社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2004－ARC－157 (21)
2004－HPC－ 97 (21)
2004／3／2

# プログラムにおける命令の並列性と逐次性について

佐藤 幸紀†, 鈴木 健一†, 中村 維男†
東北大学大学院情報科学研究科†

半導体のスケーリングが進み、CMOS トランジスタの密度が極めて高くなっている現在、豊富なハードウェアを活用した新たな並列処理を行うため、逐次プログラムの特徴を調べる必要がある。そこでプログラムの性質の尺度として命令間のデータ依存に着目し、命令トレースを入力として、命令間にデータ依存のある命令同士の距離の算出を行う。実験の結果、データ依存のある命令間の距離の分布は直後の命令において依存のある可能性が最も高いことが示される。得られたデータ依存の結果を基に統計的にデータ依存を解析することにより、プログラム中の命令の並列性と逐次性を調べる。これらの結果を基に、新たなアーキテクチャに向けて知見を述べ、その実現手法について議論する。

## Parallel and serial properties of instructions in programs

Yukinori SATO† and Ken-ichi Suzuki† and Tadao NAKAMURA†
†Graduate School of Information Sciences, Tohoku University

Considering today's advanced CMOS technology scaling that allows high transistor density, the features of programs should be investigated to perform parallel processing with a novel architecture using the enriched hardware resources. We measure the distance between data-dependent instructions in order to evaluate it. The results say that when the distance is one, the dependence is the highest. Analyzing the whole data obtained, parallelism and sequentiality of codes are further investigated. From these results, a suitable architecture and its implementation are described.

## 1 はじめに

半導体の微細加工技術が進むにつれて、マイクロプロセッサ 1 チップ上で利用可能なトランジスタ数が 10 億から 100 億個に達しようとしている [1]。これらのハードウェアを処理時間短縮のために有効活用するプロセッサが求められている。プロセッサの処理時間の短縮はプログラム中から並列性を抽出しプロセッサ上で並列実行することにより可能である。豊富なトランジスタ資源を有効活用するためには、単に並列実行可能なハードウェアを用意するだけではなく、プログラム中に内在する並列性の形式に適するハードウエアにより並列実行する必要がある。

現在主流となっているスーパースカラ方式はプログラム中の命令列から独立な命令を抽出し並列実行するという命令レベル並列性を用いている。しかしながら、命令レベル並列による処理速度向上はハードウエア的な困難さによる限界がある。十分な並列性を抽出するためには、命令が並列実行可能かどうかを検出する命令ウインドウの大きさを大きくしなければならない。巨大な命令ウインドウは命令ウインドウにおける遅延とハードウェア量を増加させる [2]。また、並列度の増大に従い演算に必要なオペランドの数が増加するため、レジスタにおいて必要な同時アクセス可能なポート数は増加する。レジスタのポート数の増加はレジスタの遅延とハードウェア量の両方を増加させる。よって、現状のスーパースカラ方式の並列度を向上させても、豊富なハードウェア資源を高速かつ有効に稼動させることは難しい。

プログラムに含まれる演算間のコミュニケーションを調べることはプログラムより得られる並列性の性質を得る上で重要である。演算間のコミュニケーションにおいて、大部分のオペランドは生成された直後に使用されるという報告がある [3]。また、SPEC2000 ベンチマークにおいて、平均 66%の命令が演算結果を直後の演算器に転送するバイパスロジックを利用してオペランドを得るという報告もある [4]。これらの結果は、一般に、プログラムは逐次的な性質が強いことを示唆している。

半導体技術の微細化が進むにつれて、これまで無視できた配線における遅延が回路内の遅延において大きな割合となる。配線の遅延時間は配線の長さに比例する。そこで、処理の時間的局所性を利用し、配線の長さをアーキテクチャレベルで長

距離配線を抑制するように制御することにより、配線遅延の影響を小さくすることが求められる。そこで、プログラムに含まれる演算間のコミュニケーションを解析することによりプログラムの局所性を抽出し、得られた局所性をプロセッサ上の空間的局所性に対応付ける。これより、近距離の局所的なバンド幅を向上させることにより不必要な長距離配線が抑制され、配線遅延の影響を受けにくいアーキテクチャが実現可能となる。

本論文では、プログラムの性質に適し、かつ、配線遅延の影響を受けにくいように処理に空間的な局所性を持たせた並列処理を行うため、プログラムの並列性と逐次性について調査を行う。実際のベンチマークプログラムの実行トレースを入力として命令間にデータ依存のある命令同士の距離を算出し、統計的にデータ依存性を解析することにより、プログラム中の命令の並列性と逐次性を調べる。その結果に基づいて、新たなアーキテクチャに向けた知見を述べる。そして、その1つの解が我々が提案している SHIFT Machine[5][6][7] であることを示す。

## 2 並列性と逐次性の解析手法

プログラムの実行は動的なデータフローグラフにより表現可能である。データフローグラフにおいてノードは命令を表し、ノード間のエッジは演算間のコミュニケーションを表す。前のノードの演算結果を利用する場合、そのノードは結果を利用するノードとエッジにより結合される。データフローグラフにおいて、実行可能なノードはデータ依存のみによって定まり、データの揃っているノードはプログラム中のどの部分であっても実行可能である。ハードウェア等の制約がない理想的な環境においては、データ依存が解決した命令はその直後に実行される。

現在主流のロードストアアーキテクチャにおいては、命令セットアーキテクチャにより定義されるレジスタにより命令間のコミュニケーションを行う。命令セットにより定義されるレジスタを用いて事前に依存関係が保証された実行コードを用いて、プログラムカウンタが指し示す命令を実行する。プログラムカウンタは逐次的に1命令ずつ命令をフェッチしていく。このため、データ依存が解決した場合においても、その直後に命令が実行されない。従って、依存の状況にかかわらず、命令の実行に際して命令間に時間的な距離を持つことになる。スーパースカラ方式において並列実行を試みる場合でも、命令ウインドウから外れている命令はデータ依存がない場合でも並列に実行されないため、時間的な距離を持つことになる。

プログラムにおけるデータ依存関係を定量的に評価するために、データ依存のある命令間の時間的な距離についての尺度を与える。命令間の実行における時間的な距離をその間に実行された命令数と定義する。この命令間の距離を用いて、データ依存のある命令間の距離をデータ依存距離とする。データ依存距離はレジスタに値が書き込まれてから、そのレジスタが読まれるまでに実行される命令数を測定することにより求めることが可能である。

あるプログラムの実行トレース中の各命令について $d_i$ を距離 $i$ 命令だけ後方に離れた命令が依存がある状態、$\bar{d}_i$ を距離 $i$ 命令だけ後方に離れた命令が依存のない状態とする。$\bar{d}_i$ は $d_i$ の余事象である。トレース中の全ての命令に対して依存の状態を求めることにより、プログラム中において距離 $i$ 命令だけ離れた全ての命令における依存がある事象の集合である $D_i$ と、依存のない事象の集合である $\bar{D}_i$ が求まる。$D_i$ はデータ依存距離の分布であり、$D_i$ を求めることにより様々な定量的な評価を行うことができる。距離 $i$ 命令だけ離れた命令の依存の状態を $S_i \in (D_i, \bar{D}_i)$ とすると、後続の $n$ 命令の依存の状況 $S_{1\cdots n}$ は $1 \leq i \leq n$ のとき (1) 式のように表される。

$$S_{1\cdots n} = S_1 \cap S_2 \cap \cdots \cap S_n \tag{1}$$

このとき、後続の $n$ 命令の依存の状況は $2^n$ 通りある。ある命令間のデータ依存距離の事象は異なる距離間で独立であると仮定すると、任意の後続の $n$ 命令の依存の状況が与えられたとき、全ての命令を通してその状況が起こる確率 $P_{S_{1\cdots n}}$ は乗法定理より (2) 式のように求められる。

$$P_{S_{1\cdots n}} = \prod_{i=1}^{n} P_{S_i} \tag{2}$$

式 (2) より後続の命令が特定の依存状況になる確率を求めることが可能となる。全ての依存の状況の確率を求めることにより、一度書き込まれたレジスタが次に値を書き込まれるまでに行われる読み込みの回数であるレジスタファンアウト [8] (degree

of use of register instances[3]) の期待値が算出される。レジスタファンアウトは命令の逐次性の指標となる。

スーパースカラ方式における命令ウインドウを想定し、並列性を抽出する場合を考える。並列実行を保証するためには実行する命令がウインドウ内の全ての先行する命令と依存のない必要がある。あるウインドウ内に依存関係の要素は ${}_nC_2$ 個だけあり、$2^{{}_nC_2}$ 通りの依存の状態がある。全ての命令を通してウインドウ内の $j$ 番目の命令とその命令から距離 $i$ 命令だけ離れた命令との依存関係の状態の事象の集合を $S_i^j$ とすると、命令ウインドウの大きさが $n$ の場合においてウインドウ内に起こりえる任意の状況 $W(n)$ は式 (3) のように表せる。

$$W(n) = \bigcap_{j=1}^{n-1} S_{n-j}^j \cap S_{n-j-1}^j \cap \cdots \cap S_1^j \quad (3)$$

命令ウインドウ内の位置を示す $j$ はデータ依存距離の分布と無関係であるため、式 (3) に $1 \leq i \leq n$ における $S_i \in (D_i, \bar{D}_i)$ となる確率を代入することにより、全ての命令を通してのその状況が起こる確率 $P_{W(n)}$ が求められる。ウインドウ内の依存関係の全ての場合を算出することにより、ウインドウ内で並列に実行可能な命令数の期待値を求めることが可能である。並列実行可能な命令数は得られる並列性の指標となる。

## 3 実験

プログラム中のデータ依存関係における並列性と逐次性を調査することは、アーキテクチャをどのように設計していくかという知見を得る上で非常に重要である。そこで、実際のベンチマークプログラムにおける命令トレースを入力として、命令間にデータ依存のある命令同士の距離を算出する。得られたデータ依存の結果を基に統計的な解析を行うことにより、プログラム中の命令の並列性と逐次性について調べる。

動的なデータ真依存の起こった回数を調べるために、サイクルレベルシミュレータである SimpleScalar ツールセット version3.0 [9] を改良することにより、プログラムの命令トレースを採取した。命令セットとして MIPS 互換の PISA を使用し、gcc-2.6.3 を用いてシミュレータへの入力となる実行コードを生成した。ベンチマークとして SPEC

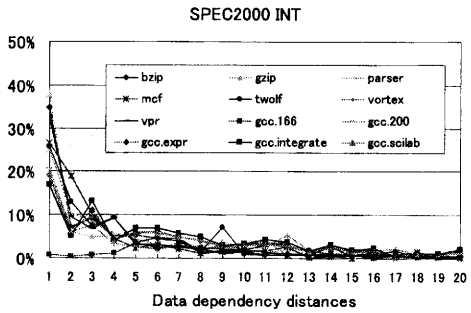


図 1: SPEC2000int におけるデータ依存距離

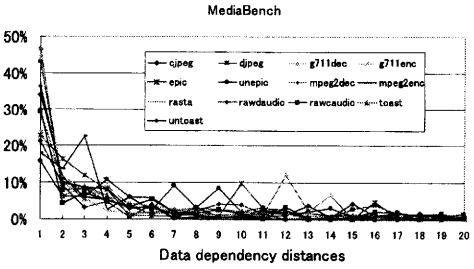


図 2: MediaBench におけるデータ依存距離

CPU2000 INT、MediaBench[10] の 2 種類を用いた。SPEC CPU2000 INT はそれぞれのベンチマークが複雑な制御フローを持つ。MadiaBench はデータ並列性を多く含むメディア処理の分野で広く使用されているベンチマークである。各ベンチマークセットにおいてシミュレーションの環境の制約から全てのベンチマークを実験することができなかった。各ベンチマークにおいてリファレンスデータセットを用いて終了するまで実行を行った。

図 1 は SPEC2000int の各ベンチマークを用いて、プログラム中の全ての命令における後続 20 命令間のデータ依存距離の割合の分布を測定した結果を示す。横軸はデータ依存距離を、縦軸は実行した全ての命令数と依存のある命令の割合を示す。gcc.166 を除いてデータ依存距離が 1 であるとき最もデータ依存が頻繁に起こっていることが分かる。直後の 3 命令においては、依存のある可能性の累積は平均して 40%に達する。また、命令間の距離が離れるほど、依存の割合が減少していることもわかる。gcc において複雑な int プログラムである 166.i を入力データとして利用した場合に、特異なデータ依存距離の傾向を示すが、gcc のその他の入力データの場合、他のベンチマークと同様の傾向となった。これは、レジスタへの書き込み

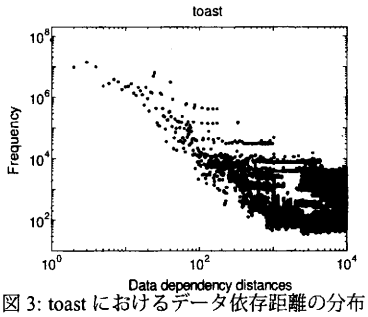

図 3: toast におけるデータ依存距離の分布

およびレジスタからの読み込みは動的なプログラムのパスに依存するためである。gcc における入力データ 166.i はレジスタに値が書き込まれても全く使用されない場合の極端な例を示している。

図 2 は MediaBench の各ベンチマークを用いて、プログラム中の全ての命令における後続 20 命令間のデータ依存距離の割合の分布を示す。いくつかのベンチマークはデータ依存距離が 1 のときの割合が 40%を越える。音声ファイルの形式を変換するアプリケーションである toast はデータ依存距離が 1 である割合が 47%を示し、本実験で最も高い結果となった。また、直後の 3 命令において依存のある可能性の累積は SPEC2000int よりも高い 50%に達する。このことより、MediaBench の方が隣接する命令との依存関係が強い傾向があることがいえる。また、SPEC2000int の場合と同様に命令間の距離が離れるほど、依存の割合が減少していることもわかる。

図 3 にプログラム中の全ての命令に対する後続 10000 命令間におけるデータ依存距離の分布の測定結果を示す。データ依存距離が 1 である割合が今回実験した中で最も高いベンチマークである MediaBench の toast を代表として選んだ。図 3 において横軸はデータ依存距離を対数スケールにより 10000 命令まで表し、縦軸は依存のある事象の出現頻度を対数スケールにより表している。図 3 は図 1、2 と比べて非常に長いデータ依存距離を測定した結果である。命令間の距離が非常に長い場合でも、命令間の距離が近いほど依存のある頻度は高く、命令間の距離が離れるほど依存のある頻度が減少していることが分かる。これは、コンパイラにおいてなるべく命令間の距離が近い場合の依存関係の解消のための通信にレジスタを割り当てているからと考えられる。依存がある状態の出現頻度の高い事象は命令間距離が離れるに従って少なくなり、命令間の距離が 20 を越えると 100 分の

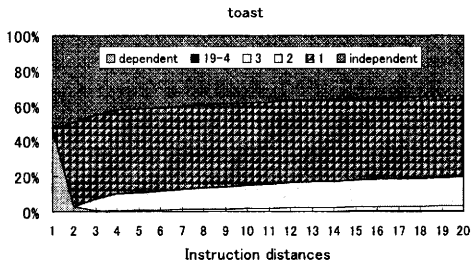

図 4: toast における後続命令との依存関係

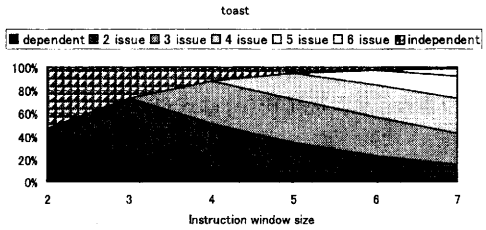

図 5: toast における命令ウインドウ内の並列性

1 以下となることが分かる。しかしながら、プログラムを動的に実行する際のループや関数呼び出し等を間に持つレジスタは、長く値を保持するために、命令間の距離が離れた場合においても、なかなか依存がある頻度が収束しない。

図 4 は MediaBench の toast におけるある距離だけ離れた命令間における依存関係の分布を後続 20 命令まで式 (2) を基に求めた結果である。横軸は命令間の距離を示し、縦軸は依存のある命令数に対応する割合である。グラフの上部に積み上げられたものほど依存のない事象を、下部にあるものほど依存がある事象であることを示す。大部分の命令は依存がないか 1 命令のみと依存があるということが分かる。これは、大部分の命令が演算の途中経過をレジスタに保持しすぐ演算の入力として利用するためである。また、後続 20 命令間においては多数の依存がある可能性は低いことが分かる。これは、多数回読み込みが行われるレジスタはデータ依存距離が離れた場所に多く存在する傾向があるためである。

図 5 は MediaBench の toast におけるウインドウ内の同時実行が可能な命令の数の割合の分布をウインドウの大きさが 7 となるまで式 (3) に基づき求めた結果である。式 (3) により全検索を行い解析することは多項式以上の組合わせの解析が必要となるためため、ウインドウの大きさは 7 までとした。横軸はウインドウの大きさを示し、縦軸は

表 1: レジスタファンアウトと並列実行可能な命令数

| Mediabench | register fanout | ILP | SPEC2000int | register fanout | ILP |
|---|---|---|---|---|---|
| cjpeg | 0.82 | 4.30 | bzip | 0.86 | 4.29 |
| djpeg | 0.97 | 3.90 | gzip | 0.82 | 4.27 |
| epic | 0.87 | 4.55 | mcf | 0.81 | 4.36 |
| g711dec | 0.75 | 3.90 | parser | 0.77 | 4.70 |
| g711enc | 0.75 | 3.90 | twolf | 0.83 | 4.55 |
| mpeg2dec | 0.93 | 4.47 | vortex | 0.69 | 5.39 |
| mpeg2enc | 0.85 | 4.59 | vpr | 0.78 | 4.36 |
| rasta | 0.78 | 4.72 | gcc.166 | 0.32 | 6.78 |
| rawcaudio | 0.73 | 4.72 | gcc.200 | 0.82 | 5.03 |
| rawdaudio | 0.57 | 5.13 | gcc.expr | 0.85 | 5.10 |
| toast | 0.89 | 3.73 | gcc.integrate | 0.90 | 5.14 |
| unepic | 0.83 | 5.26 | gcc.scilab | 0.82 | 5.09 |
| untoast | 0.73 | 4.13 | | | |
| average | 0.81 | 4.41 | average | 0.77 | 4.92 |

並列実行可能な命令数であり、グラフの上部に積み上げられたものほど並列実行可能な命令数が多く、下部にあるものほど少ないことを示す。図 5 より全てが並列に実行可能であるものや、全てが並列実行できないものは少数であり、その中間的な場合が多いことが分かる。

表 1 は SPEC2000int および MediaBench において式 (2) により求めたプログラム実行中の 20 命令間のレジスタファンアウトの期待値と、式 (3) により求めたプログラム実行中のウインドウの大きさが 7 のときの並列実行可能な命令数の期待値を示す。各ベンチマークにおいてレジスタファンアウトはほぼ 1 であることが分かる。これにより、20 命令間という近い距離にある多くのデータ依存は 1 度のみ使われることがいえる。限られた区間におけるレジスタファンアウトが高いということは、レジスタに値が書きこまれてからすぐに値が読み込まれるということであり、依存を順次解決していく逐次的な処理であるといえる。SPEC2000int と MediaBench を比べた場合、平均してわずかに MediaBench の方がレジスタファンアウトが高いので、より逐次的な処理であるといえる。

表 1 において、ILP が示すウインドウの大きさが 7 のときの並列実行可能な命令数は 3 から 6 に分布している。並列実行可能な命令数は、データ依存距離の分布において距離 1 に依存のある割合が高いものやレジスタファンアウトが高いものほど少ない傾向がある。これは、プログラムの逐次性の傾向が強まるほど、近距離で得られる命令レベル並列性が小さくなるため並列性を得ることが困難であることを示している。

## 4 考察

この章では、前章までに調査したプログラムの依存の性質を基に将来のアーキテクチャに必要とされる要件を考察する。プログラム中の演算間においては並列性あるいは逐次性の性質を持つ。並列性と逐次性のどちらかに着目することにより、更なる並列処理が可能となると考えられる。並列性に着目し、依存のない命令を近接する命令に配置することにより命令レベル並列性を得ることが可能である。しかし、命令間の距離が近い命令は依存のある可能性が高いため、それらをスケジューリングし並列性を得ることは高い並列性を得ようとするほど困難になる。

命令間の距離が遠い命令は依存のない可能性が高いということに着目すると、命令間の距離が遠くに位置する依存のない命令間において並列性を抽出することが可能となる。これらは、既存のスレッドレベル並列性、ループレベル並列性といった、命令レベル並列性よりも大きな粒度で得られる並列性に対応する。また、命令間の距離が遠くなるように依存のない命令の配置を行うことは、依存のある命令をデータ依存距離が近くなるように配置することになる。このとき、隣接する命令の集合は依存のある命令列となり、プログラム中の演算間におけるコミュニケーションは非常に逐次性の強いものとなる。

このような逐次性の性質が強い場合に適するアーキテクチャの方式として MISD (Multiple Instruction stream - Single Data stream)[11] がある。MISD 処理において命令流は実行される命令列を意味し、データ流は命令流により使用される入力と一時的な結果を含む一連のデータである。MISD 方式により利用可能となる空間的局所性は、演算間のコミュニケーションと中間的な結果を保持するコストを最小化する。また、MISD 方式において空間的かつ時間的並列性を利用することにより高い実行バンド幅をが得られる。さらに、ハードウェアの持つ空間的局所性は、配線遅延を隠蔽することが可能である。

MISD 処理モデルに基づき、我々は SHIFT Atchitecture とその実装である SHIFT Machine を提案している [5][6][7]。図 6 に SHIFT Machine の概念を示す。SHIFT Machine は 1 次元に接続された PE の列により構成される。各 PE は 1 サイクルに 1 つの命令を実行しその結果を隣接する次の PE に転送する。SHIFT Machine は 1 つの PE を 1 つのステージとし、パイプライン的に処理が進行するため、処理に空間的局所性を利用することが可能である。SHIFT Machine は図 6 に示すように多数のスレッドを多重化することにより並列処理を行う。

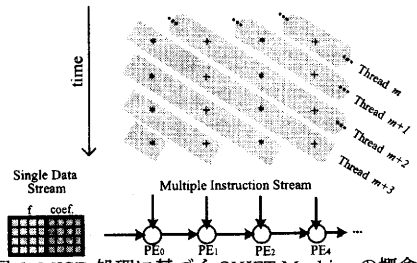

図 6: MISD 処理に基づく SHIFT Machine の概念図

ここで、スレッドとは連続する依存のある命令の列により構成され、スレッド間は互いに独立に実行することが可能なものと定義する（文献 [12] においては strand と定義している）。SHIFT Machine の性能評価に関しては [5][6] にある。

## 5 結論

本論文においてプログラムにおける命令間のデータ依存の性質を調査した。その結果、直後の命令に依存がある可能性は全実行命令数のおおよそ 20% から 50%に達し、直後 3 命令においては依存のある可能性の累積が 40%から 50%に達した。また、データ依存距離の分布からレジスタの値が後続の命令でどの程度使われるかの期待値と同時実行可能な命令数の期待値を算出したところ、MediaBench は SPEC2000int よりも逐次的な処理が多く、命令レベル並列性を得にくいことが分かった。

プログラムの並列性と逐次性をの性質を踏まえて、多命令発行が想定される将来のプロセッサにおいて適するアーキテクチャの検討を行った。プログラムの並列性と逐次性の性質に適するアーキテクチャの方式として MISD 処理モデルがあり、この MISD 処理モデルに基づき、我々は SHIFT Atchitecture とその実装である SHIFT Machine を提案している。

今後の課題として、本研究で得たプログラムの並列性と逐次性の性質についての結果を SHIFT Machine は有効に利用しているかということを他の方式と比べて定量的に評価すること、コンパイラ等による具体的なスケジューリング手法の提案を行っていくことが挙げられる。

## 謝辞

本論文を作成するに当り、有益な助言を頂いた東北大学大学院情報科学研究科佐野健太郎助手に感謝する。

## 参考文献

[1] Alan Allan, et al. 2001 technology roadmap for semiconductors. *IEEE Computer*, Vol. 35, No. 1, pp. 42–53, Jan 2002.

[2] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th annual international symposium on Computer architecture*, pp. 206–218. ACM Press, 1997.

[3] Manoj Franklin and Gurindar S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 25th annual international symposium on Microarchitecture*, pp. 236–245. IEEE Computer Society Press, 1992.

[4] Il Park, Michael D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pp. 171–182. IEEE Computer Society Press, 2002.

[5] 中村維男, 佐藤幸紀. 配線遅延を考慮したマルチスレッド方式アーキテクチャSHIFT Machine の提案. 情報処理学会研究報告 2002-ARC-148, pp. 31–36, 2002.

[6] Clecio D. Lima, Kentaro Sano, Hiroaki Kobayashi, Tadao Nakamura, and Michael J. Flynn. A technology-scalable multithreaded architecture. *Proc. of the 13th Symp. on Computer Architecture and High Performance Computing*, pp. 82–89, 2001.

[7] Tadao Nakamura. Toward architecting and designing novel computers. In *Proceedings of 8th Asia-Pacific Conference, ACSAC 2003*, pp. 8–13. Springer, 2003.

[8] K. Sankaralingam, R. Nagarajan, S.W. Keckler, and D.C. Burger. A technology-scalable architecture for fast clocks and high ILP. In *5th Workshop on the Interaction Between Compilers and Computer Architectures (INTERACT-5)*, Jan. 2001.

[9] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, Vol. 25, No. 3, pp. 13–25, 1997.

[10] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pp. 330–335. IEEE Computer Society, 1997.

[11] Michael J. Flynn. Very high-speed computing systems. *IEEE Proceedings of the IEEE*, pp. 1901–1999, December 1966.

[12] Ho-Seop Kim and James E. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *Proceedings of the 29th annual international symposium on Computer architecture*, pp. 71–81. IEEE Computer Society, 2002.