Vol.2013-ARC-207 No.13
Vol.2013-HPC-142 No.13
2013/12/16

マルチコアプロセッサSKYにおける TLP抽出のための最適化手法

土井 渓太[1]　塩谷 亮太[1]　安藤 秀樹[1]

概要：我々はシングルスレッドの性能向上のためにマルチスレッド実行をサポートするSKYと呼ぶプロセッサアーキテクチャと並列化コンパイラを開発する研究を行ってきた．これまでの研究では，除去することのできるスレッド間依存が存在したり，スレッドの生成が消極的などの問題が存在していた．そこで本論文では，いままでより多くのTLPを抽出するためのハードウェア支援やコンパイラ手法を提案する．最適化の内容は，レジスタ値の送信命令同士の依存による影響の除去，最適な分割位置の決定，死んでいる保存レジスタ値によるスレッド間依存の除去，積極的なスレッド生成である．これらの最適化を行った結果，最適化前と比較して4スレッドの場合，平均で1.47倍の性能向上を達成した．

1. はじめに

マルチコアプロセッサを用いたシングルスレッドのプログラムの性能向上のために，シングルスレッドのプログラムを複数のスレッドに分割し，並列実行することが望まれている．しかし，これは並列プログラミングを行うことができるプログラマが少ないことや，プログラムの生産性が低下するためあまり行われていないのが現状である．

シングルスレッドの並列化を容易にするために，並列プログラミングを簡単化する手法やプログラムを自動並列化する手法，分割を支援するハードウェアを用いる手法がある．並列プログラミングを簡単化するための手法としてOpenMP [5]，OpenMPI [9] などのAPIを利用する方法が存在する．しかし，これらを利用する場合でもソース中にコードやプラグマを追加する必要があるため，依然としてプログラマの負担が大きい．また，プログラマの負担をなくすために自動で並列化を行う商用コンパイラが存在する．しかし，コンパイラによる並列化だけでは，並列性を抽出できないプログラムも存在する．それに対して，ハードウェアの支援によって並列化を容易にする様々なアーキテクチャの研究 [3,4,8] が行われている．

我々は，ハードウェア支援により並列化を容易にするアーキテクチャの一つとしてSKY [2,10] を提案している．SKYは専用コンパイラによって既存のバイナリ中にスレッドへの分割に必要な命令を自動で挿入し，ハードウェア支援によってシングルスレッドを分割，実行する．命令の挿入はコンパイラが完全自動で行うため，プログラマの負担はない．また，バイナリから並列化を行うため，プログラマが自由にプログラミング言語を選択することができるだけでなく，ソースコードが手元にないプログラムの並列化や，エンドユーザが各々の環境に合わせたコンパイルを行うことが可能である．

これまでのSKYの研究においては，必ずしも高い性能が得られていない場合があった．この原因を調査した結果，並列化において，除去可能なスレッド間依存が存在したり，スレッドの生成が消極的であるなどの問題によりTLPを十分に抽出できていないことがわかった．本論文では，これらの問題について詳しく説明し，その解決手法を提案する．提案手法により問題を解決した結果，SPECfp2000ベンチマークプログラムにおける4スレッドのマルチスレッド実行ではシングルスレッド実行と比較して平均で2.74倍，問題解決前と比較して1.47倍の性能を達成した．

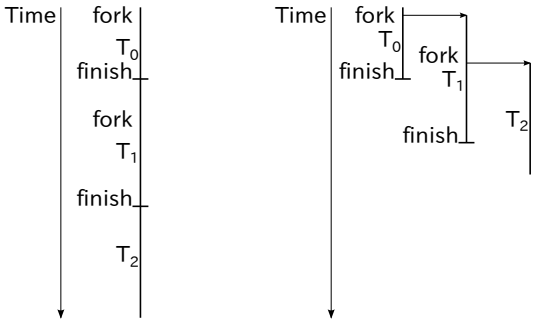
本論文の以降の構成は以下の通りである．2節ではSKYについて簡単に説明し，3節では，存在した問題について詳しく説明する．4節では，各問題の解決方法を提案する．そして，5節で提案手法について評価を行い，6節でまとめる．

2. SKYプロセッサ

SKY [2] とは，シングルスレッド用プログラムをマルチスレッド実行するためのマルチコアプロセッサである．シングルスレッド用プログラムのスレッドへの分割は，専用のコンパイラにより自動で行われる．

本節では，まずSKYのマルチスレッドモデルについて

[1] 名古屋大学大学院工学研究科
Graduate School of Engineering, Nagoya University



(a) シングルスレッド実行　　(b) マルチスレッド実行

図 1　SKY のマルチスレッドモデル

説明する．その後，SKY のハードウェアとコンパイラについて説明する．

2.1　SKY のマルチスレッドモデル

SKY のマルチスレッドモデルは，スレッドの並列実行のオーバヘッドを削減するために，通常のマルチスレッドモデルと比べて次に示す制約を課している．この制約は，Multiscalar [8] と同様のモデルである．

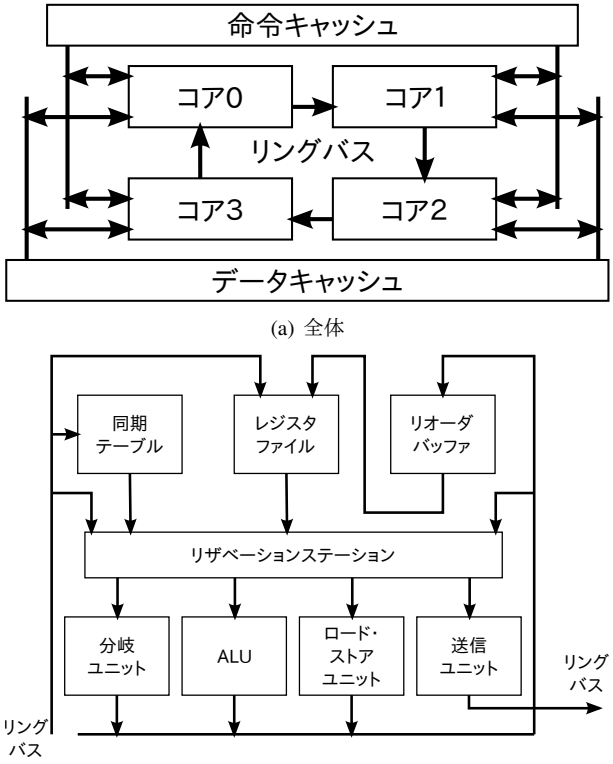
- 各スレッドは，逐次実行における動的に連続する部分で構成される
- 各スレッドは，逐次実行の順において自分の直後の部分を子スレッドとして生成する

SKY では，コンパイラがプログラムのフォーク点と子の開始点を決定し，専用命令である fork 命令と finish 命令を挿入する．ここで，フォーク点とはスレッドを新たに生成するプログラム上の位置であり，子の開始点とは子スレッドが実行を開始するプログラム上の位置である．ハードウェアでは，挿入した fork 命令と finish 命令を実行することでスレッドの生成と終了の動作を行う．

SKY でマルチスレッド実行を行なっている様子を**図 1** に示す．**図 1(a)** は，fork 命令と finish 命令が挿入された命令列をシングルスレッド実行した場合を示しており，**図 1(b)** は，それを並列実行した場合の様子を示している．図のように各スレッドは fork 命令の実行により子スレッドを生成し，子スレッドを生成したスレッドは finish 命令を実行するとスレッドの実行を終了する．

2.2　ハードウェア構成

SKY のアーキテクチャを**図 2** に示す．**図 2(a)** に示すように，SKY は複数のコア（スーパスカラプロセッサ）で構成され，各コアは単方向のリングバスで結合されている．図に示すように，n コアを持つ SKY では，コア i はコア $(i+1) \bmod n$ にのみデータを送り，コア $(i-1) \bmod n$ からのみレジスタ値を受け取る．以下，コア $(i+1) \bmod n$ をコア i の後続コア，コア $(i-1) \bmod n$ をコア i の先行コアと呼ぶ．スレッドは実行開始してから終了までコア 1 つ



(a) 全体

(b) コア内部

図 2　SKY のアーキテクチャ

を占有する．あるコア上のスレッドは fork 命令を実行することによって，後続コアに子スレッドを生成する．他のスレッドが後続コアを既に占有している場合は，fork 命令は無効化され，子スレッドを生成しない．2.1 節で示した性質より，fork 命令を無効化した場合もスレッドが逐次に実行されるだけで，プログラムの意味は保たれる．

図 2(b) にコア内部の構成を示す．SKY のコアは通常のスーパスカラの構成に加え，コア間でのレジスタの通信用に送信ユニットと同期テーブルを持つ．レジスタの送信は，send と呼ぶ専用の命令を送信ユニットで実行することにより行う．受信は同期テーブルを用いてハードウェアにより暗黙的に行われる．

2.3　コンパイラ

SKY のコンパイラはスレッドの並列実行による性能利得が大きくなるようにプログラムを分割し，その後スレッド間で送信すべきレジスタを求め，それを送信する send 命令をプログラムに挿入する．SKY においてプログラム分割とは，フォーク点と子の開始点を定めることである．

コンパイラは次の 4 ステップで動作する．

(1) プログラムの分割候補の決定
(2) 性能利得の推定
(3) 採用する分割の決定
(4) send 命令の挿入

2.3.1　プログラムの分割候補の決定

SKY では，プログラムの制御フローグラフにおける制御

等価 [7] な部分に着目してプログラム分割の候補を決定する．制御フローグラフにおいて，基本ブロック X から出口ノードに向かう全てのパスで Y を通るとき，Y は X を**後支配** [7] するといい，入り口ノードから Y へ向かう全てのパスで X を通るとき，X が Y を**支配** [1] するという．X が Y を支配し，Y が X を後支配するとき，X と Y は制御等価であるという．

SKY の実行モデルを満たすためには，子の開始点がフォーク点を後支配する必要がある．しかし，このような組は非常に数が多く，現実的な計算量で分割できないため，制御等価な組をプログラム分割の候補とする．

2.3.2 性能利得の推定

各候補についてスレッド並列実行による**性能利得**をプロファイルを用いて計算する．あるスレッドと子スレッドの並列実行による性能利得とは，2 つのスレッドの実行がオーバラップする間に子スレッドで実行された命令の数と定義する．コンパイラは，フォーク点と子の開始点の間の距離，およびスレッド間でデータ依存関係にある命令間の距離を求め，その最小値を性能利得の推定値とする．ここで点 p_1 から点 p_2 までの距離とは，p_1 から p_2 に至る各パス上の命令数の，パスの実行確率による重み付き平均値とする．

2.3.3 採用する分割の決定

スレッドの分割候補から性能利得が最も大きくなる分割の組み合わせを選択する．SKY のマルチスレッドモデルでは，1 つのスレッドは 1 回のみ新しいスレッドを生成する．この制限により，ある分割を実行すると他の分割が実行できない場合が存在する．分割可能な組み合わせのうち，最も性能利得が大きくなるものを最終的なプログラムの分割とする．

選択後は，フォーク点と子の開始点を制御等価な基本ブロックの先頭として，fork 命令と finish 命令を挿入する．

2.3.4 send 命令の挿入

あるスレッドにおいて子の開始点で生きているレジスタ値が子の開始点に**到達** [1] することが決定する点に send 命令を挿入する．送信するレジスタは，**真の送信レジスタ**と**転送レジスタ**の 2 種類が存在する．真の送信レジスタは，現スレッドで定義され，以降のスレッドで使用する可能性があるレジスタである．転送レジスタは現スレッド以前のスレッドで定義され，現スレッド以降のスレッドで使用する可能性があるレジスタである．転送レジスタに関しては，フォーク点において値の到達が既に決定しているため，フォーク点の直後に send 命令を挿入する．

3. SKY の問題

性能向上を妨げている原因を調査したところ次の 4 つの問題が判明した．

(1) send 命令間の依存による TLP 利用の阻害
(2) スレッドの分割箇所の制限
(3) 死んでいる保存レジスタ値のスレッド間依存
(4) 消極的なスレッド生成

本節ではこれらの問題について説明する．

3.1 send 命令同士の依存による TLP 利用の阻害

利得計算においては，真の送信レジスタの send 命令と転送レジスタの send 命令の間にある真の依存を考慮していない．これは，転送レジスタの send 命令とスレッド内の後続の命令の間に依存が一切なく，後続の命令が send 命令を追い越して実行することが可能であるため，TLP の利用を妨げないと考えたためである．しかし，転送レジスタの send 命令の実行が待たされすぎると，この命令が ROB の先頭で詰まり，ROB が一杯となるため後続命令の実行が停止する問題が存在した．

そのため，ハードウェアによって転送レジスタの send 命令による ROB の詰まりを解消し，後続命令の実行を続ける方法を 4.1 節で提案する．

3.2 スレッドの分割箇所の制限

これまでは子の開始点の位置を基本ブロックの先頭として利得計算を行なっていた．そのため，性能利得が最大になる分割を行うことができない場合が存在した．そのような場合のプログラムの例を**図 3** に示す．

同図のプログラムでは，基本ブロック BB1 と BB4 が制御等価な関係であるため，BB4 以降をを子スレッドとする分割候補が生成される．従来通り基本ブロックの先頭である破線 A の位置でスレッドを分割した場合は，命令 i3，i4 と i5 の間に距離が短いスレッド間のデータ依存が存在する．この依存により，この分割候補は性能利得が得られないと判断され，非採用となる．しかし，破線 B の位置でスレッドを分割した場合は，同じイタレーションの i3，i4 と i5 は同一スレッドになり，その間に存在したスレッド間のデータ依存が除去される．残りのスレッド間のデータ依存は，i5 から次イタレーションの i5 のようにループボディのサイズと等しい距離になる．そのため，破線 B の分割候補はループ全体の並列化が可能な性能利得が大きい分割と判断され，採用される．

このように子の開始点の位置を基本ブロック内の別の位置に移動することで性能利得を向上させ，TLP の抽出量を増加させることが可能である．この移動の際には，基本ブロック内の命令スケジューリングも行い，TLP の抽出量が最大になるようにする．しかし，総当りで子の開始点の位置の移動と命令スケジューリングを行う場合は計算量が大きくなる．そこで本論文では，少ない計算量で最適な子の開始点の位置を決定する方法を提案する．この計算方法については 4.2 節に示す．

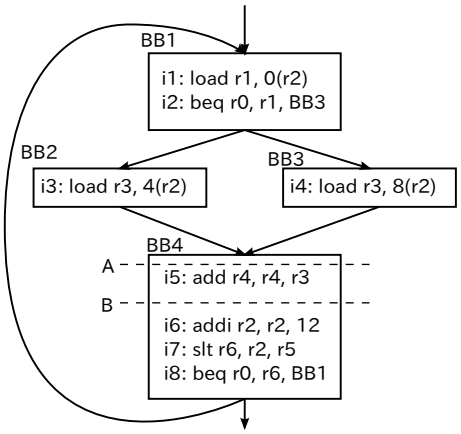


図 3 TLP の抽出が上手くいかない例

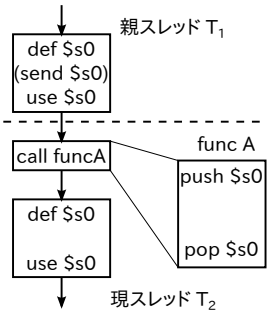


図 4 死んでいるレジスタ値の依存が生じる例

3.3 死んでいる保存レジスタ値のスレッド間依存

死んだレジスタであってもスタックへの退避が行われる場合が存在する．この退避は，関数がどこで呼び出された場合でも保存レジスタ値を保護するために存在し，プログラムの意味を守るために必要な操作である．このようなレジスタ値はレジスタライブ解析では生きていると判断される．そのため，あるスレッドが生成したレジスタを子スレッドが退避する場合は，プログラムの意味上では死んでいるレジスタであっても子スレッドに送信していた．この死んでいる保存レジスタの送信とスタックへの退避との間に生じるスレッド間依存によって，TLP の抽出が阻害される問題が存在した．

図 4 に生きていると判断したが，プログラムの意味上では死んでいるレジスタ値が生じる例を示す．図は制御フローを表しており，図中の$s0 は保存レジスタであり，def，use，send，push，pop はそれぞれレジスタ値の定義，使用，送信，スタックへの退避と復元を示し，call は関数呼び出しを示している．図にあるように$s0 は親スレッドにおいて定義，使用される．そして，現スレッドにおいては，親スレッドにおいて定義された$s0 は func A の呼び出し後のスタックへの退避でのみ参照されるプログラムの意味的には死んでいるレジスタ値となる．過去の研究では，レジスタライブ解析によって子スレッドの開始点において$s0 は生きていると判断されるため，def $s0 の後に send 命令を挿入していた．そのため実行時には，この send とスタックへの退避の間にスレッド間依存が生じており，TLP 抽出が妨げられていた．このスレッド間依存を生じさせるレジスタ値の定義-使用関係は，プログラムの意味を保持する上で不必要な関係であるため，除去することが可能である．

このレジスタ値の定義-使用関係を除去する方法として，次の 3 つが挙げられる．

(1) 定義側である send 命令を除去
(2) 参照側であるスタックへの退避を除去
(3) ダミーの値を定義する命令を挿入し問題のレジスタを明示的に破壊

単純に send 命令を除去すると，スタックへの退避時にレジスタ値が未定義かつ未受信状態のままとなり，このスタックへの退避命令が実行できなくなる．参照側であるスタックへの退避を除去する方法は，関数が別の場所から呼び出される可能性があるため，保存すべき保存レジスタ値を破壊してしまう恐れがある．別の場所から呼び出された場合に対応するために，スタックへの退避を除去する際に関数のコピーを作成する方法もあるが，命令数が大幅に増加するためメモリの利用効率などに悪影響を与える．そこで本論文では，ダミーの値を定義する命令を挿入し，レジスタ値を明示的に破壊することでこの問題に対処する．具体的な方法については，4.3 節で述べる．

3.4 消極的なスレッド生成

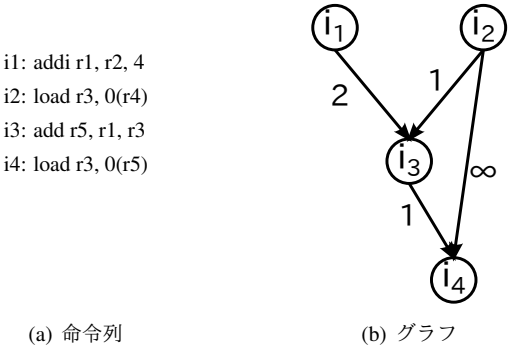
これまでは，fork 命令実行時に後続コアが他のスレッドに占有されている場合，スレッドの生成を諦めていた．しかし，fork 命令の無効化後すぐに後続コアで実行中のスレッドが終了した場合，諦めていたスレッドを割り当てることで性能利得を得られる場合がある．性能利得が得られる場合の判断方法とスレッド生成の方法について 4.4 節で示す．

4. 提案手法

本節では，3 節で説明した各問題を解決するための手法を提案する．

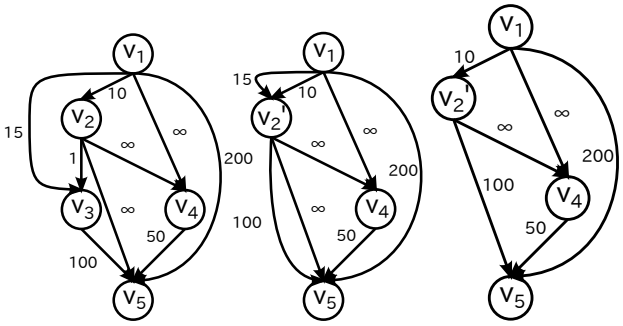
4.1 未受信の転送レジスタの自動転送

3.1 節で述べたように真の送信レジスタの send 命令と転送レジスタの send 命令間に生じた依存のため，転送レジスタの send 命令によって ROB が詰まりストールが生じる．この問題を解決するために，受信待ちの転送レジスタの send 命令を無効化し，NOP とすることでストールを回避する．そして，レジスタ値の転送はハードウェアにより自動で行うようにする．その方法として，同期テーブルに転送するかどうかのフラグを用意し，フラグが立っているレジスタ値を受信した場合はその受信値を後続スレッドへと転送する．send 命令をデコードした時に，対応するレジスタ値が未受信かつ自スレッドにおいて未定義の場合，未



(a) 命令列　　(b) グラフ

図 5　グラフへの変換



(a) 合成前　　(b) 合成途中　　(c) 合成後

図 6　ノードの合成

受信の転送レジスタの send 命令とわかる．この場合フラグをセットし，send 命令を無効化する．

4.2　子の開始点の位置調整

3.2 節で述べたように，子の開始点を基本ブロックの先頭として性能利得を推定していたため，十分な性能利得を持ったスレッドに並列化できてない場合が存在した．この問題を解決するために，各分割候補の性能利得の推定の際に基本ブロック内命令スケジューリングを行いつつ，子の開始点の最適な位置を調べ，分割候補毎の最適解における性能利得を推定する．以下，分割候補において子の開始点を挿入する基本ブロックのことを調整対象の基本ブロックと呼ぶ．

以下，子の開始点を挿入する基本ブロックのことを調整対象の基本ブロックと呼ぶ．以降では，まず問題のモデル化について示す．次に，計算量の削減のために入力の簡単化について示し，その後モデル化した問題を解くアルゴリズムを示す．最後に，モデル化した問題の解から子の開始点の位置を調整し，コードを生成するまでの流れを示す．

4.2.1　問題のモデル化

子の開始点の位置調整問題は，重み付き有向グラフを 2 つのサブセット S,T に分割するカット問題に置き換えることが可能である．子の開始点の位置調整問題と重み付き有向グラフの関係を以下に示す．

- ノード：
  - 開始ノード v_s：調整対象の基本ブロック後方の命令をまとめたもの
  - 終端ノード v_e：調整対象の基本ブロック前方の命令をまとめたもの
  - 他のノード：調整対象の基本ブロック内の命令
- エッジ：
  - 順序制約
  - 基本ブロックの境界 (基本ブロックを超えた命令スケジューリングを防ぐため)
- エッジの重み：
  - 真のデータ依存によるエッジの場合はデータ依存距離
  - 逆依存と出力依存，基本ブロック境界によるエッジの場合は ∞
- サブセットへの分割：現スレッドと子スレッドにプログラムを分割
- サブセット S：現スレッド
- サブセット T：子スレッド

図 5 に命令列からグラフへの変換例を示す．ただし，図 5 は調整対象の基本ブロック内の命令のみについて示しており，v_e と v_s に相当する部分は省略している．

今回解くグラフのカット問題の制約と評価関数および目的を以下に示す．

- 制約：
  - 全てのカットエッジが S から T に向かうエッジ
  - 開始ノード v_s は S に属す
  - 終端ノード v_e は T に属す
- 評価関数：カットエッジ上の最小の重み
- 目的：評価関数の値を最大にする

SKY ではプログラムの逐次実行における動的に連続した部分をスレッドとして生成するため，プログラムの流れは必ず現スレッド→子スレッドの順にする必要がある．そのため，制約に示すようにサブセット間の順序制約が S から T に向かう方向でなければならない．評価関数はスレッド間のデータ依存距離の最小値を表している．SKY の性能利得はスレッド間の依存距離の最小値で制限されるため，依存距離を最大にする分割を選ぶことを目的としている．

4.2.2　問題の簡単化

グラフのカット問題では，ノードの数やエッジの本数が多いほど計算にコストがかかるため，計算量の削減のためにノードの合成を行い，グラフを簡単化する．ノードの合成とは，複数のノードを 1 つのノードへと変換する動作である．合成されたノードは，命令の集合を表し，集合内部の命令の順序制約は常に守られる．ノードの合成は，2 つのノードを別々のサブセットに分けると最適解にならない可能性が高い場合に行われる．

図 6 を用いて，**図 6(a)** のグラフの v_2 と v_3 を合成し v_2' に変換する場合の流れを説明する．ノードの合成では，はじめに v_2 と v_3 の間のエッジを取り除く．そして，v_2 と v_3

を除去し．新たなノードを v_2' 生成する．このとき，**図 6(b)** に示すように v_2 と v_3 に接続されていたエッジを全て v_2' に接続する．最後に v_2' と v_1 のように複数のエッジがノード間に存在する場合は，エッジの向きが同一方向の場合は重みが小さいものを残し，エッジの向きが異なる場合はその 2 つのノードの合成を行う．重みの小さいエッジを残すのは，カットエッジの重みの最小値が評価の対象になるため，同時にカットされるエッジは小さいものだけ考慮すれば良いからである．異なる向きのエッジがある場合，それらのノードが別々のサブセットになるカットは制約に違反するためできない．そのため，異なる向きのエッジでつながるノードはノードの合成により 1 つにまとめる．合成を行った結果，グラフは**図 6(c)** の形になる．

合成後のノードは．調整対象の基本ブロック内で真のデータ依存関係にあるノードである．これを対象にする理由は，基本ブロック内のデータ依存は距離が短い場合が多く，その依存がスレッドをまたぐ様な分割が最適になる可能性が殆ど無いと考えられるためである．

簡単化したグラフは，結果として次のようなグラフになる．

- v_s または v_e と他のノードの間は，基本ブロック境界をまたいでいるため，必ずエッジが存在
- 調整対象の基本ブロック内で真のデータ依存関係にある命令は合成されているため，残っているデータ依存は逆依存と出力依存だけであるから両端が v_s でも v_e でもないノード間に存在するエッジの重みは ∞

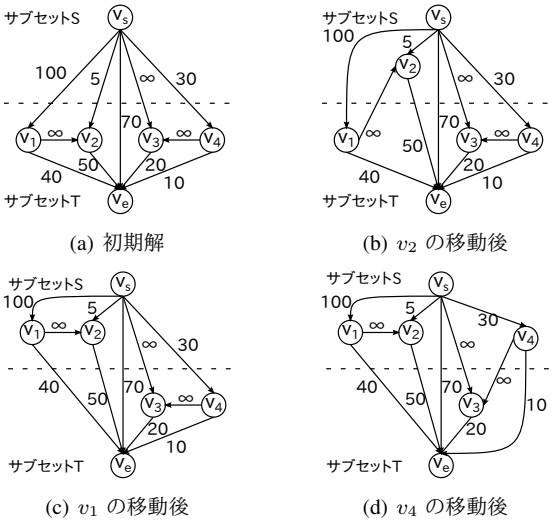
4.2.3 アルゴリズム

最適解を求めるために，初期解から評価値の改善がある限り，カットエッジを除去するためのノードの移動を繰り返し，改善しなくたった時点の解を最適解とするアルゴリズムを用いる．ここで初期解は，開始ノードのみをサブセット S とし，残りのノードをサブセット T とした状態にする．これは基本ブロックの先頭に子の開始点を挿入した場合と同じ分割である．また，ノードは T から S への一方向にのみ移動する．

本アルゴリズムは，次のステップの繰り返しで構成される．

(1) カットエッジの中で最小の重みをもつエッジをカットエッジから除去
(2) 制約条件に違反しているカットエッジの除去
(3) 解の評価

カットエッジの除去は，カットエッジの終端のノードをサブセット S に属するように移動することで行う．評価値はカットエッジの中の最小の重みを持つエッジで決定されるため，そのエッジを除去することで改善を目指す．ノードの移動によって制約条件に違反する T から S に向かうカットエッジが生じた場合は，そのカットエッジも除去するために違反しているカットエッジの T 内のノードも S に移動する．ノードの移動ができなくなるか，前回の評価値よりも悪化していた場合はこれ以上の解の改善ができないとして，繰り返しを終了する．



(a) 初期解　(b) v_2 の移動後　(c) v_1 の移動後　(d) v_4 の移動後

図 7 アルゴリズムの動作例

図 7 を例に動作の具体的な流れを示す．図中の破線はサブセットの境界を示しており，破線の上側がサブセット S，下側がサブセット T である．図 7(a)，(b)，(c)，(d) の順に解の探索が進んでいる様子を示している．

(1) **図 7(a)** に示す初期解の状態では，最小の重みを持つカットエッジは v_s と v_2 を結ぶエッジでその重みは 5 である．このカットエッジを除去するため、v_2 を移動する．
(2) **図 7(b)** の分割では，v_1 と v_2 の間にあるエッジが T から S に向かうカットエッジとなり，制約条件に違反する．そのためこのエッジを除去するために v_1 も移動する．
(3) **図 7(c)** の分割では，制約に違反しているエッジが存在しないため解の評価を行う．評価した結果，評価値は v_s と v_4 の間にあるカットエッジの重みである 30 となり，前回の評価値 5 よりも改善しているためアルゴリズムの繰り返しを続ける．v_s と v_4 の間のカットエッジを除去するため v_4 を移動する．
(4) **図 7(d)** の分割では，制約に違反しているエッジがないため評価を行う．その結果，評価値は v_4 と v_e の間にあるエッジの重みである 10 になり，前回よりも悪化している．このため前回の解である $S = \{v_s, v_1, v_2\}, T = \{v_3, v_4, v_e\}$ が最適解である．

4.2.4 子の開始点の位置調整への反映

子の開始点の位置調整は性能利得の推定の直前に行い，出力された解から分割候補の性能利得の推定を行う．性能利得は，スレッド間の真のデータ依存距離とフォーク点から子の開始点までの距離の最小値として求められる．ここで，スレッド間の真のデータ依存距離の最小値は，前節のアルゴリズムの実行で求めた最適解における評価値である．

全分割候補の性能利得が推定できた後は，従来と同じように性能利得が最大になる組み合わせを探し，採用する分割を決定する．

採用する分割を決定した後は，採用した分割のフォーク点と子の開始点に命令の挿入を行う．この際に採用した分割候補の調整対象の基本ブロック内の命令は得られた解に従って順序を定めなければならない．具体的には，調整対象の基本ブロックにおいて，各サブセット内部の命令の元の順序を保ったまま，S に属していた命令を基本ブロックの先頭側に移動し，T に属していた命令を基本ブロックの末尾側に移動する．その後 finish 命令は，S と T の境界に挿入される．

fork 命令と finish 命令の挿入後は従来通り send 命令の挿入を行う．

4.3 死んでいる保存レジスタ値の破壊

3.3 節で述べたように，死んでいる保存レジスタ値のスタックへの退避によって不要なスレッド間依存が生じる問題が存在した．この問題の解決方法として，スタックへの退避前に死んでいるレジスタ値を明示的に破壊する方法を提案する．具体的には，図 4 の T_2 において，call funcA の前に$s0 を再定義するためにゼロレジスタ同士を加算をする命令を挿入する．これにより，push $s0 と親スレッドである T_1 との間にあった定義-使用関係が除去され，より多くの TLP 抽出が可能になる．

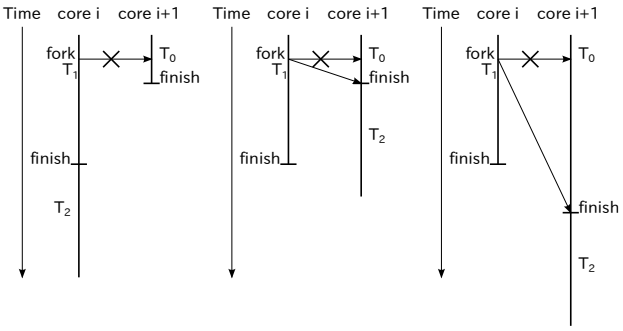
死んでいる保存レジスタ値を破壊するための命令を挿入する手順を以下に示す．

(1) 各関数呼び出し命令を実行する時点で死んでいる保存レジスタ値を計算
(2) (1) で見つけた死んでいる保存レジスタ値のスタックへの退避は依存が生じないとして性能利得を計算
(3) 性能利得が最大になるようにプログラムを分割
(4) 子の開始点以降で，当該関数呼び出し直前に破壊用の命令を挿入

4.4 スレッドの予約

fork 命令実行時に後続コアが他のスレッドに占有されていた場合のスレッド生成方法について述べる．fork 命令実行後，後続コアが空いた場合に子スレッドを生成した場合の性能向上について**図 8** を用いて考える．

図 8(a) は 2 つの連続したコアにおいて，コア i が fork 命令を実行した際に後続コアが占有されていたため fork 命令を無効化し，コア i で分割予定であったスレッドを継続して実行した場合の様子を表している．そして，図 8(b), (c) は後続コアが空いた後に子スレッドを割り当てて実行した場合の様子を示している．**図 8(b)** では，後続コアが空いた後に子スレッドを割り当てることでスレッド T_1 と T_2 を並列実行することができるため性能が向上する．一方で



(a) スレッド生成なし (b) 性能向上する場合 (c) 性能低下する場合

図 8 スレッド生成を遅らせた場合の効果

図 8(c) では，子スレッドの実行開始時点で既にコア i で実行しているスレッド T_1 が終了しており，並列実行ができない．そのため，スレッド T_1 が終了してから T_2 の割り当てまでの待ち時間の分だけ図 8(a) のようにコア i で続けてスレッドを実行した場合よりも実行時間が延びる．

以上のことから，次のように実行することが好ましいことが分かる．fork 命令を実行したスレッドが finish 命令に到達するよりも先に後続コアが空く場合，その時点で子スレッドを生成する．逆に fork 命令を実行したスレッドが finish 命令に到達した時点で後続コアが空いていない場合，子スレッドを生成せず，自コアで実行を継続する．

利益がある場合にスレッドを生成するために，後続コアが空いていない場合にスレッドの生成を先延ばしにするスレッドを予約する提案する．このスレッドの予約は，fork 命令実行時に後続コアにスレッドが存在した場合に行われる．予約が行われると後続コアが空いた時にスレッドを自動的に生成する．予約後，スレッドが生成されるよりも先に fork 命令を実行したスレッドが finish 命令を実行した場合は，予約を取り消し，予約していたスレッドは同じコアで実行される．これにより，図 8 で示した性能向上が得られる場合のみ子スレッドを生成し，子スレッドを生成すると性能低下する場合はスレッドの生成を無効化することができる．

このスレッドの予約を行うための追加ハードウェアとして，スレッドの予約の情報を保存するための専用レジスタ (スレッド予約レジスタ) と生成した場合にスレッドに送信するレジスタ値を保存するバッファ (送信予約バッファ) を用意する．スレッド予約レジスタは有効ビットと子スレッドの先頭 PC を保存するフィールドで構成されたレジスタである．送信予約バッファは論理レジスタ番号分のエントリを持つ FIFO で構成されたバッファで，各エントリは論理レジスタ番号，送信するレジスタ値で構成される．コアは，fork 命令実行時に後続コアが使用中であった場合は，子スレッドの開始 PC を自コアのスレッド予約レジスタに保存し有効ビットをセットする．そして，後続コアが空いたことが分かった場合でスレッド予約レジスタが有効の場

表 1　測定区間について

ベンチマーク	測定箇所	実行時間に占める割合	スキップ命令数	測定命令数
ammp	mm_fv_update_nonbon 関数内の最後の ii ループ	70%	2.271936G	300M
applu	ssor 関数内の istep ループ	99%	1.194404G	2.347G
apsi	DVDTZ 関数全体	10%	10.510029G	538M
art	scan_recogize 関数内の j ループ	82%	6.851494G	1.57G
equake	main 関数内 Time integration ループ	61%	2.249227G	105M
mesa	DrawMesh 関数全体	95%	257.929M	296M
mgrid	mg3xdemo 関数内 IT ループ	50%	348.381M	891M
swim	calc3 関数全体	30%	2.945917G	105M

合は後続コアに子スレッドを生成した後にスレッド予約レジスタを無効化する．スレッド予約中の send 命令の実行に関しては，送信予約バッファに送信するレジスタ値と論理レジスタ番号を保存する．予約したスレッドが生成された場合は，送信予約バッファに保存されたレジスタ値を後続コアへと送信する．finish 命令実行時にスレッド予約レジスタが有効であった場合はスレッド予約レジスタと送信予約バッファの全エントリを無効化し，自コアで分割する予定だったスレッドを続けて実行する．

5. 評価

本節ではまず，評価環境について述べ，その後 4 コアの場合の提案手法の有効性について評価する．

5.1 評価環境

トレース駆動シミュレータを作成し，評価を行った．評価に用いたトレースは，SimpleScalar Tool Set Version 3.0 [6] を利用して採取した．並列化する前のベンチマークのバイナリは，GNU GCC Version 2.7.2.3（コンパイルオプション: -O6 -funroll-loops）を用いて作成した．ISA は SimpleScalar PISA である．

ベンチマークとして，SPECfp2000 の中から gcc でコンパイルでき，その後 SKY のコンパイラでコンパイルできた 8 本を使用した．入力として ref を使用し，シミュレーション区間としてホットスポットとなる関数やループ部分を選択した．**表 1** に各ベンチマークのシミュレーション区間を示す．評価したプロセッサは，コアプロセッサを**表 2** に示す構成とし，マルチスレッド実行時のコア数を 4 とした．キャッシュと主記憶は全てコア間で共有しており，**表 3** に示す構成である．

また，比較対象として Intel Composer XE2013 1.117 で並列コンパイルし，実機 Intel Xeon E5-2670 上で測定した場合の性能向上率を測定した．表 1 の測定区間で測定するため，プログラム中に時間計測用のコードを埋め込み測定を行った．Intel Composer XE によるコンパイル時のオプションは，シングルスレッドでは-O6 -funroll-loops -no-simd とし，マルチスレッドではそれに加えて-parallel -par-threshold90 -par-num-threads=4 とした．

表 2　コアプロセッサの構成

Pipeline width	4-instruction wide for each of fetch, decode, issue, commit
Reservation station	64 entries
LSQ	64 entries
ROB	128 entries
Function unit	4 iALU, 1 iMULT/DIV, 4 Ld/St, 4 fpALU, 1 fpMULT/DIV/SQRT, 4 Send
Branch prediction	PAp(4-bit history, 16K-entry PHT) 10-cycle misprediction penalty

表 3　キャッシュの構成

L1 I-cache	64KB, 2-way, 32B line
L1 D-cache	64KB, 2-way, 32B line 2-cycle hit latency
L2 cache	8MB, 8-way, 64B line 12 cycle hit latency
Main Memory	300-cycle minimum latency

5.2 評価モデル

以下のモデルを評価した．

- 最適化なし：最適化を全く行わないモデル
- 自動転送：未受信の転送レジスタを自動的に転送するモデル
- 開始点調整：子の開始点の位置を調整するモデル
- 明示的破壊：保存レジスタ値を明示的に破壊するモデル
- スレッド予約：スレッドの予約を行うモデル
- 全最適化：最適化を全て適用したモデル
- Intel：Intel Composer XE における性能評価のモデル

また，自動転送+開始点調整の様に記述した場合は自動転送と開始点調整の両方を行ったモデルである．

5.3 評価

3 コアにおけるシングルスレッド実行からの性能向上率を**図 9** に示す．このグラフは，Intel モデル以外は SKY シミュレータにおけるシングルスレッド実行から 4 コアで並

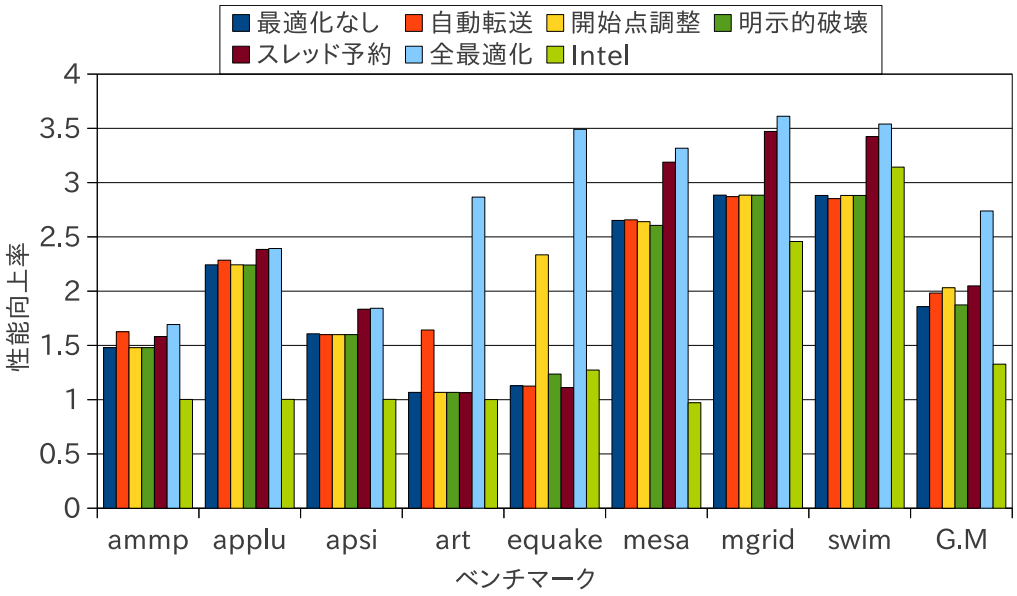


図 9 4 コアにおける性能

列化コードを実行した場合の性能向上率を示している．そして，Intel モデルは実機上のシングルスレッド実行の性能から Intel Composer XE によって 4 スレッドに並列化したコードを実行した場合の性能向上率を示している．

最適化により，最適化前から平均 1.47 倍の性能向上を達成した．また，SKY では，全ベンチマークで Intel Composer XE による並列化以上の性能向上を得ており，Intel Composer XE が上手く並列化できなかったベンチマークも並列化を行うことができた．

各ベンチマークについて見ると，mesa，mgrid，swim ではスレッドの予約を行うことで高い性能向上を達成することができるようになった．これらのベンチマークは，従来から静的にはスレッド間に依存が生じないように上手く分割ができていたため，スレッドの予約により今まで無効にされていた大きな利得が得られるスレッド生成を行うことで高い性能向上を得ることができた．

art は，3.1 節で述べた問題により，最適化前は逐次実行に近い形になっていた．転送レジスタの即時実行により，ストールを除去することで並列性が抽出できるようになった．そして**図 10** に示すように，転送レジスタの即時実行後はスレッドの予約と組み合わせることで高い性能を得ることができた．

equake は，子の開始点を調整することで並列性が抽出できるようになった．**図 11** に示すように，スレッドの予約と死んでいる保存レジスタ値の破壊の片方だけを採用したでは性能はあまり向上せず，両方採用することで大きく向上した．スレッドの予約の追加だけで性能が向上しない理由は，死んでいる保存レジスタ値による依存によって並列性が制限されており，スレッドを生成しても並列性が抽出できないためである．死んでいる保存レジスタ値の破壊を

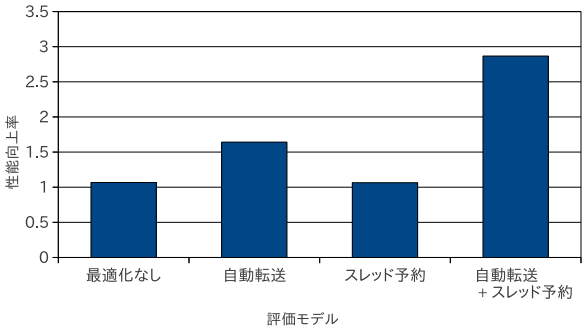


図 10 art における性能の変化

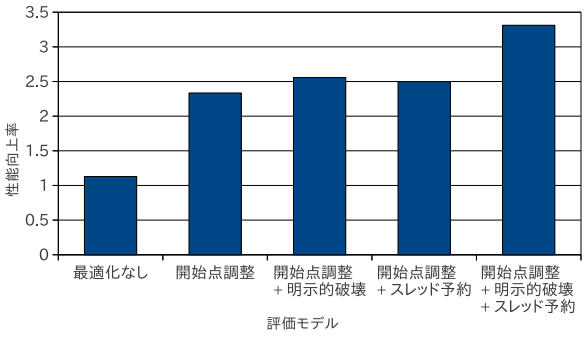


図 11 equake における性能の変化

追加するだけでは性能向上しない理由は，依存が解消されたものの消極的なスレッド生成により，性能向上に寄与するスレッド生成の無効化が多く存在するためである．そのため，両者を解決することで初めて大きな性能向上を得ることができる．

今回の提案手法で大きな性能向上を得ることができなかった ammp と applu に関しては，測定対象の区間にスピルコードが多数存在していた．このスピルコードによって生じたスレッド間のデータ依存が並列性の抽出を阻害して

いる．そのため，今回の提案手法では問題に対応できず，並列性を抽出することができなかった．今後スピルコードによるスレッド間依存を除去することができれば，今以上の並列性を抽出できる可能性がある．

6. まとめ

マルチコアプロセッサにおけるシングルスレッド性能の向上のためにプログラムの並列化を容易にする研究が行われている．その研究の 1 つとして，我々は SKY を提案してきた．これまでの SKY では，除去することのできるスレッド間依存やスレッド生成の消極性により，TLP を十分に抽出することができていなかった．

これらの問題を解決するために，未受信の転送レジスタの自動転送，子の開始点の位置の最適化，死んでいる保存レジスタ値の明示的破壊によるスレッドをまたいだ定義-使用関係の除去，スレッドの予約によるオーバラップ実行できるスレッドの生成を提案した．

その結果，SPECfp2000 の 8 個のベンチマークにおいてシングルスレッド実行と比較して 4 コアのマルチスレッド実行で平均 2.74 倍の性能向上を達成し，提案手法を実装する前と比較して平均 1.47 倍の性能向上を達成した．

参考文献

[1] Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Reading, Massachusetts (1986).
[2] Kobayashi, R., Iwata, M., Ogawa, Y., Ando, H. and Shimada, T.: An On-Chip Multiprocessor Architecture with a Non-Blocking Synchronization Mechanism, *Proceedings of the 25th EUROMICRO Conference*, pp. 432–440 (1999).
[3] Ohsawa, T., Takagi, M., Kawahara, S. and Matsushita, S.: Pinot: Speculative Multi-threading Processor Architecture Exploiting Parallelism over a Wide Range of Granularities, *Proceedings of the 38th International Symposium on Microarchitecture*, pp. 81–92 (2005).
[4] Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K. and Chang, K.: The Case for a Single-Chip Multiprocessor, *Proceedings of the 7th International Conference on Architectual Support for Programming Language and Operating Systems*, pp. 2–11 (1996).
[5] OpenMP.org: OpenMP Specifications Version3.0, Japanese, `http://openmp.org/wp/openmp-specifications/`.
[6] SimpleScalar LLC: SimpleScalar LLC, `http://www.simplescalar.com/`.
[7] Smith, M. D., Horowitz, M. A. and Lam, M. S.: Efficient Superscalar Performance Through Boosting, *Proceedings of the 5th Intarnational Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 248–259 (1992).
[8] Sohi, G. S., Breach, S. E. and Vijaykumar, T. N.: Multiscalar Processors, *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 414–425 (1995).
[9] The Open MPI Development Team: Open MPI: Open Source High Performance Computing, `http://www.open-mpi.org/`.
[10] 岩原佑磨：マルチコアプロセッサ SKY におけるスタック分離に関する研究，名古屋大学大学院工学研究科博士課程 (前期課程) 修士学位論文 (2011).