「プログラム実行の局所性」の活用法に関する検討

林田, 隆則
九州大学大学院システム情報科学府情報工学専攻

村上, 和彰
九州大学大学院システム情報科学研究院情報工学部門

出版情報：IEICE technical report. Computer systems || 100(248) || p65-72. 100 (248), pp.65-72, 2000-07-26. 電子情報通信学会
バージョン：
権利関係：

「プログラム実行の局所性」の活用法に関する検討

林田 隆則 † 村上 和彰 ‡
† 九州大学大学院 システム情報科学府 情報工学専攻
‡ 九州大学大学院 システム情報科学研究院 情報工学部門
E-mail: smartcore@c.csce.kyushu-u.ac.jp

「プログラム実行の局所性」とは，プログラムが有する一般的な性質であり,「プログラムの実行時間の大部分は，プログラム中のごく少数の命令の実行により費やされている」というものである．プログラム中の高頻度実行部分に対して何らかの最適化を施すことで高性能化，低消費電力化などを実現する手法が実行局所性活用法である．本稿では，再構成可能ファンクションユニット (RFU) を用いる性能向上手法に着目し，RFU で実行するために抽出する高頻度実行部分の割合を変化させた場合に，RFU 搭載プロセッサにおける性能がどのように変化するかについて検討を行う．検討のために，RFU を搭載したプロセッサをモデル化し，そのプロセッサ上での実行をシミュレーションして，速度向上比を測定した．

プログラム実行の局所性，再構成可能コンピューティング，参照局所性，性能評価

Exploiting the Locality of Instruction Execution

Takanori Hayashida Kazuaki Murakami
Depertment of Computer Science and Communication Engineering Graduate School of Information Science and Electorical Enginnering, Kyushu University
E-mail: smartcore@c.csce.kyushu-u.ac.jp

The locality of instruction execution is an intrinsic property of programs which mean most of their execution time is consumed executing only a few instructions. Exploiting this property is an effective way to improve processor's performance and/or reduce its energy consumption. Our strategy is to map the most frequently executed instructions into reconfigurable functional units(RFU). We present two different models for the use of the RFU and simulated their performance.

Locality of Instruction Execution, Reconfigurable Computing,Locality of Reference,Performance Evaluation

1　はじめに

プログラムの実行時間 ET は以下のように定義することができる．

$$ET = IC \times CPI \times CCT \tag{1}$$

ここで，IC は実行命令数，CPI は 1 命令当りの平均所要クロックサイクル数，CCT はクロック・サイクル時間である．

プログラム実行の高速化は，IC,CPI,CCT のそれぞれを低減することによって実現することができる．これらのうち，CCT は半導体技術に大きく依存しており，半導体技術の進歩によって大幅に低減することが可能である．一方，IC および CPI は，プロセッサのアーキテクチャや命令セット，およびコンパイラ技術に大きく依存する．IC や CPI をアーキテクチャの改良によって低減する手法は，次の 2 つのアプローチに大別することができる．

並列化 ：「命令レベル並列性 (ILP:Instruction Level Parallelism)」を活かして，時間方向の並列化，空間方向の並列化を施す．前者の手法はパイプライン処理，後者はスーパスカラ処理や VLIW(Very Long Instruction Word) 方式の採用などがその代表例である．

“Make the Common Case Fast” [1] ：「プログラムの実行の局所性 (Locality of Instruction Execution)」を活用して，頻繁に実行が行われる部分 (Common Case)(以下，高頻度実行部分) の実行サイクル数 ($= IC \times CPI$) を低減するように最適化を施す．

本稿では，後者の “Make the Common Case Fast” に着目し，その 1 実現手法である「再構成可能ユニット (RFU:Reconfigurable Functional Unit)」について議論を行う．本稿では，2 章で実行の局所性について述べ，3 章でその実行の局所性を活用する手法について述べる．次に 4 章にて本稿で対象とする RFU を用いた実行局所性活用手法について述べる．5 章で実験の結果について述べたのち，6 章で考察を行う．7 章で今後の研究課題を述べ，8 章にて関連研究と本実験における結果の比較を行い，9 章にてまとめを行う．

2　プログラム実行の局所性

「プログラム実行の局所性」(または，単に「実行局所性」) とは，プログラムが有する一般的な性質であり,「プログラムの実行時間の大部分は，プログラム中のごく少数の命令の実行によって費やされる」というものである．この性質は,「90/10 則」，すなわち「プログラムの実行時間の約 90%はプログラム中のわずか 10%の命令の実行によって費やされている」という経験則により知られている [**?**].

図**??**は，この経験則を確認するために，SPEC ベンチマークの実行局所性の測定を行った結果である．図**??**において，各プログラムの実行時間の 90%を占めているのは，オブジェクトコード中のおよそ 10%，あるいはそれよりも少ない部分であることがわかる．

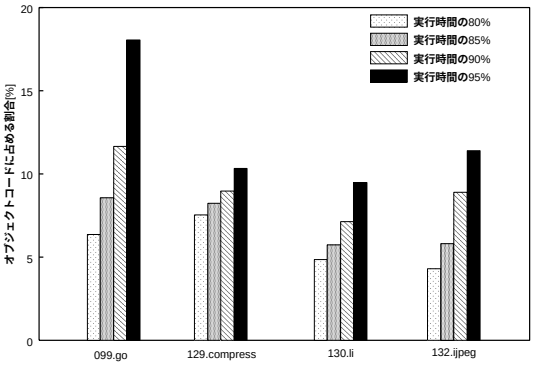

図 1: SPEC ベンチマークの実行局所性

3　実行局所性の活用

実行局所性の活用には，主として次のようなものがある．

(1) カスタム化 ：プログラムにおいて高頻度実行部分を抽出し，当該機能を直接的に実現する複合命令や機能ユニットを設けることで，IC の削減をはかる手法である．古くは CISC 型プロセッサにおける複合命令がこれにあたる．最近では，以下のような例が見られる．

- Intel 社 [**?**] の MMX 等のマルチメディア処理向け SIMD 拡張命令
- Tensilica 社 [**?**] の Xtensa が提供する，ユーザにより定義可能な拡張命令
- Northwestern 大学の Chimaera[**?**] 等で提案されている再構成可能機能ユニット (RFU)

(2) 部分並列化 ：プログラム中で高頻度実行部分を抽出し，当該部分のみを並列処理することで性能向上を目指す手法である．当該部分の命令レベル並列性 (Instruction Level Parallelism) やデータ並列性 (Data Parallelism) が高い場合有効な手法である．古くはベクトル型スーパコンピュータに見られた，スカラ処理ユニットとベクトル処理ユニットの組合せがこれにあたる．最近では以下のような例が見られる．

[1] 本来の意味は,「プログラムの実行において通常稀にしか起きないケースよりも，頻繁に起こるケース (Common Case) に対してアーキテクチャを最適化する」というもの．

データ並列性の活用 ：Intel 社 [?] の MMX などのマルチメディア処理向け SIMD 拡張命令に見られるような，スカラ・プロセッサに SIMD エンジンを組み込む手法.

命令レベル並列性の活用 ：九州大学のハイパースカラ方式 [?] や，BOPS 社 [?] の iVLIW に見られるような，スカラ・プロセッサに VLIW エンジンを組み込む手法

(3) メモリ階層化 ：実行局所性は，プログラムが有するもう 1 つ別の性質,「参照の局所性」としても観測される.「参照の局所性」とは,「一度参照された命令／データ自身，ならびに，その近傍のデータは近い将来参照される可能性が高い」という性質である．この参照の局所性を活用する手法として，古くはキャッシュ・メモリ，最近では以下のような例が見られる.

トレース・キャッシュ(Trace Cache) ：従来の命令キャッシュでは，主記憶上で連続する複数個の命令を，1 つのキャッシュ・ブロックとして主記憶から命令キャッシュにコピーする．これに対してトレース・キャッシュでは，実行した命令のコピーを，実行順に従って配置したトレースをキャッシュ内に保持する．[?] 実行順に従って命令がキャッシュ内に蓄えられているため，一連の処理を繰り返し行うループ処理などでとくに有効である.

スクラッチパッド・メモリ ：実行局所性をメモリ階層化手法に直接応用する手法である．すなわち，プログラム中の高頻度実行部分を，主記憶やキャッシュ・メモリとは異なる「スクラッチパッド・メモリ」と呼ぶ高速メモリに明示的に配置することで，高性能化を目指す手法である．文献 [?] により提案されている手法は，これの応用で，低消費電力化を目指した手法である.

4 再構成可能機能ユニット (RFU)

「再構成可能機能ユニット (RFU)」とは，実行局所性をカスタム化により活用する 1 実現手法である.

あるプログラム中の高頻度実行部分は，必ずしも特定の 1 箇所に集中しているとは限らない．この場合，当該機能全てを 1 つの「カスタム」機能ユニットで実現するのは困難である．さらに，高頻度実行部分は，プログラムによっても異なる．従って，複数のプログラムに対して高頻度実行部分を抽出し，それらを全て「カスタム」機能ユニットで実現するのは非現実的である.

そこで,「カスタム」機能ユニットを FPGA(Field Programmable Gate Array) のような再構成可能ハードウェアで実装し，必要に応じてその構成，すなわち RFU が実現する機能を変更することで，ユニット数を超える機能を実現することが可能になる.

RFU は，対象とするプログラム中で高頻度実行部分の機能 (算術論理演算機能やロード／ストア機能などの一連の処理) を全て 1 個の RFU で実現する．この時，当該機能を実現するにはある一定のクロック・サイクル数を要する．また，RFU においてある機能を実行する場合には，あらかじめ当該機能を構成 (コンフィグレーション) する必要がある．このコンフィグレーションにもある一定のクロック・サイクル数を要する.

5 性能評価

5.1 評価モデル

RFU を用いて実行局所性を活用する場合の，性能に対する効果を評価するために，2 種類の評価モデルを作成した.

モデル SC： 静的コンフィグレーション (Static Configuration) を行うモデル．全ての RFU に対するコンフィグレーションはプログラムの実行前に一度だけ行われる．プログラムの実行中にコンフィグレーションが変更されることがない.

モデル DC： 動的コンフィグレーション (Dynamic Cofiguration) を許すモデル．RFU に対するコンフィグレーションは，プログラムの実行中，必要に応じて適宜行われる.

これらのモデルに対して，以下のようなパラメータを与える.

N_r ：利用可能な RFU 数.

δ ：RFU において，ある高頻度実行部分の機能 i を実行するのに必要なクロック・サイクル数 CCF_i(Clock Cycles for Function) を決定する関数.

ω ：RFU において，ある高頻度実行部分の機能 i をコンフィグレーションするのに必要なクロック・サイクル数 CCC_i(Clock Cycles for Configuration) を決定する関数.

これらのパラメータを与えた評価モデルを，それぞれ

- SC(N_r, δ, ω)
- DC(N_r, δ, ω)

と表記する.

5.2 性能モデリング

評価指標として，1 個のプログラムの実行に要するクロック・サイクル数 (CPP:Clock cycle Per Program execution) を用いる．

$$CPP = \frac{ET}{CCT} = IC \times CPI \tag{2}$$

(1) CPP_{base} ：RFU を用いないベース・モデルでの CPP.

(2) CPP_{rfu} ：RFU を用いた各評価モデルにおける CPP. CPP_{rfu} は，次の式で与えられる．

$$CPP_{rfu} = CPP_{base} - \sum_{i=1}^{N_f} \{(CCF_i^b - CCF_i)E_i - CCC_i \cdot R_i\} \tag{3}$$

ここで，

CCF_i^b ：ベース・モデルにおいて，機能 i を実行するのに要するクロック・サイクル数

N_f ：RFU でその機能を実現される高頻度実行部分の総数

E_i ：高頻度実行部分の機能 i の実行回数

R_i ：高頻度実行部分の機能 i を RFU にコンフィグレーション，または再コンフィグレーションする回数.

である．

5.3 評価方法

1. SUN Ultra 1(UltraSPARC 170MHz,1CPU) 上でプログラム・トレースを採取する．コンパイラは gcc version 2.7 を用い，トレースの採取には qpt2 を使用する．トレースを採取するプログラムは，SPECint95 ベンチマーク・プログラムの 099.go，129.compress，130.li，132.ijpeg で，入力は train を使用する．

2. 採取したトレースから，高頻度実行部分を求める．ここで，

   - 評価モデル SC：命令の種類に関わらず,「命令アドレスが連続した命令列」を対象に，その出現頻度の高いものを高頻度実行部分とする．
   - 評価モデル DC：「分岐命令を含まない命令アドレスが連続した命令列」を対象に，その出現頻度の高いものを高頻度実行部分とする．

3. 以下のいずれかの優先順位に従って，RFU で実行する高頻度実行部分の機能 i を決定.

   E_i 優先：実行回数が多いほどに優先度を高くする．評価モデル SC に適用する．

   $(CCF_i^b - CCF_i)$ 優先：所要クロック・サイクル数の削減効果が高いほど優先度を高くする．評価モデル DC に適用する．

4. 高頻度実行部分の抽出を行う際，プログラムの所要クロック・サイクル数に対して，もしくは実行された全命令に対して抽出を行う割合，すなわち，高頻度実行部分とみなす基準を変化させる．

5. 評価モデル DC において，再コンフィグレーションの必要が生じた場合は，LRU(Least Recently Used) ポリシーに基づいて，再コンフィグレーションの対象となる RFU を決定する．

6. 評価モデルを定義するパラメータのとり得る値は，表**??**に定義されるとおりとする．

6 評価結果と考察

6.1 評価モデル SC

図**??**および図**??**は，全実行命令に対して，RFU の適用範囲を変化させた場合の所要クロック・サイクル数の変化を測定したものである．一方，図**??**は，CPP_{base} に対して，高頻度実行部分として抽出を行う割合を変化させた場合の CPP_{rfu} の変化を測定したものである．図**??**は理想的なモデル ($SC(\infty, \delta_1, 0)$) における結果であるが，このモデル $SC(\infty, \delta_1, 0)$ においても，抽出する部分の割合がオ全実行命令の約 10%を超えたところで CPP_{rfu} の減少がほぼ飽和している．また，モデル $SC(\infty, \delta_2, \omega_2)$ に対する結果である図**??**より，CCC を考慮した場合，高頻度実行部分として全実行命令の 10%以上を抽出すると，コンフィグレーションにかかるクロック・サイクル数 CCC が増大し，CPP_{base} よりもクロック・サイクル数が増加してしまうことがある．

さらに，図**??**より，CPP_{base} の 95%を占める部分まで抽出を行っても CPP_{rfu} が増加しないことから，実行された命令のほとんどが，高頻度実行部分として抽出し，RFU で実行を行うと効果が現れる，全実行命令の 10%の部分に集中していることがわかる．

評価モデル SC における実験の結果から，RFU を用いて実行局所性を活用する場合，高頻度実行部分として抽出を行う対象として，プログラム中で全体のおよそ 10%に相当する部分を選定するのが適切であると考えられる．また，この範囲に対して RFU を適用した場合，ハードウェアのコストを無視した理想的なモデルにおいて，プログラムの実行時間をおよそ 70%程度削減することができる．さらに，RFU のコンフィグレーションが実行中に変更されないモデル SC においては，実行開始時に行う RFU のコンフィグレーションに 5000 サイクル程度かかってもプ

表 1: 評価モデルのパラメータ

	モデル SC	モデル DC
N_r	∞	{8,16,32,64,128,256}
N_f	$N_f = N_r$	$N_f = N_r$ $N_f = 1.5 \times N_r$ $N_f = 2 \times N_r$
δ	$\delta_1 : CCF_i = 1$ $\delta_2 : CCF_i = \lceil ratio \times CCF_i^b \rceil$ ただし，$ratio \in \{0.2, 0.5\}$	$CCF_i = 1$
ω	$\omega_1 : CCC_i \in \{0, 10000, 20000, 50000, 100000\}$ $\omega_2 : CCC_i = coeff \times CCF_i^b$ ただし，$coeff \in \{0, 2000, 5000, 10000, 20000\}$	$CCC_i \in \{0, 10, 50, 100, 500\}$

ログラム全体での所要クロック・サイクル数を削減することが可能である．

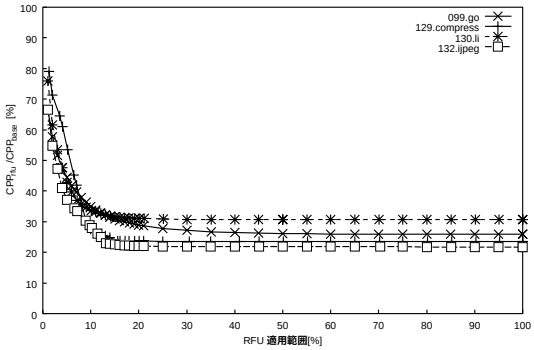

図 2: オブジェクトコードに対する RFU 適用範囲の影響-$SC(\infty, \delta_1, \omega_1(0))$

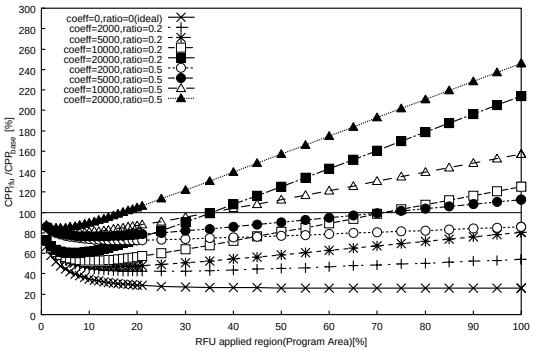

図 3: オブジェクトコードに対する RFU 適用範囲の影響-$SC(\infty, \delta_2, \omega_2)$

6.2 評価モデル DC

評価モデル DC における測定の結果を表??，表??，および表??に示す．

表??は，速度向上率順に N_f 個の高頻度実行部分を選択したとき，選択した高頻度実行部分に含まれる命令数の合計が，プログラムの実行命令数 N_{inst}(オブジェクトコードに含まれる全命令から，実行が行われなかった命令を除いたもの) に対して占める割合 R_{inst} と，プログラム実行時に，選択した全てのブロックの実行に費やされたクロック・サイクル数が CPP_{base} に対して占める割合 R_{exe} を表している．

また，表??は，N_r と N_f を決めた場合に，プログラム

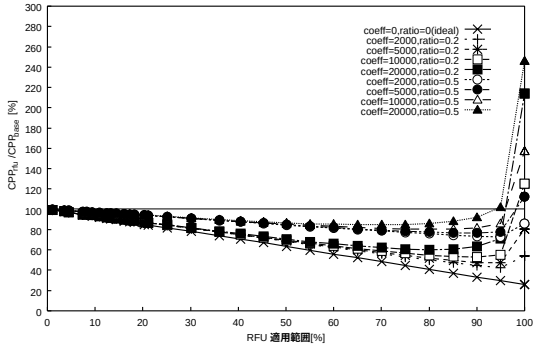

図 4: 実行時間に対する RFU 適用範囲の影響-$SC(\infty, \delta_2, \omega_2)$

の実行中に何回コンフィグレーションの変更が起こったかを示している．ただし，プログラム開始時には RFU は全くコンフィグレーションされていないものとしている．

表??は，モデル DC の CPP と，速度向上比 $Speedup(= \frac{CPP_{base}}{CPP_{rfu}})$ を示している．

モデル DC に対する結果から，RFU の数 N_r が非常に小さい (8,16 程度）の場合，高頻度実行部分の総数 N_f がユニット数の 1.5 倍を超えると，コンフィグレーションにかかるクロック・サイクル数 CCC がプログラムの所要クロック・サイクル数 CPP に大きな影響を与えることがわかる．

これは，プログラム中の多重ループにおいて最も内側に存在するループの基本ブロック数が，RFU 数を超えていて，かつ，これらのブロックがすべて高頻度実行部分として抽出されてしまったことに起因すると考えられる．このような場合，最も内側にあるループ内の処理を実行する度に RFU の再コンフィグレーションが発生するため，その度に CCC_i を要し，結果と CPP に多大な影響を与えることになる．

ある程度 N_r が大きくなり，最も内側に存在するループ内の基本ブロック数を RFU 数 N_r が超えると，コンフィグレーション変更の頻度が急激に下がるため，RFU 命令化の効果が顕著に現れて所要クロック・サイクル数を大幅に削減できていることがわかる．

なお，表??からわかるように，実験に用いたプログラム (129.*compress*) では，64 個の高頻度実行部分が，元の

プログラムにおける所要クロック・サイクル数のおよそ63%を費やしている．また，この部分はプログラムのおよそ 7.7%に相当する．この時，プログラムの所要クロック・サイクル数が約 59%削減できている．

なお，今回実験を行ったモデルは，高頻度実行部分の中にどのような命令列が存在するかについては全く考慮されていないので，全く同じ機能の高頻度実行部分が異なるものとして扱われ，無駄な再コンフィグレーションが行なわれている可能性がある．そのため，抽出された命令列を解析し，抽出した高頻度実行部分の集合を最適化することによって，さらなる速度向上を得られる可能性がある．また，各高頻度実行部分の速度向上比だけを考慮して抽出を行っており，ループ内における複数部分の抽出などに全く制限を設けていないため，RFU 数が 8 や 16 程度のモデルにおいては，ループ内で頻繁に RFU の再コンフィグレーションを引き起こし，速度向上を妨げていると考えられる．

この問題の解決方法として，同じループに属している高頻度実行部分の最大抽出数を規定する方法が考えられる．この条件を適用するには，プログラムのオブジェクトコードを静的に解析する必要があるが，本稿における実験では，この解析を行っていない．この結果，RFU 命令数が RFU 数の 2 倍のモデルにおいて，RFU 数が少ないところでは RFU が性能向上に貢献していない．

表 2: 129.compress の基本ブロック

N_f [*blocks*]	N_{inst} [*instructions*]	R_{inst} [%]	R_{exe} [%]
8	42	1.48	17.31
16	74	2.61	29.01
32	120	4.23	45.11
64	218	7.69	63.48
128	421	14.86	66.45
256	759	26.78	66.53
512	1500	52.93	66.55
1683	2834	100.00	100.00

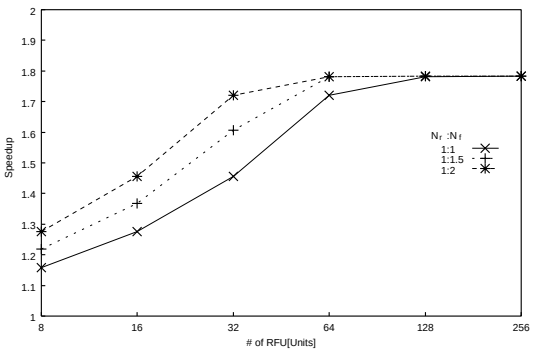

図 5: RFU 数 N_r と速度向上率の関係 (CCC_i=0 の場合)

表 3: モデル DC におけるコンフィグレーション回数

N_r	N_f	$\sum_{i=1}^{N_f} R_i$ [回]	N_r	N_f	$\sum_{i=1}^{N_f} R_i$ [回]
8	8	8	64	64	64
8	12	276	64	96	1392
8	16	1026441	64	128	3222
16	16	16	128	128	128
16	24	494	128	192	2343
16	32	2025213	128	256	8598
32	32	32	256	256	256
32	48	1022	256	384	4758
32	64	254559	256	512	4907

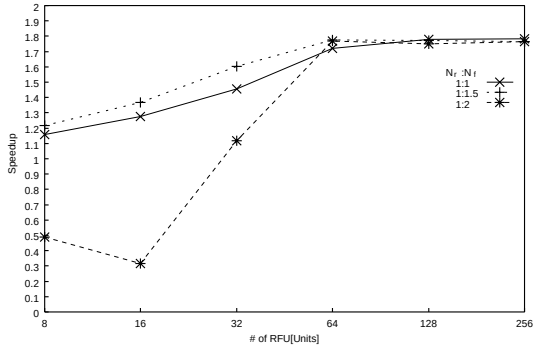

図 6: RFU 数 N_r と速度向上率の関係 (CCC_i=50 の場合)

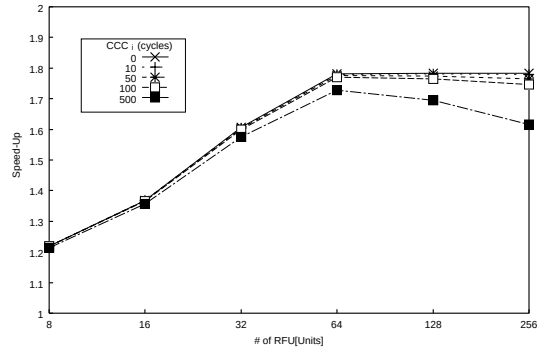

図 7: RFU 数 N_r と速度向上率の関係 ($N_f = 1.5 \times N_r$ の場合)

表 4: モデル DC における測定結果

(a)$CCC_i = 0$ の場合

パラメタ		測定結果	
N_r	N_f	CPP	Speedup
CPP_{base}		40657789	1.000
8	8	35125625	1.1575
8	12	33350317	1.2191
8	16	31859005	1.2762
16	16	31859005	1.2762
16	24	29722280	1.3679
16	32	27917433	1.4564
32	32	27917433	1.4564
32	48	25311506	1.6063
32	64	23627258	1.7208
64	64	23627258	1.7208
64	96	22835772	1.7804
64	128	22819105	1.7817
128	128	22819105	1.7817
128	192	22806667	1.7827
128	256	22800012	1.7832
256	256	22800012	1.7832
256	384	22795982	1.7836
256	512	22795525	1.7836

(b)$CCC_i = 10$ の場合

パラメタ		測定結果	
N_r	N_f	CPP	Speedup
CPP_{base}		40657789	1.000
8	8	35125705	1.1575
8	12	33353077	1.2190
8	16	42123415	0.9652
16	16	31859165	1.2762
16	24	29727220	1.3677
16	32	48169563	0.8441
32	32	27917753	1.4563
32	48	25321726	1.6056
32	64	26172848	1.5534
64	64	23627898	1.7208
64	96	22849692	1.7794
64	128	22851325	1.7792
128	128	22820385	1.7816
128	192	22830097	1.7809
128	256	22885992	1.7765
256	256	22802572	1.7830
256	384	22843562	1.7798
256	512	22844595	1.7798

(c)$CCC_i = 50$ の場合

パラメタ		測定結果	
N_r	N_f	CPP	Speedup
CPP_{base}		40657789	1.000
8	8	35126025	1.1575
8	12	33364117	1.2186
8	16	83181055	0.4888
16	16	31859805	1.2761
16	24	29746980	1.3668
16	32	129178083	0.3147
32	32	27919033	1.4563
32	48	25362606	1.6031
32	64	36355208	1.1183
64	64	23630458	1.7206
64	96	22905372	1.7750
64	128	22980205	1.7693
128	128	22825505	1.7812
128	192	22923817	1.7736
128	256	23229912	1.7502
256	256	22812812	1.7822
256	384	23033882	1.7651
256	512	23040875	1.7646

(d)$CCC_i = 100$ の場合

パラメタ		測定結果	
N_r	N_f	CPP	Speedup
CPP_{base}		40657789	1.000
8	8	35126425	1.1575
8	12	33377917	1.2181
8	16	134503105	0.3023
16	16	31860605	1.2761
16	24	29771680	1.3657
16	32	230438733	0.1764
32	32	27920633	1.4562
32	48	25413706	1.5998
32	64	49083158	0.8283
64	64	23633658	1.7203
64	96	22974972	1.7697
64	128	23141305	1.7569
128	128	22831905	1.7807
128	192	23040967	1.7646
128	256	23659812	1.7184
256	256	22825612	1.7812
256	384	23271782	1.7471
256	512	23286225	1.7460

(e)$CCC_i = 500$ の場合

パラメタ		測定結果	
N_r	N_f	CPP	Speedup
CPP_{base}		40657789	1.000
8	8	35129625	1.1574
8	12	33488317	1.2141
8	16	545079505	0.0746
16	16	31867005	1.2759
16	24	29969280	1.3566
16	32	1040523933	0.0391
32	32	27933433	1.4555
32	48	25822506	1.5745
32	64	150906758	0.2694
64	64	23659258	1.7185
64	96	23531772	1.7278
64	128	24430105	1.6642
128	128	22883105	1.7768
128	192	23978167	1.6956
128	256	27099012	1.5003
256	256	22928012	1.7733
256	384	25174982	1.6150
256	512	25249025	1.6103

7 今後の課題

RFU を用いた実行局所性活用手法の性能に対する効果をさらに正確に測定するには，以下の点について現実の RFU 搭載プロセッサに近づくように改善を行う必要がある．

- モデル DC において，RFU ではどんな処理でも 1 クロック・サイクルで実行可能としている．
- プログラムの静的な解析を必要とする情報が全く考慮されていない．
- シングルタスクのみを対象にしており，タスクスイッチなどが考慮されていない．
- プログラム・入力ともに限定されており，狭い領域における測定しかなされていない．

今後，これらを考慮したモデルを作成し，RFU を用いた実行局所性活用の効果を測定する手法を確立する．とくに，正確な効果の測定を行うためには，オブジェクトコードを静的に解析することが非常に重要である．また，本稿の実験において，高頻度実行部分の抽出は速度向上比のみに着目して行っているが，現実的なモデルを構成するには，これに加えて，抽出した高頻度実行部分がコンフィグレーション情報を生成しやすい命令列で構成されているか，RFU での実行に向いた命令列で構成されているか，などを判断基準に含める必要がある．例えば，シフト演算と他の算術論理演算との組合せなどは，RFU での実行に向いていると考えられる．

これらの点を考慮したモデルを作成し，より正確に実行局所性活用法の効果の測定を行う手法を確立することが今後の研究課題となる．

8 関連研究

Northwestern 大学にて研究が行われている Chimaera[?] は，MIPS を基本として RFU を備えたプロセッサである．文献 [?] では，本稿と同様，基本となる MIPS プロセッサ上でプログラムを実行した場合と，Chimaera 上で実行した場合で性能比較を行っている．

性能比較には，MIPS 上での命令数から定義される性能

モデルと，RFU に回路を実現した場合の回路のクリティカルパスから定義される性能モデルを使用している．前者は本稿で定義した DC モデルに近いモデルである．

Chimaera では，プログラム中において，RFU で実行することで性能向上が期待できる部分をプログラムの静的解析によってコンパイラが抽出する．抽出された部分に含まれる命令列をさらに解析することで RFU で実行する機能を決定する．また，RFU のためのスケジューラがコンフィグレーションのスケジューリングを行い，実行中のコンフィグレーションのオーバヘッドを低減している．文献 [?] では，RFU 数 1，最大 1024 種類のコンフィグレーション情報を保持できるという仮定の下で，アプリケーションによっては性能が向上するとしている．実験では，ADPCM のデコードを行うプログラムで最も性能が向上しており，RFU の処理が全て 1 クロック・サイクルで終了すると仮定した理想的なモデルにおいて，元のプロセッサの 3.52 倍の性能が得られると述べている．

本稿における実験結果と比較した場合，ハードウェアの制限がない理想的な SC モデルを Chimaera においてスケジューラが RFU の割り当てを最適化した場合のモデルに対応させると，ほぼ同じ性能向上比が得られていることがわかる (図**??**).

9 おわりに

本稿では，実行局所性活用法として，RFU を活用した性能向上手法をとりあげ，RFU を活用した場合の効果について，所要クロック・サイクル数を性能の指標として効果測定実験を行った．実験から，RFU を適用する範囲を変化させることで，性能に与える影響が大きく変化すること，また，RFU を効果的に適用した場合，所要クロック・サイクル数の削減に多大な効果をもたらすことが明らかとなった．

今後は，実行トレースに加えてオブジェクトコードの静的解析結果などを利用した効果測定の手法を提案し，RFU を活用した性能向上手法の性能予測を行う手法を確立する．さらに，他の実行局所性活用手法においても同様に効果測定手法を確立することで，実行局所性活用手法が性能に与える影響を見積もる手法を確立する．

謝辞

日頃から御討論頂く，九州大学大学院システム情報科学研究院 安浦寛人教授に感謝致します．また，折りに触れ貴重な御意見を頂く九州大学博士課程 3 年井上弘士氏，ならびに安浦・村上・岩井原研究室の諸氏に感謝します．なお，本研究は一部，文部省科学研究費補助金基盤研究 (A)(2) 一般研究「スケーラブル・システム LSI アーキテクチャの設計手法に関する研究」(課題番号:11308011)，文部省科学研究費補助金基盤研究 (A)(2) 展開研究「システム LSI 向きカスタム化可能 IP コアのアーキテクチャおよび設計支援環境の開発」(課題番号: 12358002) による．

参考文献

[1] J.L.Hennessy and D.A.Patterson,“Computer Architecture A Quantitative Approach”,Morgan Kaufmann Publishers,Inc.,1996.

[2] Scott Hauck, Thomas W. Fry, Matthew M. Holster, and Jeffrey P. Kao,“The Chimaera Reconfigurable Functional Unit”,FCCM 1997 Proceedings pp.87–96,IEEE,April 1997

[3] Zhi Alex Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee, “CHIMAERA:A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit”,27th ISCA Proceedings pp.225–235, ACM ,June 2000

[4] Sanjay Jeram Patel, Daniel Holmes Friendly, and Yele N. Patt,“Critical Issues Regarding the Trace Cache Fetch Mechanism”, Technical Report, The University of Michigan

[5] 宮嶋 浩志, 弘中 哲夫, 斎藤 靖彦, 村上 和彰, “ハイパスカラ・プロセッサ・アーキテクチャ”, 情報処理学会論文誌 Vol.36 No.8,Aug. 1995.

[6] 石原 亨, 安浦 寛人,“低消費電力化を考慮した特定用途向けメモリアーキテクチャ”, 信学技報,VLD98-141,ICD98-287.

[7] Intel 社ホームページ ,http://www.intel.com/

[8] BOPS 社ホームページ，http://bopsnet.com/

[9] Tensilica 社ホームページ，http://www.tensilica.com/