# キャッシュミス情報を利用する省電力命令スケジューリング

千代延, 昭宏
九州工業大学情報工学部知能情報工学科

佐藤, 寿倫
九州大学システムLSI 研究センター

https://doi.org/10.15017/6352

出版情報：電子情報通信学会論文誌. J89-D (12), pp.2590-2601, 2006-12. 電子情報通信学会
バージョン：
権利関係：

論　文

# キャッシュミス情報を利用する省電力命令スケジューリング

千代延昭宏[†]　　佐藤　寿倫[††]

An Energy-Efficient Instruction Scheduling Technique Exploiting Cache Miss Information

Akihiro CHIYONOBU[†] and Toshinori SATO[††]

**あらまし**　近年のマイクロプロセッサには処理性能を維持しつつ消費電力を削減することが求められている．本稿では，その削減のためマイクロプロセッサとメインメモリの間に存在する動作速度の隔たりを利用する．クリティカルパス情報に基づきレイテンシの不均一な演算器を使い分けるアーキテクチャにおいて，主記憶アクセス中に実行される命令をノンクリティカルとみなし，全て低速かつ低消費電力な演算器で実行させることを提案する．シミュレーションによりエネルギー遅延二乗積の評価を行った結果，全ての演算器が高速な場合と比較して平均 27.3%，クリティカルパス情報のみを利用する場合と比較して平均 2%削減できることがわかった．

**キーワード**　メモリウォール，クリティカルパス，低消費電力化，マイクロプロセッサ

## 1. はじめに

マイクロプロセッサの処理性能は様々な技術を用いることで改善されている．一般にプロセッサの処理性能を向上させるには，いかに大量の演算器で大量の命令を同時に実行させるかが鍵となる．例えば，投機実行は分岐命令の分岐先を事前に予測し，そのパスを投機的に実行する．そうすることで，演算器がアイドル状態になることを防ぐ．しかし，これまで処理性能を向上する際に注目されていなかったプロセッサの消費する電力が，今日では新たな問題となっている．例えば分岐先の予測に失敗すると，投機実行をしたことで無駄なエネルギーを消費したことになる．また近年のマイクロプロセッサは，大量の命令を同時に実行させるために命令ウインドウやキャッシュメモリの大規模化が進んでいる．これらもマイクロプロセッサの消費電力を増加させている．

現在多くのマイクロプロセッサは，実行しようとする命令を演算器に対してアウト・オブ・オーダに発行することで，プログラムの実行時間を少なくしている．プログラムの実行時間は，プロセッサの処理性能と実行している命令間の依存関係によって決まる．命令間の依存関係を結んだ鎖のうち，最も実行に時間を要するパスをクリティカルパスと呼ぶ．それはプログラムの実行時間を決定する命令列である [16]．図 1 に命令列中に現れるクリティカルパスを表すデータフローグラフ (Data Flow Graph: DFG) の例を示す．図 1 において矢印は命令間の依存関係を表している．つまり，依存している命令は矢印の始点にある依存先の命令実行が終了しない限り実行できない．DFG の各枝の重みは命令のレイテンシである．全ての命令のレイテンシが 1 サイクルであるとすると，最も長いパスである命令 `I:0->I:3->I:6->I:7->I:8` を結んだパスがクリティカルパスとなる．

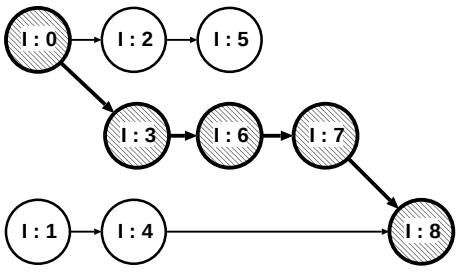



図 1　クリティカルパス
Fig. 1　Critical Path

† 九州工業大学 情報工学部 知能情報工学科，福岡県
Kyushu Institute of Technology, 680–4, Kawazu, Iizuka, 820–8502 Japan
†† 九州大学 システム LSI 研究センター，福岡県
Kyushu University, 3–8–33–3F, Momochihama, Sawara–ku, Fukuoka, 814–0001 Japan

今日のマイクロプロセッサとメインメモリとの間には，メモリウォールと呼ばれる動作速度の隔たりが存在する [15]．この動作速度の差がどのくらい広がっているかを図 2 に示す [6]．図において，横軸は年を縦軸は性能向上をログスケールで表している．グラフからわかるようにマイクロプロセッサの処理性能は年に 55%づつ向上しているが，メインメモリの性能は年に 7%づつしか向上していない．この性能の差がマイクロプロセッサの処理性能を抑制してしまう．例えばロード命令がキャッシュメモリにヒットした場合，要求されたデータは直ちにプロセッサに送られてくる．しかしミスした場合，メインメモリへのアクセスが起こるためプロセッサはストールしてしまう．我々はこのような場合にメモリアクセス命令がクリティカルパスへ悪影響を与えてしまうと考えている．

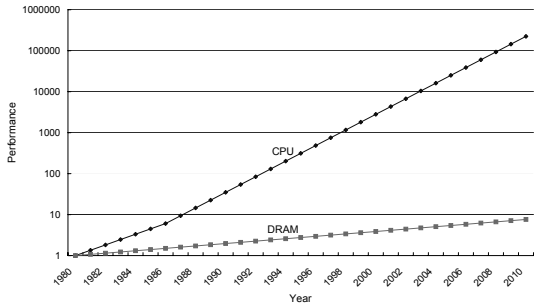



図 2　マイクロプロセッサ–メインメモリ間の性能格差
Fig. 2　Performance Gap Between Memory and Microprocessor [6]

マイクロプロセッサはリオーダバッファのエントリがいっぱいになるまで，フェッチユニットやデコードユニットが命令供給を続ける．このためメモリアクセスが起こったとき，マイクロプロセッサ中には大量の実行中の命令が存在する．DFG 上にロード命令がある場合を考える．ロード命令がキャッシュメモリにヒットする場合はクリティカルパスは変わらない．しかしミスした場合は DFG の重みが変わってしまうため，クリティカルパスが変わってしまう可能性がある．キャッシュミスしたロード命令の実行が完了されるまで，そのロード命令に依存する命令は実行を開始できないからである．キャッシュミスが起きている間に実行が可能な命令は，当該ミスを起こしたロード命令に依存していない命令だけである．プロセッサ–メインメモリ間の性能格差が一層激しくなる今後のマイクロプロセッサでは，キャッシュミスしたロード命令に依存する命令はクリティカルパス上の命令になる可能性が高い．図 3 に例を示す．図において，`I:0` と `I:1` がロード命令であるとする．図 1 同様全ての命令のレイテンシは 1 サイクルであるが, キャッシュミスが起こった場合は DFG の重みが増すものとする．`I:0` と `I:1` の両方がキャッシュメモリにヒットするならば，図中のクリティカルパスは図 1 と同様に `I:0->I:3->I:6->I:7->I:8` となる．しかし `I:1` がミスした場合は，`I:1->I:4` で重みの変化が起こるため，クリティカルパスは `I:1->I:4->I:8` へと変わる．

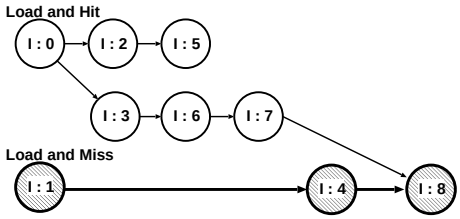



図 3　クリティカルパスの変化
Fig. 3　Exchange of the Critical Path

本稿で我々は，メモリアクセスを生じさせるキャッシュミスとクリティカルパスの関係を考慮した新たな省電力命令スケジューリング手法を提案する．以下では，本稿で用いられるキャッシュミスはキャッシュ階層の最下位で起こったキャッシュミスを指すこととする．本論文の構成は以下の通りである．次章で省電力命令スケジューリング手法について説明した後，キャッシュミス情報を利用する命令スケジューリング手法を提案する．3 章で評価に用いた環境と評価条件について説明し，4 章で評価結果を示す．5 章で関連研究について説明し，6 章でまとめと今後の課題について述べる．

## 2. 省電力命令スケジューリング

我々はマイクロプロセッサの処理性能を維持しつつ省電力化を実現するために，クリティカルパス情報と不均質な演算器を利用してきた [19]．具体的には，プログラムの実行時間に影響を与えるクリティカルパス上の命令を高速な演算器で実行させ，クリティカルパス上にない命令を低速な演算器で実行させる．このようにスケジューリングすることで，プログラム全体の実行時間を増加させることのない省電力化を目指している．本章では，我々のクリティカルパス情報を用いた省電力スケジューリングとクリティカルパスを特定する方法について述べた後，キャッシュミス情報を利用する省電力命令スケジューリングについて説明する．

### 2.1 クリティカルパス情報を用いた省電力命令スケジューリング

我々は実行サイクル数を増やさずにプロセッサの低消費電力化を実現するため，プログラムの実行時間を決めるクリティカルパスの特性に着目した [19]．クリティカルパス上の命令はプログラムの実行時間を決定するが，クリティカルパス上にない命令はクリティカルパス上の命令の開始を遅らせない限りプログラムの実行速度に影響を与えない．この点に着目してプロセッサの持つ演算器の構成を変更する．現在多くのプロセッサは演算器を複数持っているが，これらの演算器として消費電力の異なるものを用意する．例えば，一部の演算器の電源電圧を下げる．電源電圧を下げることによってレイテンシは増加するが電力消費を下げることが可能になる．このレイテンシと電源電圧の異なる演算器を持つプロセッサに対し，クリティカルパス上の命令かどうかという情報に基づいて，実行時間に影響を与えるクリティカルパス上の命令は高速かつ電力消費の大きな演算器で実行し，クリティカルパス上にない命令は低速かつ電力消費の小さな演算器で実行するように命令をスケジューリングする．この様子を図 4 に示す．

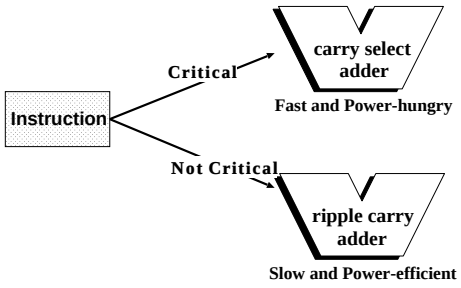



図 4　クリティカルパス情報を用いた省電力スケジューリング
Fig. 4　The Energy Efficient Instruction Scheduling Exploiting Critical Path Information

アーキテクチャの具体的な変更例を図 4 に示す．桁上げ先見加算器 (carry–lookahead adder) や桁上げ保存加算器 (carry save adder)，桁上げ選択加算器 (carry select adder) などで構成されていた，高速ではあるが電力消費の大きな演算器を順次桁上げ加算器 (ripple carry adder) を用いた低速ではあるが省電力な演算器に構成を変更する．そして，クリティカルパス上の命令かどうかを表す情報を命令に付加する．その情報に従ってクリティカルパス上の命令は前者に発行し，クリティカルパス上にない命令を後者に発行する．

以上の様にして，プログラム全体の実行時間を変化させることなくプロセッサの消費電力を下げることが可能となり，電源電圧を下げることによる動作速度低下という弊害の影響を隠蔽することが可能となる．クリティカルパス情報を利用して電力消費を削減しようとする試みは他にもあるが [12], [13]，いずれもエネルギー消費量ではなくピークの電力値を削減することが目的であり，我々の目的とは異なる．

### 2.2 パス情報テーブル

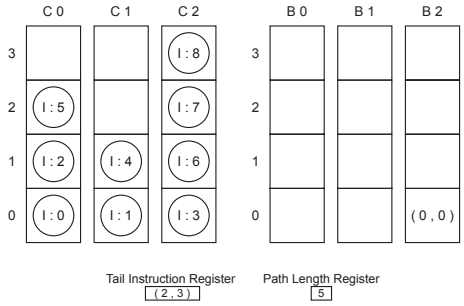



図 5　パス情報テーブル
Fig. 5　Path Information Table

本稿では，各命令がクリティカルパス上の命令かどうかを特定するために小林らが提案しているパス情報テーブル (Path Information Table: PIT) [16] を用いる．クリティカルパスを特定する機構として，PIT の他にクリティカルパス予測器 [5], [14], [19] がある．しかしクリティカルパス予測器は PIT に比べクリティカルパス予測の精度が劣ることが明らかになっているため [18]，今回はこれを用いない．

PIT を図 5 に示す．PIT は 2 つのテーブルから構成されている．また各テーブルは複数の FIFO キューを持っている．図において，C0，C1，C2，そして，B0，B1，B2 がそれぞれ FIFO キューである．左側のテーブルは命令間の依存関係を保持する．図 5 は図 1 中の命令を保存している．FIFO キューの各エントリは命令ウインドウの対応する命令へのポインタを持っている．同一 FIFO キューに依存する命令の情報が順に挿入されていく．DFG が分岐した場合や依存情報を保存する FIFO が飽和した場合の分岐情報は，右側のテーブルで保持される．例えば，図 5 において右側のテーブル内の FIFO キュー B2 の (0, 0) は，左側のテーブルにおける I:3 が FIFO キュー C0 の I:0 に依存していることを示している．B0 と B1 にはポインタがな

い．これは対応している FIFO キュー C0，C1 のパスが他のパスには依存していないことを示している．各 FIFO キューのエントリが保持するポインタは対応する命令が発行されるまで保持される．また，PIT は末尾命令レジスタ (Tail Instruction Register) と最長パスレジスタ (Path Length Register) という二つのレジスタを持っている．末尾命令レジスタは最長パスの末尾命令へのポインタがどのエントリに保持されているかの情報を持つ．最長パスレジスタは文字通り最長パスの長さ情報を持つ．各レジスタの情報は新たに命令がディスパッチされる度に更新される．PIT が保持するこれらの情報を利用することで，クリティカルパスを特定できる．

PIT は DFG の最長パスを見つけることでクリティカルパスを特定する．しかし，PIT は全てノードの重みを 1 に固定している．このため PIT の作る DFG はロード命令がキャッシュにヒットしてもミスを起こしても変化しない．以上のことから，PIT は 1 章で説明したキャッシュミスによるクリティカルパスの変化に対応できないといえる．同様に PIT を用いて不均質な演算器を使い分けた場合，発行される演算器のレイテンシの影響を受けてクリティカルパス上の命令ではなかった命令がクリティカルパス上の命令になってしまうことがある．しかし上述したように，PIT の方がクリティカルパス予測器よりもクリティカルパスの特定精度が高い．このため，PIT を用いて不均質な演算器を使い分けることが現状では最適である．

### 2.3 動的演算器ゲーティング

我々は動的演算器ゲーティング (Dynamic Functional Units Gating: DFUG) と名付けた省電力な命令スケジューリング手法を提案する．DFUG はマイクロプロセッサとメインメモリ間の速度差を利用する．対象とするマイクロプロセッサは 2.1 節で説明したように，クリティカルパス情報を利用して不均質な演算器を使い分けているとする．つまりクリティカルパス上の命令を高速かつ消費電力の大きな演算器で実行し，クリティカルパス上にない命令は低速かつ消費電力の小さな演算器で実行している．DFUG はこの命令スケジューリング手法を改良して命令をより省電力に実行する．

DFUG は二種類のモードをキャッシュミス情報によって切り替える．クリティカルパス情報によって高速・低速な演算器を使い分けるノーマルモードと，低速な演算器しか使わない省電力モードである．キャッシュミス中に実行される命令は，ミスを引き起こした命令には依存していない．よって，キャッシュミスが解決された後はノンクリティカルな命令になることが予想される．DFUG ではこの現象を利用する．ノーマルモードでプログラムの実行を開始する．ノーマルモードで実行中にキャッシュミスが起きると，それをトリガにしてモードが省電力モードに切り替わる．省電力モードでプログラムを実行中にキャッシュミスが解決されると，再びノーマルモードに戻る．この様子を図 6 に示す．このように二つのモードをキャッシュミス情報を用いて切り替えることで，処理性能に影響を与えることなくより省電力なプログラム実行が可能になると考えられる．

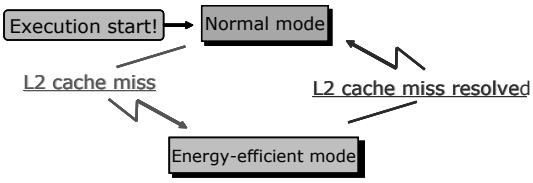



図 6　モードの切り替わり
Fig. 6　Mode Transition

現在のマイクロプロセッサはキャッシュミス中でもストールしないで命令の実行を続ける．ノン・ブロッキングキャッシュはキャッシュミス中でも他のロード命令がキャッシュメモリへアクセスすることを可能にする [4]．このような場合，キャッシュミスが重なって起こることも考えられる．キャッシュミス中に新たなキャッシュミスが生じた場合，新たなミスは無視するという方式と全てのキャッシュミスが解決されるまで省電力モードを延長するという方式が考えられる．我々はキャッシュミスが重なって起こった場合に，全てのキャッシュミスが解決されるまで省電力モードを延長する手法を拡張演算器ゲーティング (Enhanced Dynamic Functional Units Gating: E-DFUG) として DFUG と区別する．

E-DFUG と DFUG の違いを図 7 に示す．図 7(a) は DFGU を図 7(b) は E-DFUG をそれぞれ表す．図中の横軸は時間を示し，右側に行くほど時間が経過したことを表す．まず図 7(a) の DFUG について説明する．キャッシュミスが起こるまではノーマルモードで動作しており，クリティカルパス情報によって高速，低速な演算器を使い分けている．キャッシュミスが発生すると省電力モードに切り替わる．この間実行される命令は全て低速な演算器で実行される．高速な演算器は使用されない．図 7(a) で省電力モードは，モー

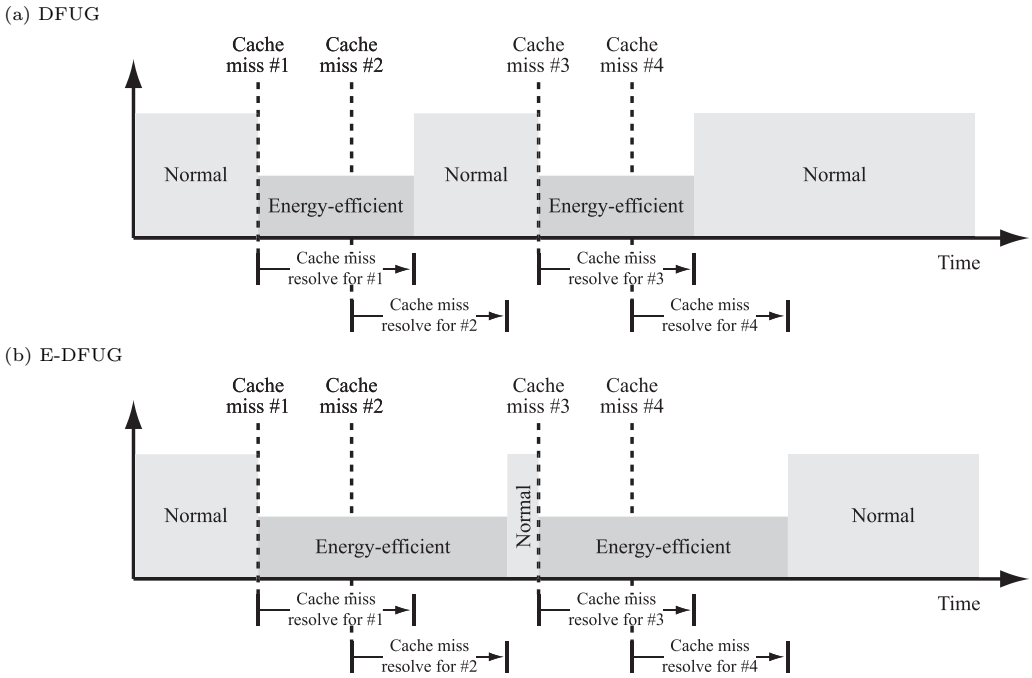



図 7　E-DFUG と DFUG の違い
Fig. 7　The difference between E-DFUG and DFUG

表 1　プロセッサ構成
Table 1　Processor Configuration

| | |
|---|---|
| Fetch Bandwidth | 8 instructions |
| Branch Predictor | 1K-set 4-way set-associative BTB, 4K-entry 8-history-length gshare predictor,<br>64-entry return address stack, 6-cycle miss penalty, updated at commit stage |
| Insn. Windows | 64-entry instruction queue, 32-entry load/store queue |
| Issue Width | 8 instructions |
| Commit Width | 8 instructions |
| Functional Units<br>Latency<br>(total/issue) | 6 Int, 4 FP, 2 Ld,St<br>fast iALU 1/1, slow iALU 2/1, iMUL 3/1, iDIV 20/19, fADD 2/1, fCMP 2/1, fCVT 2/1, fMUL 4/1,<br>fDIV 12/12, fSQRT 24/24, |
| Insn. Cache<br>Data Cache | 64KB, 2-way, 64B blocks, 1-cycle latency<br>64KB, 2-way, 64B blocks, non-blocking load, hit under miss, 3-cycle latency |
| L2 Cache | unified, 1MB, 8-way, 64B blocks, 11-cycle latency |
| Bus Width | 8B |
| Main Memory Latency | 130-cycle latency |

ドをノーマルモードから省電力モードに切り替えたキャッシュミス (図中 Cache miss #1，#3) が解決されるまで続く．省電力モードで実行中に起こった新たなキャッシュミス (図中 Cache miss #2，#4) はモード切り替えに影響を与えない．省電力モードからノーマルモードへ戻るのは**ノーマルモード → 省電力モード**のモード切り替えを起こしたキャッシュミス (図中 Cache miss resoleve #1，#3) が解決したときである．

次に図 7(b) の E-DFUG の場合について説明する．DFUG の場合と同様にキャッシュミスが起こるまではノーマルモードで実行をしている．キャッシュミスが起こると低速な演算器のみを使う省電力モードへ切り替わる．既に述べたように E-DFUG では省電力モード中に起きた全てのキャッシュミスが解決されるまでモードの切り替えのタイミングを延長する．このため省電力モードからの復帰が起こるのは，**ノーマルモード → 省電力モード**のモード切り替えを起こしたキャッシュミスの解決時ではなく，連続するキャッシュミス

中の最後に起きたキャッシュミスの解決時である．図 7(a) では無視されていた Cache miss #2, #4 が解決されるまでの時間分，省電力モードが延長されていることがわかる．

このように E-DFUG では重複するキャッシュミスを無視しない．このため図 7 に示したように省電力モードになる期間が増加し，E-DFUG は実行時間を延長することなく DFUG よりもより省電力にプログラム実行を行えると予想される．

DFUG と D-DFUG は不均質な演算器を使い分けているプロセッサに適用される．そのため，省電力モード中に低速な演算器のみに命令を発行させる機構は元々備えている．また，キャッシュミスとその解決を監視する機構も複数のキャッシュミスを許可するプロセッサには備わっている．以上のことから，DFUG と E-DFUG はわずかなハードウェア的な拡張だけで実装が可能である．

## 3. 評価方法

シミュレーションに用いたプロセッサモデルとベンチマークプログラムについて説明し，どのような環境で DFUG と E-DFUG を評価したかを述べる．用いたシミュレータは memory extension [2] を施した SimpleScalar Tool Set [1] である．前章で説明した DFUG，E-DFUG をシミュレータに実装して評価をおこなった．プロセッサ構成を表 1 に示す．プロセッサは演算器を 6 個持つ．事前に全ての高速・低速な演算器の組み合わせを評価した結果，高速な演算器は 2 つ，低速な演算器は 4 つという組み合わせがエネルギー利用効率の点で最適な構成であることが分かっている [3]．このため，本稿でもこの構成を用いて評価を行う．高速な演算器と低速な演算器のレイテンシはそれぞれ 2 : 1 となるようにする．低速な演算器は，高速な演算器をパイプライン動作させるものに相当し，レイテンシは 2 に増加するが，スループットは 1 のままである．これらの演算器の電源電圧は文献 [7] で紹介されているものから二種類の組み合わせを用いる．この組み合わせを表 2 に示す．プロセッサのリーク電流については今回は考慮しない．今後の課題とする．

クリティカルパス情報を用いて使い分ける演算器として浮動小数点ユニット (FPU) を選択することも考えられるが，そうした場合，低速な場合のレイテンシが性能へ与えるインパクトが大きくなりすぎると判断した．このことと実装の容易さから，本稿では提案するアーキテクチャを整数演算器に適用して評価する．FPU への応用は将来の課題とする．

2.2 節で述べたように，本稿ではクリティカルパス上の命令かどうか判断するために PIT を用いる．命令セットは MIPS R10000 を拡張した SimpleScalar/PISA である．使用したベンチマークプログラムは SPEC 2000 CINT の中から選んだ 7 本で，それぞれの入力は表 3 に示す通りである．いずれのプログラムもはじめの 10 億命令をスキップし，続く 5 億命令をシミュレーションした．評価にはプロセッサの処理性能とエネルギー遅延二乗積 (Energy-Delay Square Product: $ED^2P$) [11] を用いる．$ED^2P$ はパワー削減による遅延の増加と省電力効果のトレードオフを定量的に評価するために用いられる指標である．電圧制御を用いて省電力化を試みる場合の評価に適している [11]．$ED^2P$ は，高速・低速な演算器がそれぞれ何回ずつ使用されたかをカウントし，その回数とそれぞれの演算器を使用した際のエネルギー遅延を乗して求めることとした．

表 2　供給電圧と動作周波数の組み合わせ [7]
Table 2　The Combinations of Supply Voltage and Clock Frequency [7]

| | | | |
|---|---|---|---|
| Config $A$ | Processor Clock | 1.6GHz | 800MHz |
| | Processor Core $V_{dd}$ | 1.484V | 1.036V |
| Config $B$ | Processor Clock | 1.2GHz | 600MHz |
| | Processor Core $V_{dd}$ | 1.276V | 0.956V |

表 3　ベンチマークプログラム
Table 3　Benchmark Programs

| *Benchmark* | *Input Set* |
|---|---|
| gzip | input.source |
| vpr | net.in arch.in |
| gcc | 166.i |
| parser | ref.in |
| vortex | lendian1.raw |
| bzip | input.source |
| mcf | inp.in |

## 4. 評価結果

評価にはプロセッサの処理性能と $ED^2P$ を用いる．図 8 と図 11 にそれぞれ結果を示す．処理性能の結果についてはグラフが高い方が好ましく，$ED^2P$ の結果についてはグラフが低い方が好ましい．グラフにおける **0f/6s** は全ての演算器が低速な場合を示している．**DFUG**，**E-DFUG** は命令のスケジューリング手法にそれぞれ DFUG と E-DFUG を用いた場合を

表 4　キャッシュミス率
Table 4　Cache Miss Rate

| | gzip | vpr | gcc | parser | vortex | bzip | mcf | average |
|---|---|---|---|---|---|---|---|---|
| DL1 cache | 3.5% | 6.4% | 17.1% | 4% | 1.8% | 3.5% | 40% | 10.9% |
| L2 cache | 3.4% | 5.6% | 8.2% | 16.7% | 1.7% | 20.1% | 52.3% | 15.3% |

示している．**PIT** は PIT の特定するクリティカルパス情報を用いて不均質な演算器を使い分けた場合の結果，つまり常にノーマルモードの場合の結果を示している．評価に用いるプロセッサが持つ演算器は有限なため，DFUG，E-DFUG のノーマルモード中や PIT では命令が要求した演算器が既に全て使用中な場合が発生する．発行されるべき演算器が既に全て使用されているがもう一方の演算器に余裕がある場合は，当該命令を空いている演算器で実行するようにスケジューリングしている．これは要求する演算器が空くまで命令を待たせるよりも空いている演算器で実行させた方が性能と消費電力の両面で良い結果が得られているからである．また，比較用に PIT-unrestricted を用意した．PIT-unrestricted はクリティカルパス上の命令を必ず高速な演算器で，クリティカルパス上にない命令を必ず低速な演算器で実行する．例えば，実行可能なクリティカルパス上の命令が 3 つある時は高速な演算器を 3 つ使用できる．この時，低速な演算器は 3 つしか使用できない．

### 4.1　ベンチマークプログラムのキャッシュミス率

初めに，各ベンチマークプログラムのキャッシュミス率を明らかにする．各キャッシュにおけるミス率を表 4 に示す．`mcf`，`bzip`，`parser` のキャッシュミス率は他のプログラムに比べて比較的高いことがわかる．キャッシュミス率が高いこれらのプログラムでは，DFUG と E-DFUG を適用する機会が多く存在すると期待できる．一方 `gcc` は L1 データキャッシュ(DL1) のキャッシュミス率は大きいが，L2 キャッシュのミス率は小さい．今回評価に用いたプロセッサモデルでは．DFUG と E-DFUG は L2 キャッシュミスの期間中に適用される．よって，このような傾向のプログラムには適用可能な機会が少ないことが予想される．その他のベンチマークプログラムについてもキャッシュミス率が小さいため，DFUG と E-DFUG が適用される機会は少ないと考えられる．

### 4.2　処理性能への影響

次に DFUG と E-DFUG を適用した場合，処理性能へどのような影響があるかを調査する．図 8 に処理性能への影響を示す．図 8 において，縦軸はプロセッサの処理性能を，横軸は各ベンチマークプログラムと結果の平均を表している．各結果は全ての演算器が高速な場合の結果で正規化されている．プロセッサの処理性能は評価プログラムによって変わっている．全ての演算器が低速な場合，最大で 24%，平均で 10.2%の性能低下が見られる．この性能低下は深刻である．また，性能低下のばらつきが大きいことも好ましくない．

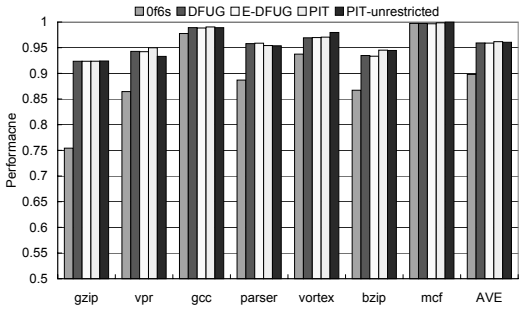



図 8　処 理 性 能
Fig. 8　Performance Result

PIT のみでスケジューリングを行った場合，最大で 7.7%，平均で 3.8%の性能低下が見られる．高速・低速な演算器に制限を設けない PIT-unrestricted は最大で 7.6%，平均で 4%の性能低下が見られる．**PIT** と **PIT-unrestricted** の結果の違いについて考察する．図 8 について考察する．PIT の特定するクリティカルパス情報に従ったのにも関わらず，`vpr`，`gcc`，`parser`，`bzip` などは **PIT-unrestricted** では **PIT** に比べて処理性能が低くなっている．これは PIT が必ずしも正確なクリティカルパスを特定していないことに起因すると考えられる．これは以下の理由による．

前述したように PIT は命令ウインドウ内の命令の DFG からクリティカルパスを特定する．つまり，命令ウインドウ内の命令が異なれば特定されるクリティカルパスも異なる．**PIT** と **PIT-unrestricted** では同一の命令が異なるレイテンシで実行される場合が起こりうる．この時，そのような命令に依存する命令の発行されるタイミングが異なる．その結果，命令ウインドウ内の命令が **PIT** と **PIT-unrestricted** で一致

しなくなる．このため，両者で異なったクリティカルパスを特定することがある．PIT はプログラム実行時に動的にクリティカルパスを特定しているため，このような不確定性を避けることはできない．また，プログラム全体のクリティカルパスとは異なったクリティカルパスを特定することがある．

DFUG を適用した際の処理性能について考察する．**DFUG** の場合，性能低下は最大で 7.7%，平均で 4.1%である．**E-DFUG** の場合は，最大で 7.7%，平均で 4.2%の性能低下となっている．全ての演算器が高速な場合と比較して，`mcf` はほとんど処理性能を低下させていない．これは表 4 に示したように，元々メモリアクセスによりプロセッサがストールする頻度が高かったためと考えられる．ベンチマークプログラムによっては DFUG と E-DFUG は PIT との処理性能の差がやや見られるが，ほとんどの結果には差が見られない．よって，処理性能の面では良い結果が得られているといえる．

### 4.3 命令の内訳

次にどのような命令が高速・低速な演算器を使用しているかという内訳を図 9 に示す．**cp_fast** はクリティカルパス上の命令が高速な演算器で実行されたことを表している．**ncp_fast** はクリティカルパス上にない命令が高速な演算器で実行されたことを示している．同様に，**cp_slow**，**ncp_fast** についてもクリティカルパス上の命令が低速な演算器で，クリティカルパス上にない命令が高速な演算器でそれぞれ実行されたことを示している．**DFUG_slow** は省電力モード中に低速な演算器で実行された命令を示している．前述したように，**PIT** では一方の演算器が全て使用されている場合は，もう一方の演算器で命令を実行させている．このため **cp_slow**，**ncp_fast** のような場合が起こる．

**PIT** と **PIT-unrestricted** を比較する．`gcc` と `vortex` は **PIT-unrestricted** の方が高速な演算器の使用率が高いが，それ以外のプログラムは **PIT** の方が高速な演算器の使用率が高い．これは **ncp_fast** が起こっているためである．

PIT から DFUG へスケジューリング方法が変わると，高速な演算器を使用した割合が平均で 3.7%減っている．4.1 節で予測したように最も変化が大きかったのはキャッシュミス率が高かった `mcf` で，高速な演算器を使用した割合が 12.9%減っている．変化率が次に大きいのは `bzip` と `parer` で，それぞれ 4.4%，3.2%使用率を下げている．**E-DFUG** の場合，**PIT** と比較して平均で 4.8%高速な演算器の使用率が減っている．変化が最も大きかった `mcf` は 18.9%使用率を削減できている．次に変化が大きい `bzip` と `parser` は，それぞれ 5.2%，3.4%使用率を下げている．高速な演算器の使用頻度はキャッシュミス頻度に比例して低下しており，提案手法が正しく適用されていることがわかる．すでに図 8 で見た様に，高速な演算器の使用頻度が低下してもプログラムの実行性能には大きな差はない．

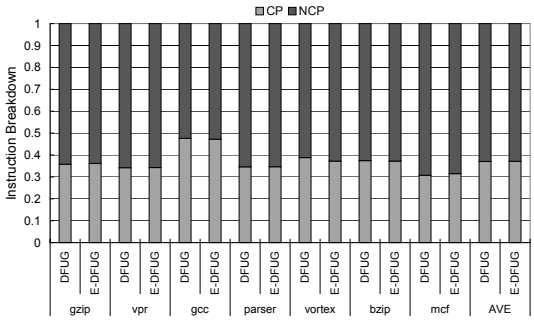



図 10　省電力モード中の命令の内訳
Fig. 10　Instruction Breakdown under the Energy-efficient Mode

図 10 に省電力モード中に実行された命令に対し，PIT がどう判断したかの内訳を示す．図において，**CP** は省電力モード中に実行された命令のうち PIT がクリティカルパス上の命令と判断した命令の割合を示している．同様に **NCP** は，PIT がクリティカルパス上の命令ではないと判断した命令の割合を示している．図 10 中で **CP** に分類される命令は，まさに図 3 に示した状態で実行された命令 (`I:0`, `I:3`, `I:6`, `I:7`) である．PIT は命令ウインドウ中にある DFG の最長パスをクリティカルパスとする．このため，図 3 に示したパスの変化には対応出来ず，`I:0->I:3->I:6->I:7->I:8` をクリティカルパスとしてしまう．図 1 に示したクリティカルパス (`I:0->I:3->I:6->I:7->I:8`) 上のロード命令がキャッシュミスを起こした場合，PIT は省電力モード中に実行可能な命令 (`I:1`，`I:2`，`I:4`，`I:5`) を元々ノンクリティカルとしている．このため，そのような命令が図 10 で **NCP** と分類される．省電力モードで実行中の **CP**，**NCP** の割合は PIT が特定するクリティカル，ノンクリティカルの割合とほぼ同じである．また，DFUG，E-DFUG での違いも見られない．省電力モード下で実行された命令には **NCP** が多く見られるが **CP** も存在する．このことから，図 3 で

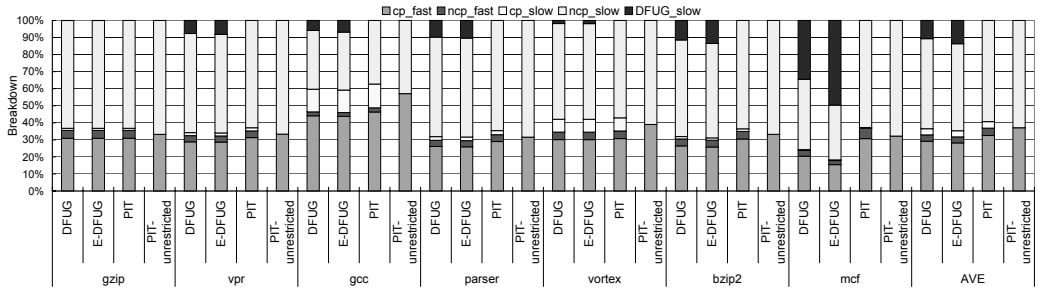
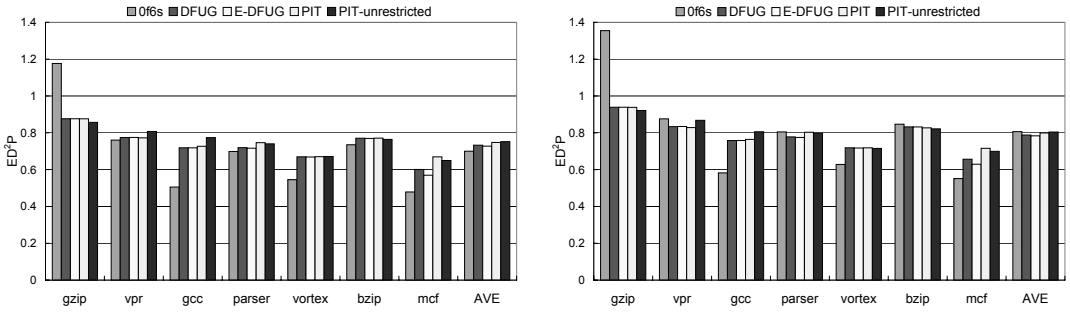



図 9　命令の内訳
Fig. 9　Instruction Breakdown




Config A　　Config B

図 11　エネルギーへの影響
Fig. 11　Energy–Efficiency Results

説明したキャッシュミスによるクリティカルパスの変化が実際に起きていることがわかる．

以上の結果から，PIT を用いて 2.1 節で説明したクリティカルパス情報を用いた省電力命令スケジューリングのみを行うよりも，DFUG，E-DFUG を用いて命令スケジューリングを行った方がより省電力化が期待できる．

### 4.4　消費エネルギーへの影響

最後にエネルギー削減効率についてみる．図 11 に $ED^2P$ の結果を示す．図 11 の右側は表 2 の `Config A` を用いた時の結果を，左側は `Config B` を用いた時の結果をそれぞれ表す．

`Config A` の場合，全ての演算器が高速な場合と比較して **PIT** で平均 25.3%，**PIT-unrestricted** で平均 24.9%，**DFUG** で平均 26.8%，**E-DFUG** で平均 27.3%それぞれ $ED^2P$ を削減している．高速な演算器の使用率が最も下がった `mcf` は **PIT** が 33.1%，**PIT-unrestricted** が 35.1%の $ED^2P$ 削減であるのに対し，**DFUG** で 33.9%，**E-DFUG** で 43.1%$ED^2P$ の削減を達成している．一方 `bzip` は **PIT** が 22.9%，**PIT-unrrestricted** は 23.6%の $ED^2P$ 削減であるのに対し，**DFUG** で 23%，**E-DFUG** で 23.1%の $ED^2P$ 削減に留まっており，DFUG と E-DFUG で期待される効果が顕著に現れていない．`parser` は **PIT** が 25.4%，**PIT-unrestricted** が 26%の $ED^2P$ 削減であるのに対し，**DFUG** で 28.1%，**E-DFUG** で 28.4%$ED^2P$ の削減を達成している．

**PIT-unrestricted** について考察する．`gcc` 以外の全プログラムで **PIT-unrestricted** でスケジューリングした際に $ED^2P$ が **PIT** よりも改善している．このうち，**PIT-unrestricted** で処理性能が低下していたプログラムは `vpr`，`parser`，`bzip` である．一方，`gzip`，`vortex`，`mcf` は処理性能，$ED^2P$ の両方で最良な結果が得られている．このような結果になる理由は，4.2 節で述べたように PIT が命令ウインドウ内の DFG しか考慮していない点にある．

次に DFUG，E-DFUG を適用した場合について考察する．キャッシュミス率が高かったかたプログラムのう

ち，mcf と parser については予想どおりエネルギー効率が改善した．しかし，bzip にはほとんど差が見られなかった．これはキャッシュミス情報を用いた命令スケジューリングが，ノーマルモードで実行されている命令のクリティカルパスに影響を与えたためと考えられる．図 12 と図 13 を用いて説明する．図 12 は I:4 よりも I:7 の命令実行が先に終了する場合を図 13 は I:6 の実行中に省電力モードからノーマルモードにモードが戻り I:4 が発行される場合をそれぞれ表している．各図において，I:1 がキャッシュミスを起こすロード命令である．また，キャッシュミス中に実行可能な命令は I:2，I:3，I:5，I:6，I:7 とする．このためキャッシュミス中に省電力モードが適用されると，これらの命令は全て低速な演算器で実行される．各図ではキャッシュミスを起こす命令，低速な演算器で実行される命令が混在している．このため，図 1 のように全ての命令のレイテンシが同じではないことに注意されたい．図 12，図 13 において，網掛けされている命令は省電力モードで実行される命令を示している．このため，これらの命令の重みは 2 である．網掛けされていない命令の重みはキャッシュミスを起こす I:1 を除いて全て 1 である．I:1 の重みはメインメモリのレイテンシで決まる．省電力モードが期待通りに作用すると，図 12 に示すように I:2，I:3，I:5，I:6，I:7 の各命令はキャッシュミス中に低速な演算器に発行され，実行が終了する．つまり，I:8 の命令発行は I:4 の実行終了に依存する．しかし図 13 に示すように，I:3->I:6:->I:7 の実行が終了しないうちにキャッシュミスが解決された場合，I:4 の実行が I:7 よりも先に終了することがある．このような場合，I:8 の命令発行は I:7 の実行終了に依存する．これは省電力モードで実行可能な命令が大量にある場合に起こりやすい．図 13 の場合，キャッシュミス解決後直ちに I:8 の実行が開始できず，後続命令全体の実行が遅れてしまう．以上のように，省電力モードを適用することで本来ならばキャッシュミス解決までに実行が終了するパスの実行が終わらないことがある．これによるプログラムの実行時間増大の影響が低速な演算器を使用することで削減できるパワーの効果よりも大きいことが bzip でエネルギー効率が改善しない原因である．

次に Config B の場合を考察する．全ての演算器が高速な場合と比較して **PIT** で平均 20.1%，**PIT-unrestricted** で平均 19.6%，**DFUG** で平均 21.2%，**E-DFUG** で平均 21.7%$ED^2P$ を改善し

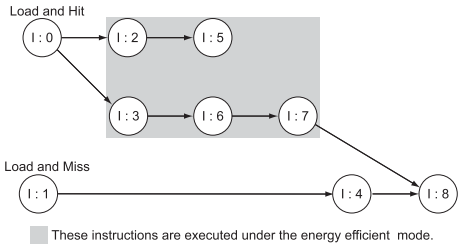



図 12　キャッシュミス中に当該ミスに依存しないパスの命令の実行が終了する場合
Fig. 12　The Execution of the Instructions Not Depending on the Current Cache Miss Is Finished under the Miss

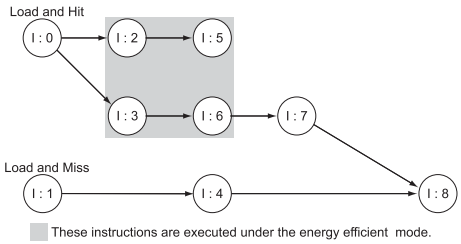



図 13　キャッシュミス中に当該ミスに依存しないパスの命令の実行が終了しない場合
Fig. 13　The Execution of the Instructions Not Depending on the Current Cache Miss Is Not Finished under the Miss

ている．**PIT** よりも **DFUG** で 1.1%，**E-DFUG** で 1.6%$ED^2P$ を改善できている．高速な演算器の使用率が最も下がった mcf では **PIT** は 28.5%，**PIT-unrestricted** は 30.1%，**DFUG** で 34.5%**E-DFUG** で 37.1%$ED^2P$ を削減している．**PIT** と比較すると，**DFUG** で 6%，**E-DFUG** で 8.6%$ED^2P$ を削減できている．逆に bzip では，**PIT** は 17.4%，**PIT-unrestricted** は 17.9%の $ED^2P$ 削減に対し，**DFUG** と **E-DFUG** ではそれぞれ 16.9%しか $ED^2P$ を削減できていない．parser では **PIT** は 19.8%，**PIT-unrestricted** は 20.3%，**DFUG** で 22.3%，**E-DFUG** で 22.6%$ED^2P$ を削減している．**DFUG** と **E-DFUG** は **PIT** よりもそれぞれ 2.5%と 2.8%づつ $ED^2P$ を改善できている．

Config A と Config B のどちらの場合でも，提案手法を適用することで $ED^2P$ の平均値が改善することを確認することができた．

## 5. 関連研究

プログラム実行における処理性能はクリティカルパスによって制限される．クリティカルパス上の命令を

特定できるのなら，それらの命令を高速に実行させることができる．クリティカルパス予測 [5], [14], [19] はクリティカルパスを動的に特定する技術である．クリティカルパス情報はプロセッサの処理性能を改善するだけでなく，その省電力化へも利用できる [12], [13]. Pyreddy ら [12] は，Tune ら [14] によって提案されたヒューリスティックスをプロファイル情報の獲得に利用して，静的にクリティカルパスを特定している．プロファイル情報を利用するコンパイラによって命令にはクリティカルパス情報が付加され，クリティカルとマークされた命令は高速かつ高消費電力な演算器で実行し，クリティカルではないとマークされた命令は低速かつ低消費電力な演算器で実行する．彼らはこのようなアーキテクチャは処理性能を低下させることなく省電力化を可能にすると結論付けているが，実際の電力削減効率については評価をしていない．Seng ら [13] は動的な方法を利用している．彼らはクリティカルパス予測器をクリティカルパス上にない命令を特定するために用いることを提案している．クリティカルパス上にない命令を予測することで，処理性能向上と省電力化を両立させている．彼らはクリティカルパス情報を用いているが，キャッシュミス情報を命令スケジューリングへは利用していない．

Marculescu [10] らはキャッシュミスをトリガとしてキャッシュミス中のマイクロプロセッサへの供給電圧と動作周波数を動的に変更する動的電圧制御 (Dynamic Voltage Scaling: DVS) を提案している．Li [8] らは Variable Supply–Voltage scaling(VSV) を提案している．VSV は L2 キャッシュミス中に供給電圧を低下させて L2 キャッシュミスに依存しない命令を低速に実行させる．VSV では命令レベル並列性が高いときは供給電圧を低下させない．DVS 技術は Multiple Clock Domain マイクロアーキテクチャ [9] に適用することができる．Multiple Clock Domain マイクロアーキテクチャはプロセッサチップを複数の動作周波数で管理する領域に分ける GALS(Globally Asynchronous Locally Synchrous) 技術を利用している．近藤ら [17] は DVS 手法を拡張した DPT(Dynamic Processor Throttling) を提案している．DPT はマイクロプロセッサとメインメモリ間の処理の不均衡を特定する．DPT はプロセッサ–メインメモリ間の処理のバランスが取れていない場合，供給電圧と動作周波数を低下させる．しかし，彼らはクリティカルパス情報を利用していない．

## 6. まとめと今後の課題

プロセッサ–メインメモリ間の性能格差は年々増加しており，それらはプロセッサにおけるストールを頻発させる．また，現在のプロセッサには処理性能向上と省電力化が求められている．これらを考慮して，プロセッサ–メインメモリ間の性能格差を利用して命令をスケジューリングする DFUG と E-DFUG を提案した．DFUG と E-DFUG はキャッシュミス情報を利用して命令実行に利用できる演算器を制限する．提案手法を評価した結果，全ての演算器が高速な場合と比較して平均 4.2%の処理性能低下で平均 27.3%(`Config A` 時) の省電力化を達成することができた．PIT のみを用いてスケジューリングを行う場合と比較しても，平均 0.4%の性能低下で平均 2%(`Config A` 時) の省電力化を達成できている．

今回は動的消費電力のみに着目し省電力化を行ったが，今後は静的消費電力を削減していく必要がある．

**謝　　辞**

本研究の一部は，文部科学省科学研究費補助金 (No. 16300019, No.176549) の援助によるものです．また，SimpleScalar Memory Hierarchy Extensions kit を提供して下さったテキサス大オースチン校の Prof. Doug Burger に感謝します．

**文　　献**

[1] D. Burger, T. M. Austin, “The SimpleScalar Tool Set, Version 2.0”, Technical Report CS-TR-97-1342, Computer Science Department, University of Wisconsin Madison, June 1997.

[2] D. Burger, A. Kagi, M. Hrishikesh, “Memory Hierarchy Extensions to the SimpleScalar Tool Set”, Technical Report TR99-25, Department of Computer Science, University of Texas at Austin, April 1999.

[3] A. Chiyonobu, T. Sato, “Investigating Heterogeneous Combination of Functional Units for a Criticality–based Low–power Processor Architecture”, 3rd International Symposium on Information and Communication Technologies, June 2004.

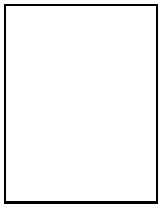
[4] K. I. Farkas and N. P. Jouppi, “Complexity/performance tradeoffs with non-blocking loads”, 21st International Conference on Computer Architecture, April 1994.

[5] B. Fileds, S. Rubin, R. Blodik, “Focusing Processor Policies via Critical-Path Prediction”, 28th International Symposium on Computer Architecture, July 2001

[6] J. L. Hennessy, D. A. Patterson,“Computer Architecture: A Quantitative Approach”, Morgan Kaufmann, May 2002.

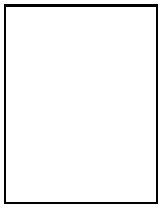
[7] Intel Corporation, "Intel Pentium M Processor Datasheet", April 2004.
[8] H. Li, C-Y. Cher, T. N. Vijaykumar, K. Roy, "VSV: L2-Miss-Driven Variable Supply-Voltage Scaling for Low Power", 36th International Symposium on Microarchitecture, December 2003.
[9] G. Magklis, M. L.Scott, G. Semeraro, D. Albonesi, S. Dropsho, "Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor", the 30th International Symposium on Computer Architecture, June 2003.
[10] D. Marculescu, "On the Use of Microarchitecture-Driven Dynamic Voltage Scaling", Workshop on Complexity–Effective Design, June 2000.
[11] M. Martonosi, David Brooks, Pradip Bose, "Modeling and Analyzing CPU Power and Performance: Metrics, Methods, and Abstractions," SIGMETRICS 2001 / Performance 2001 - Tutorials, 2001.
[12] R.Pyreddy, G.Tyson: "Evaluating Design Tradeoffs in Dual Pipelines", Workshop on Complexity-Effective Design, June 2001.
[13] J.S.Seng, E.S.Tune, D.M.Tullsen: "Reducing Power with Dynamic Critical Path Information", 34th International Symposium on Microarchitecture, December 2001.
[14] E. Tune, D. Liang, D. M. Tullsen, B. Calder, "Dynamic Prediction of Critical Path Instructions", 7th International Symposium on High Performance Computer Architecture, January 2001.
[15] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious", Computer Architecture News, March 1995.
[16] 小林良太郎, 安藤秀樹, 島田俊夫, "データフロー・グラフの最長パスに着目したクラスタ化スーパースカラ・プロセッサにおける命令発行機構", 2001 年並列処理シンポジウム JSPP2001,pp.31-38, 2001 年 6 月.
[17] 近藤正章, 中村宏, "主記憶アクセスの負荷情報を利用した動的周波数変更による低消費電力化", 情報処理学会論文誌コンピューティングシステム, Vol.45, No.SIG6 (ACS6), pp. 1-11, 2004 年 5 月.
[18] 千代延昭宏, 佐藤寿倫, "プログラムの実行時における命令の重要度決定に関する検討" 情処研報 2003-ARC-154-1, pp. 1–6, 2003 年 8 月.
[19] 千代延昭宏, 佐藤寿倫, 有田五次郎, "低消費電力プロセッサアーキテクチャ向けクリティカルパス予測器の評価", 電子情報通信学会論文誌　和文論文誌 C, Vol.J86-C, No.8, pp. 826–835, 2003 年 8 月.

（平成 x 年 xx 月 xx 日受付）



**千代延昭宏**

2002 九州工業大学情報工学部卒．2004 同大学大学院情報工学研究科博士前期課程修了．同年，九州産業大学情報科学部実習助手．2005 より 日本学術振興会特別研究員．現在，九州工業大学大学院情報科学研究科博士後期課程在籍．プロセッサアーキテクチャの研究に従事．情報処理学会，IEEE 各会員．



**佐藤　寿倫**　（正員）

1989 京都大学工学部卒．1991 同大学大学院工学研究科修士課程了． 同年，株式会社東芝入社．マルチプロセッサアーキテクチャ，消費電力見積り手法，および組み込み用マイクロプロセッサの研究開発に従事．九州工業大学情報工学部助教授を経て，現在，九州大学システム LSI 研究センター教授．博士 (工学)．マイクロプロセッサアーキテクチャ，VLSI 設計手法に興味を持つ．情報処理学会，ACM，IEEE 各会員．