命令の特徴について記述する. CPU 上で各命令を実行する際に発生するレイテンシは,Issue レイテンシと Operation レイテンシによって構成される. Issue レイテンシとは後続の命令が同一の FU を実行できるようになるまでの遅延時間を表す. すなわち, この値は直前の命令がパイプラインに投入されてから, 後続の命令をパイプラインに投入できるようになるまでに必要となるクロックサイクルによって定義される. 一方で,Operation レイテンシは命令がパイプラインに投入されてから演算が終了するまでにかかる時間を表す.

本論文においては特に Issue レイテンシについて着目する. 昨今における CPU においては FU が高度にパイプライン化されており,Issue レイテンシの短い命令が多い. 特に整数演算では同一 FU を利用する後続命令を次のサイクルにはパイプラインに投入できるよう設計されている. 一方で複雑な演算を行う FU においてその実装が反復的な手法による実装がなされた場合,Issue レイテンシは大きなものとなる. このような場合においては毎クロック命令を演算パイプラインに投入するということは不可能となる.

本研究においては衝突に関する問題について, 命令をコア間で委託実行をすることにより解決を図る. すなわち, 先行命令の Issue レイテンシが完了していない後続命令について, 他のコアによる資源を利用して実行することで先行命令の Issue レイテンシによって発生した待ち時間を削減することが可能となる. 提案手法においてコア間で命令を授受しあう際にはオーバーヘッドが発生することが予想される. したがって Issue レイテンシがオーバヘッドよりも短い命令を他コアへ送信した場合, 計算時間が逆に伸びてしまう. そのため, 本研究にこにおいては整数演算及び論理演算等の命令のような Issue レイテンシの短いものについては送信を行わないものとする. それに対し Issue レイテンシの長い命令については, 後続の命令の待ち時間がオーバーヘッドよりも長い場合, その差分だけ性能向上を見込む事が可能である. これは Issue レイテンシが長い命令ほど効果的であり, またこのような命令がより長い連続で発効された場合, より性能の向上が見込まれる. そのような命令の代表として, 本研究では特に浮動小数点演算について着目し, コア間の委託実行を行うことによる性能向上について評価を行った.

IV. 提案するアーキテクチャ

本論文では次のような構成をとるマルチコア CPU を想定する: 1) スーパースカラの命令発行幅は 8 とする. 2) 各コアは 8 つのスレッドスロットを持つ. これらのコアをグループに分け, 同一グループに属するコア同士での浮動小数点演算命令の授受を可能とする. 整数演算装置 (IU) については従来の CPU の構成と同様なものとし, 各コアで占有的ものとする. 本論文における提案手法はマルチコア技術とマルチスレッディング技術を合わせたものと考えることができる. なお, グループに所属するコアの数をグループサイズと呼称することとする.

フェッチされた命令はその後, 命令間のデータ依存, 構造依存が解消されるまで待機する為のリザベーションステーションへディスパッチステージで割り当てられる. 次に命令は FU, リオーダバッファ, バイパスネットワークに対して割り当て要求を行う. 従来のアーキテクチャでは, この割り当て要求を行った際に FU が先行命令によって使用中である場合, または先行命令の Issue レイテンシが残っている場合後続命令は待機状態となる.

提案するアーキテクチャにおいては, このタイミングで各コアは自身の FU への割り当て要求を行うと同時に, 同一グループに属する他のコアの FU についても割り当て要求を

行う. 自身の FU がビジー状態であり, かつ別のコアが当該 FU についてビジー状態でなかった場合, この要求を承認し割り当てを実行する. 一方で, 他コアの機能ユニットが開放されるタイミングが, 待ち時間が解消されるよりも遅い場合には命令の委託は行わない. 同様に, 競合が解消されるまでにかかる時間がオーバーヘッドよりも短い場合についても命令の委託は行わない. これは命令の委託を行う際にはその実行時間に加えコア間の移動に必要となるオーバーヘッドが余計にかかり, 命令委託を行った方が実行速度が低下するためである. そして, 自身の機能ユニットおよび同一グループに属する全てのコアの FU において割り当て要求が通らなかった場合, この命令は待機状態となる.

コア間での命令の委託を実行するには, 先行する命令の Issue レイテンシがどれくらい残っているかを把握する必要がある. 本研究においては各 FU にカウンタを設置した. このカウンタはその FU が利用を開始されてから何クロックが経過をしたかを記録する. このカウンタは FU にアクセスがあった時にカウントを開始し, その値が Issue latency を超えたあとはその値に固定される. したがってこの値がその FU の Issue latency よりも小さい場合は, その FU はまだ後続命令を実行できる状態にないということを示す. このカウンタに保存されている値とオーバーヘッドを比較し, その値の方が小さければ命令の委託実行を行うことを決定する.

マルチコア CPU においては委託先の候補が複数存在する場合が発生する. したがって, この委託先を決定するアルゴリズムが必要となる. 本論文において採用したアルゴリズムについては第 V 節で説明を行う.

以下の図 2 はグループサイズが 2 であるマルチコア CPU におけるひとつのグループについて着目した場合のアーキテクチャの概要を示す. Optimizer が浮動小数点演算命令のコア間委託を実行する. 片方のコアで衝突が発生した場合, 命令はそのオペランド, スレッド ID, プロセス ID などの情報と共に Optimizer を通して別のコアへ送信される. 委託されたコアは演算を代理として実行し, その結果を Optimizer へ返す. Optimizer はその結果を元のコアに渡し, 委託元のコアは Optimizer によって渡された計算結果, 及びプロセス ID, スレッド ID を用いてライトバックを行う.

FPU を複数のコア間で共有するアーキテクチャについてはこれまでに提案が行われている [4][5][6]. これらにおいては, 浮動小数点演算を行う際には必ずコアから外部の FUP への命令移動が必要となり, オーバーヘッドがかかるが, 提案手法においては命令の委託を行うときのみにオーバーヘッドがかかるという違いが存在する.

V. FU の割り当てアルゴリズム

ここでは命令委託を行う際の割り当て先を決定するアルゴリズムについて説明する. 本論文では各コアで割り当て先に優先度をつけて命令の授受を行う優先度つき割り当てアルゴリズムと, 最後に使用されてから最も時間の経ったコアに対して割り当てを行う Least Recently Used(LRU) アルゴリズムについて実装を行った. これらのアルゴリズムには単に割り当て先を決定するという点に加え, あるコアが命令の委託先として集中することによる実行タスクの偏りが発生する事象を避け, 全てのコアが平等に命令の分配先となるようにするという役割がある. また, 節 VII-A で後述するが命令の委託によって委託先のコアにおいて衝突が発生し, 委託先先のコアがまた別のコアへ委託を行うという事象が発生しうる. したがってどのコアに対して割り当てを行うかを決定するアルゴリズムはこの点においても大きな役割がある.

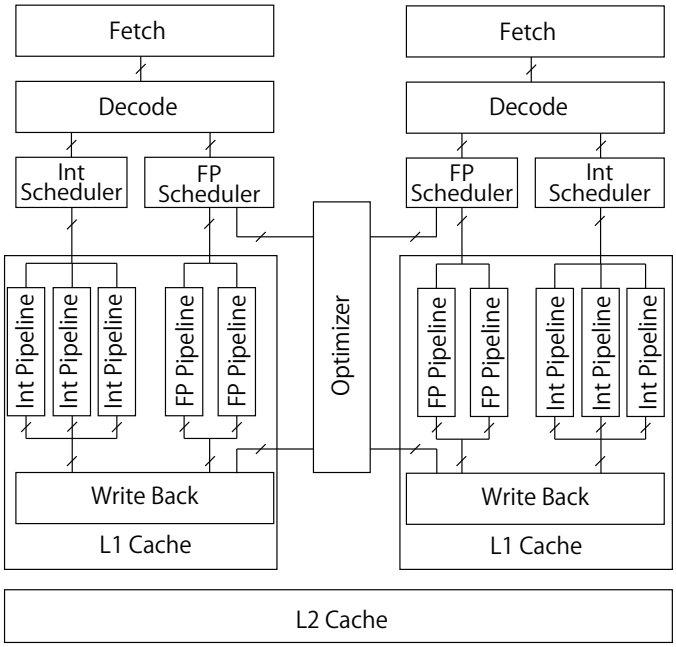

図 2. 提案アーキテクチャ概要

A. 優先度付き割り当てアルゴリズム

まず, 各コアをグループサイズで決定された数字に従ってグループ分けを行う. 次に, これらのコアに対してグループ内でユニークな数字のインデックスをゼロからグループサイズから 1 を引いたものまで順に付番する. あるコアにおいて命令の委託を行う際は, まず自分のインデックスに 1 を足したものに割り当て要求を行う. 割り当て要求を行ったコアの FU もビジー状態であった場合, ビジーでないコアが発見されるまでこのインデックスをインクリメントしていく. したがって, あるコアに対してそのインデックスが 1 大きいコアが最も高い優先度を持つこととなる. 一方で, 最も優先度が低くなるのは自身のインデックスから 1 引いたものとなる. なお, この割り当てアルゴリズムにおいてインデックスは循環リストのように扱われ, 自身のグループ内でのインデックスにインクリメントを行った結果がグループサイズを超える場合, 割り当て先のインデックスはまたゼロから順に参照されていく. 図 3 は 8 コアによって構成される CPU について, グループサイズ 2 の場合における ID が 0 と 7 のコアの FU の割り当て先の優先度を示したものである.

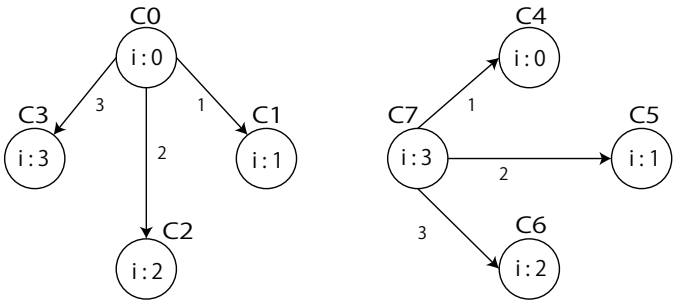

図 3. グループサイズ 2 における優先度付き FU 割り当ての例

このアルゴリズムにおける利点は, 仕組みがシンプルであるためその実装コストが比較的少なくなることである. 一方

で割り当て先のコアの命令の流れを考慮しないため, その流れを乱す可能性が高くなる.

B. LRU

LRU はキャッシュメモリや仮想メモリが扱うデータのリソースへの割り当てを決定する際に利用されるアルゴリズムであるが, 本研究における FU の割り当てアルゴリズムにも応用することが可能である. 各コアに所属する FU ごとに, 割り当て要求が行われたタイミングを記録することで, より利用率の低いコアの FU を優先的に選択して命令委託を行うことが可能となる. これにより各コアにおける FU の利用率の平滑化, 及び使用されていないリソースの有効利用が可能となる.

また,LRU を利用することで, そのコアが実行中のタスクの短期的な特徴を把握することが可能となる. すなわち,FPU へのアクセスが他のコアと比べより以前に行われていた場合, 次のクロックで FPU へアクセスする可能性は低くなる. したがって, そのようなコアに対して命令委託を行う場合, 委託先の命令列を乱すことなく割り当てを実行することが高い確立で可能となる.

一方で各 FU ごとにその使用履歴について記録を残し, またこの記録を利用して最も使用履歴の古いものを選ぶという工程を経る必要があるため, 必要となるハードウェアコストが比較的大きくなるという特徴がある.

なお, 本研究における LRU においては, 各 FU に対するアクセスがその FU が所属するコア自身の命令によるアクセスであるか, 他コアからの委託実行によるアクセスであるかを分けており, 自身の命令によるアクセスのみを LRU の計算に用いている. これは, 本来整数演算が主であるコアが命令の委託実行を請け負ったために, ごく最近 FPU が利用されたと判断され, 割り当て先としてのその優先度が下がることを避けるためである. 自身が発行した命令によるアクセスと, 委託実行によるアクセスとを判別することで, 整数演算を主に行うコアにおいて, 委託による FPU へのアクセスが直前に起こった場合についても割り当て先としてその優先度を高い状態に維持することができる.

VI. シミュレーション

ここでは提案アーキテクチャのシミュレーション方法について記載する. 本論文ではオープンソースのシミュレータである Multi2Sim[7] を利用することで CPU の詳細な挙動を取得した. Multi2Sim は C 言語で記述された CPU-GPU ヘテロジニアスアーキテクチャのシミュレーションフレームワークであり, サイクルごとの実行駆動型シミュレーションが行えるため CPU の挙動を詳細に観察することが可能である. スーパースカラーアーキテクチャのシミュレーションにおいては SimpleScalar[8] が使用されることが多い. しかし,SimpleScalar はマルチコア環境, マルチスレッド環境に対応していない. そのため提案するアーキテクチャを実装するには多くの修正が必要となる. 一方で Multi2Sim は Fine Grain Multithreading(FGMT),Coarse Grain Multithreading(CGMT),Simultaneous Multithreading(SMT) に対応している上に, マルチコア構成もパラメータによって容易に設定できるため提案アーキテクチャをシミュレーションする上で適切であると考え, 本研究で利用するシミュレーターとして採用した.

A. シミュレーション方法

性質の異なる複数のタスクが異なるコア上で同時に実行されているという一般的な実行環境を再現するため, 浮動小数点演算を多く含むベンチマークである Splash2[9] と, 整数演算を多く含むベンチマークである Mediabench[10] をタスクとして混在させて実行を行った. これらのプログラムを混在させて実行することで FPU を頻繁に利用し, 衝突が頻発するコアがある一方で, FPU を使わないコアが存在するという状況を再現することができる.

まず Splash2 におけるそれぞれのプログラムを単体で実行し, それぞれのプログラムに内包されている衝突によって発生する待機状態がどの程度であるか, またそれらの待機状態がどの程度解消可能であり, 性能向上はどの程度見込めるかという点について評価を行う. 次に Splash2 と Mediabench を混合したタスクを実行することで, より一般的な実行環境下において FPU が飽和状態に陥るコアと,FPU の使用率の低いコアが混在する状態について再現し評価を行う. これらのシミュレーションを行う際には, 実装方法による影響を避けるため, CPU は理想なものとしてそのオーバーヘッドの値をゼロとした.

これらに加え, 命令の委託実行を行う際に発生するオーバーヘッドの性能に対する影響を評価するため, 後者のタスクについてオーバーヘッドの値を 1 から 4 までとして実行を行い, 得られた IPC について比較を行った.

これらの内容について, 典型的なマルチコアアーキテクチャである, 命令の委託実行を行わないものをベースラインとし, その結果とグループサイズを 2,4,8 としたものとを比較し評価を行う.

B. シミュレーション環境

シミュレーションで設定したプロセッサの詳細な構成について説明する. 本稿においては各機能ユニットは全て並列にアクセスできるものと想定してシミュレーションを実行した. このような構成をとった理由はアーキテクチャの設計手法によるシミュレーション結果への影響を抑えるためである. 本論文においては各 FU は反復型による実装ではなく, パイプライン構成による実装とする [11].

今回のシミュレーションで想定する機能ユニットについて, 特に FPU においてその実装する数, Issue レイテンシの長さ,Operation レイテンシの長さを表 1 に示す [12]. なお,Multi2Sim では,x86 命令における fexp,flog,fsin,fcos,fsincos,ftan,fatan,fsqrt 命令は Floating-point complex ユニットにまとめて実装したものとされ, 全て同一のレイテンシとして扱われる.

表 I
機能ユニットの実装数とレイテンシ

FU name	Number	IssueLat	OpLat
FP COMPARE	1	2	3
FP ADD	1	2	5
FP MUL	1	4	8
FP DIV	1	20	40
FP COMPLEX	1	50	100

VII. シミュレーション結果

実験に使った 8 コア 8 スレッド, スーパースカラーの命令発行幅 8 の CPU について, まず splash2 における全てのプログラムを個別に実行し, その結果を比べたものについて報告を行う. 図 4 はそれぞれのプログラムにおいて, 各機能ユニッ

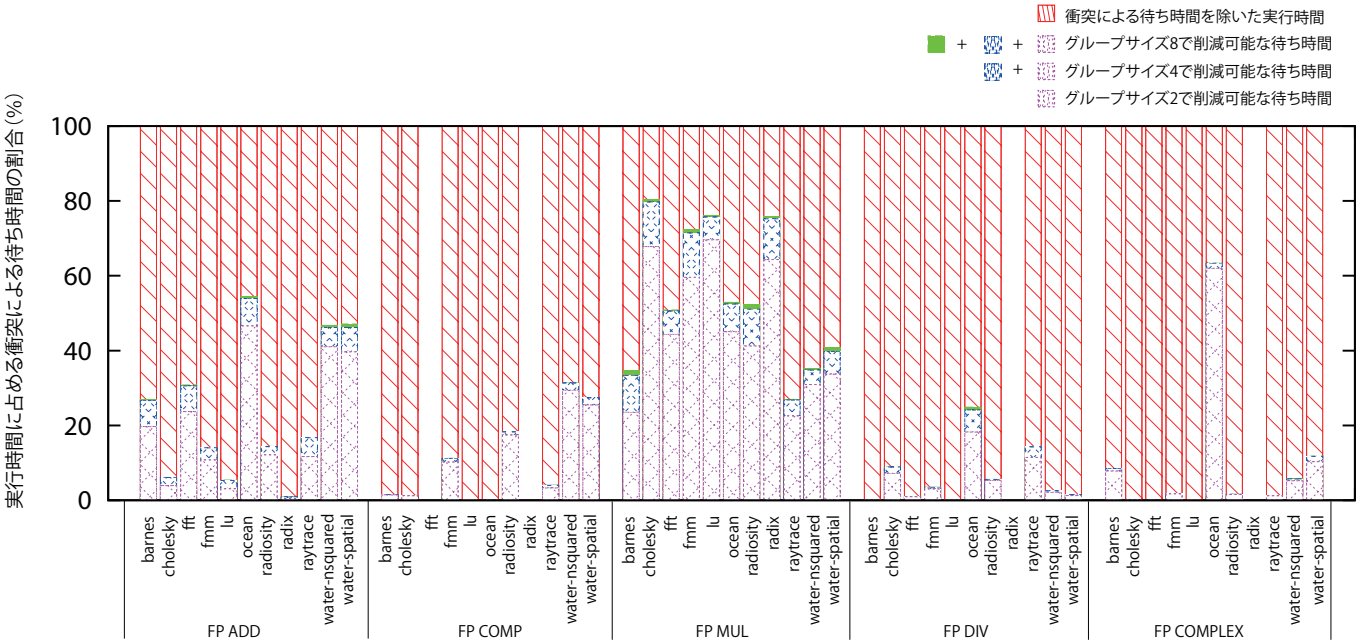

図 4. Splash2 における衝突による待ち時間が占める割合

トを使用する命令を全て完了するまでにかかった時間の内, 衝突による待ち時間が占める割合, 及び命令の委託実行により解決可能であるものの割合を表した図である. cholesky を実行した際の浮動小数点乗算器を例に説明を行う. このプログラムに含まれる全ての浮動小数点乗算命令を完了するのに必要とした時間を 100%とする. この実行時間には衝突による待ち時間も含まれており, 実行時間のうち約 68%がグループサイズ 2 で解決可能な衝突によるものであった. また, グループサイズ 2 では解決できないがグループサイズ 4 で解決することができる衝突が約 12%存在し, グループサイズ 8 でのみ解決可能な衝突が約 1%存在した. したがってグループサイズを 4 としてこのプログラムを実行した場合, 浮動小数点乗算にかかる時間を約 80%削減することが可能となる.

本シュミレーションにおいてグループサイズを 8 とした場合, ほぼ全ての衝突を解消することが可能であった. したがって, 図 4 においてグループサイズ 2,4,8 の値を全て足したものが, 各 FU を利用する命令を実行し終わるまでに掛かった時間のうち, 衝突による待ち時間が占める割合のほぼ全てを示している.

FU ごとに比較を行うと, 浮動小数点乗算器を利用する際が突出して衝突が起きやすく, 次に浮動小数点加算器で衝突多くが発生していることが確認できる. これらはより複雑な演算と比べ, その構成が高度にパイプライン化されており, Issue レイテンシが短いという特徴がある. 一方でこれらの命令はプログラム内で頻繁に利用されるため衝突が多く起きたと考えられる. 一方で平方根などの複雑な演算命令は Issue レイテンシが長い一方で, それらが発行される頻度が低いために衝突が起きる確立が低く, 衝突により発生する待ち時間が実行時間に占める割合も比較的短くなったと考えられる.

表 VII は, グループサイズごとの各 FU の待ち時間の割合の平均値を示す. グループサイズが 2 から 4 になった時が最も待ち時間の削減率が高いことが確認できる. また, グループサイズ 8 においては衝突のほぼ全てが解消され, 待ち時間の実行時間に占める割合が非常に小さくなっている.

表 II
グループサイズごとの実行時間に対する待ち時間の割合

	グループサイズ			
	1	2	4	8
FP ADD	23.97%	19.33%	4.38%	0.25%
FP COMP	8.64%	8.04%	0.60%	0.01%
FP MUL	54.38%	45.70%	8.01%	0.66%
FP DIV	5.67%	4.53%	1.07%	0.08%
FP COMPLEX	8.56%	8.18%	0.38%	0.00%

A. *FU* の割り当てアルゴリズムの連鎖的命令委託への影響

ここでは第 V 節に記述した, 命令の委託実行を行ったために, 割り当て先のコアの命令列を乱すという連鎖的な事象に対し, 割り当て先を決定するアルゴリズムによる違いについて評価を行う. Splash2 と MediaBench を混合させたタスクについて, グループサイズを 8, コア間の命令の委託にかかるオーバーヘッドをゼロとしてシミュレーションを行った.

表 III は各コアが命令を実行しようとした際に, 自身の FU が他のコアによって使用されているために他のコアへ命令の委託を行う必要が発生する確率を FU ごとに表したものである. LRU による割り当て先の決定においては浮動小数点加算命令, 浮動小数点乗算命令以外の命令については委託先の命令列を乱す確立が 1%より小さいことが確認できる. 優先度により割り当て先を決定するアルゴリズムと比べて, LRU による割り当ての方が委託先のコアの命令列を乱しにくいことが確認できる. これは,LRU がより FU の使用される確率が低いコアを選ぶという点に加え, 優先度による割り当てではタスクスケジューリングにおけるプロセッサ親和性も影響を及ぼしている. あるコアに浮動小数点演算の多いタスクがマッピングされた場合, プロセッサ親和性により将来においてもそのタスクがマッピングされる. このコアが, 資源競合を起こしたコアにおいて優先度の高い委託先であった場合、委託された命令によって自身の命令列が乱される可能性が高くなる。

B. オーバーヘッドの性能への影響

図 VII-B はコア間の命令委託実行を行う際に発生するオーバーヘッドを考慮してシミュレーションを実行した結果で

表 III
連鎖的な命令の委託実行が発生する確率

	委託先決定アルゴリズム	
	LRU	優先度
FP ADD	2.49%	6.94%
FP COMP	0.06%	0.78%
FP MUL	7.22%	24.14%
FP DIV	0.29%	2.38%
FP COMPLEX	0.15%	2.13%

表 IV
オーバーヘッドを考慮した性能向上率

	グループサイズ		
	2	4	8
オーバーヘッド 0	4.25%	7.34%	7.45%
オーバーヘッド 1	3.38%	6.28%	6.36%
オーバーヘッド 2	2.49%	3.88%	4.33%
オーバーヘッド 3	2.14%	2.80%	4.02%
オーバーヘッド 4	0.24%	0.67%	0.86%

ある. コア間で命令の委託実行を行わないものをベースラインアーキテクチャとする. これに対しグループサイズを 2 から 8 とし, オーバヘッドは 0 から 4 までを想定した. オーバーヘッドが 4 サイクルかかる時点で性能向上はほぼ見込めなくなり, それ以上値が大きくなるとベースラインアーキテクチャとほぼ同じ性能となった. オーバーヘッドの値が 4 より大きい場合に性能がベースラインアーキテクチャと等しくなる理由は, 提案アーキテクチャではオーバーヘッドにかかるサイクルが解消される待ち時間よりも長い場合命令の委託を行わないため, 実行される命令の流れがベースラインアーキテクチャとほぼ等しくなるためである。

また, グループサイズが 4 の場合と比べるとグループサイズを 8 にした場合の性能向上率が非常に小さいことが確認できる. これはグループサイズを 4 とした時点で, 解消可能な衝突の多くを解決できるためである.

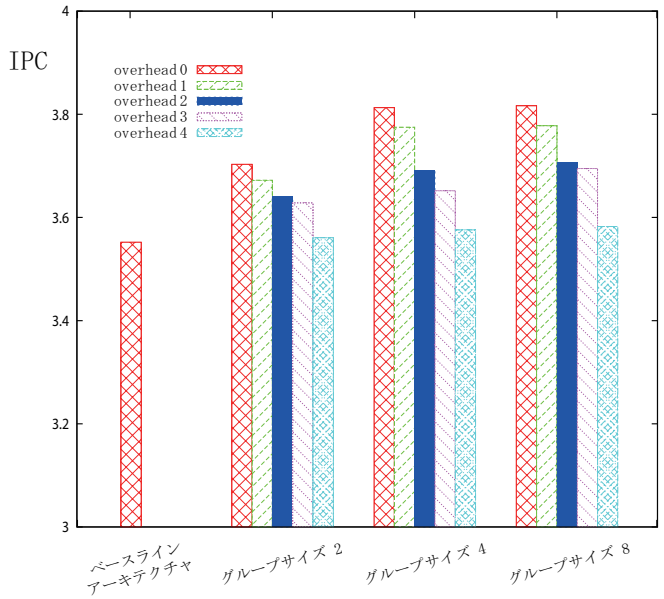

図 5. コア間移動のオーバーヘッドの性能に対する影響

表 IV はベースラインアーキテクチャに対する, 各グループサイズ, オーバーヘッドごとの性能向上率を表したものである. オーバーヘッドの値が 1 から 2 になった場合と, 3 から 4 になった場合において性能向上率が大きく下落していることが確認できる. これらはオーバーヘッドの値が 2 になった時浮動小数点加算命令が, 4 になった時浮動小数点乗算命令がそれぞれ委託実行をできなくなることに起因する. これら 2 つの命令は浮動小数点演算を行う際に最も頻繁に使われるものの一つであり, そのため性能向上に大きな影響を与えた.

VIII. 結論と今後の課題

本稿ではマルチコア, マルチスレッディング CPU における FU の割り当て要求の衝突により発生する待ち時間について, 特に浮動小数点演算命令に注目し, コア間で相互に命令の委託実行をすることで削減が可能であることを示した. 提案手法において, コア間での命令の授受にかかるオーバーヘッドの値をゼロとした場合グループサイズ 2, グループサイズ 4, グループサイズ 8 でそれぞれ 4.25%,7.34%,7.45%の性能向上を見込めることを示した.

提案手法についてハードウェア実装を行う場合, 命令の授受を行う機構やスケジューリングユニットなどの追加コストが必要となる. したがって, 追加したコストに対して本稿で示した性能向上比が見合うものであるかを検証する必要がある. この点については今後の課題として取り組んでいきたいと考えている.

参考文献

[1] D. M. Tullsen, S. J. Eggers, and H. M. Levy, “Simultaneous multithreading: Maximizing on-chip parallelism,” in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 392–403.

[2] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, *Piranha: a scalable architecture based on single-chip multiprocessing*. ACM, 2000, vol. 28, no. 2.

[3] D. Processor, “Cmp implementation in systems based on the intel® core™,” *Intel® Centrino® Duo Mobile Technology*, vol. 10, no. 2, pp. 99–108, 2006.

[4] P. Kongetira, K. Aingaran, and K. Olukotun, “Niagara: A 32-way multithreaded sparc processor,” *Micro, IEEE*, vol. 25, no. 2, pp. 21–29, 2005.

[5] M. Butler, “Amd ” bulldozer ” core-a new approach to multithreaded compute performance for maximum efficiency and throughput,” in *IEEE HotChips Symposium on High-Performance Chips (HotChips 2010)*, 2010.

[6] M. R. Kakoee, I. Loi, and L. Benini, “A shared-fpu architecture for ultra-low power mpsocs,” in *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 2013, p. 3.

[7] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, “Multi2sim: A simulation framework for cpu-gpu computing,” in *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2012.

[8] T. Austin, E. Larson, and D. Ernst, “Simplescalar: An infrastructure for computer system modeling,” *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[9] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The splash-2 programs: Characterization and methodological considerations,” in *Proc. of the 22nd International Symposium on Computer Architecture*, June 1995.

[10] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, “Mediabench: A tool for evaluating and synthesizing multimedia and communications systems,” in *Proc. of the 30th Int’l Symposium on Microarchitecture*, Dec. 1997.

[11] K. Takaki, T. Kurihara, and Y. Li, “On the performance improvement of an architecture towards sharing fpus across cores for the design of multithreading multicore cpus,” in *The 3rd International Workshop on Computer Systems and Architectures*. IEEE, Dec. 2015, pp. 408–411.

[12] A. Fog, “Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus,” *Technical University of Denmark*, 2014.