SIMD命令を活用したソフトウェアによる動的分岐予測の提案と予備評価

橘内和也† 城田祐介†
松崎秀則† 前田誠司†

Cellプロセッサが持つSPEは分岐予測器が省略されており，分岐ストールを抑制する代替手段として分岐ヒント命令が用意されている．そこで，SIMD命令を活用したソフトウェアによる動的分岐予測手法を提案し，その有効性について検討する．
試作した分岐予測処理コードでは2レベル分岐予測をベースとし，ループブロック内での分岐予測を10～12のSIMD命令で完了する．テストプログラムによる評価では，分岐予測処理を適用する際に，命令配置を工夫することで，最大約4.8%の性能改善が見られ，提案手法がSPE上のプログラムの実行性能の向上に役立つことがわかった．

Performance Evaluation of Software Dynamic Branch Prediction with SIMD instructions

Kazuya Kitsunai,† Yusuke Shirota,† Hidenori Matsuzaki† and Seiji Maeda†

A Synergistic Processor Element (SPE) of a Cell Processor has branch hint instructions instead of hardware branch predictor for reducing branch stalls. We propose and evaluate a software dynamic branch prediction method for boosting the performance of programs on SPE.
Our implementation is based on the technique of 2-Level Branch Predictor. By using SIMD instructions of SPE, branch prediction is completed in 10 or 12 instructions inside a loop block. With our evaluation, performance improvement of our test program has been measured to be appproximately 4.8% at the maximum. Based on these results, we have found that software branch prediction applied with instruction scheduling improves the performance of programs on SPE.

1. はじめに

Cell Broadband Engine™ (Cell) は PowerPC Processor Element (PPE) と Synergistic Processor Element (SPE) を複数個搭載した高いメディア処理能力を有するマルチコアプロセッサである．SPEはSIMD命令を持ち，ベクトルデータの処理を効率よく行うことが可能である．また，SPEは今日の一般的なプロセッサに比べてシンプルな構成をとっており，例えば，汎用プロセッサの多くが備えているハードウェア分岐予測器が省略されている．そのため，SPEは分岐命令によって次に実行される命令アドレスが変更されると，18～19サイクルのストールが発生する．ストール発生を抑制する手段として，SPEの命令セットには分岐ヒント命令が用意されている．

本研究では，コンパイラ最適化におけるSPE用プログラムの性能向上手段の一つとして，これまでハードウェア分岐予測器が行ってきた動的分岐予測を，SPEが備えるSIMD命令と分岐ヒント命令を活用してソフトウェア実行により行うことを提案する．分岐予測は様々な方式がすでに提案されているが，今回はグローバルヒストリを持つ2レベル分岐予測方式を基に，分岐予測処理コードを試作し，テストプログラムを用いて予備評価を行い，提案方式の有効性を確認する．

2. 背景

2.1 Cell Broadband Engine™

Cellは，東芝，ソニー・コンピュータエンタテインメント，IBMの3社が共同で開発した，高いメディア処理性能を特長とする高性能プロセッサであり，ゲーム機やHD画質のテレビといったコンシューマエレクトロニクス機器に向けた用途が考えられている．

† 東芝研究開発センター
Toshiba Corporate Research and Development Center

Cellの構成を図1に示す[1]．Cellは汎用のPPEと，メディア処理に有効なSIMD演算に特化したSPEを複数搭載したヘテロジニアスマルチコア構成となっている．PPEとSPEはInternal Interrupt Controller (IIC)，Memory Interface Controller (MIC)，Bus Interface Controller (BIC)と共にElement Interconnect Bus (EIB)に結合されている．MICにはメインメモリとしてXDR^{TM} DRAMが接続され，またBICには各種I/Oが接続される．

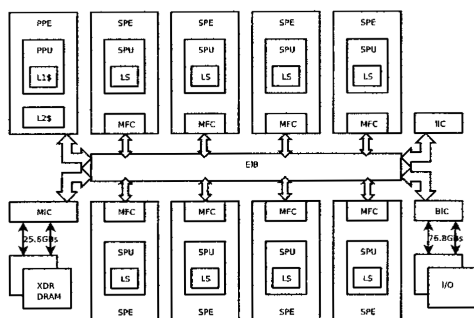

図1 Cell Broadband Engine^{TM}の構成

Cellの高いメディア処理性能の鍵となるのは，SPEと言える[2]．SPEは，3.2GHzの動作周波数で，最大2命令まで同時発行可能なin-order実行を行う．SPEでは256KBのLocal Storage (LS)に対して直接アクセスすることが可能であり，メインメモリへのアクセスはMemory Flow Controller (MFC)が行うDMAによって行われる．SPEの命令セット[3]はSIMD演算に特化しておりバイト(8-bit)，ハーフワード(16-bit)，ワード(32-bit)，ダブルワード(64-bit)，クワッドワード(128-bit)単位でSIMD演算を行うことができる．レジスタは128ビット幅のSIMDレジスタを128本備えている．SIMD処理以外での128ビットより小さいデータ型はpreferred slotと呼ばれる指定された位置に置き，処理をする．

2.2 SPEが備える分岐ヒント命令

CellのSPEでは，ハードウェアコストの観点から分岐予測器が省略された代わりとして，分岐ヒント命令をサポートしている[2]．分岐ヒント命令は，対象とする分岐命令に対して分岐先の命令アドレスのヒントを与えることができ，このヒントが正確であれば，SPEは分岐命令によるストールなしで分岐命令の次の命令を実行することができる．

分岐ヒント命令には，分岐先アドレスを即値で指定するものの他に，レジスタ内に保持した値で指定するもの（r-form）が存在し，プログラム実行中にソフトウェアによる動的な分岐先アドレス計算が可能である．分岐先アドレス計算がソフトウェアで記述できるため，プログラムの特性に応じた分岐予測手法を選択することができ，ハードウェアで行う分岐予測より柔軟な分岐予測が可能と考えられる．

3. ソフトウェア分岐予測

分岐予測方式は，2レベル分岐予測方式を基にした．2レベル分岐予測は，グローバルヒストリを格納するGlobal History Regiser(GHR)と，命令アドレス領域毎に分岐予測方向を決定するための4状態の2ビット予測器を保持するPattern History Table (PHT)から成る分岐予測方式である．

ただし，これをソフトウェアで行う場合，処理サイクル数が増えれば，分岐予測によるストール数の削減効果が失われてしまう．そこで，設計は以下の点を考慮して行った．

- 処理にかかるサイクル数の削減
  より少ないサイクル数になるよう，使用する命令を選択
- 処理をレジスタ内の操作で完結
  LSへのロード／ストアを無くし，処理のサイクル数を削減
- SPE命令にあわせたPHTデータ構造
  SPEが持つSIMD命令にあわせて，PHTのデータ構造を設計．多少，データ領域に無駄があっても，処理命令数が少なくなるようにした．
- 豊富なSPEレジスタの活用
  SPEレジスタを多少占有しても，分岐予測処理全体でサイクル数が減ることを優先させた．

3.1 分岐予測処理の流れ

複数の2ビット予測器の状態を保持するPHTは，通常4KB程度のデータ量となるため，SPEのLSに格納する必要がある．このエントリを毎回ロード／ストアするコストを考えると，それだけでSPEの分岐ペナルティである18〜19サイクル程度のサイクル数となるため，ソフトウェア分岐予測を行っても性能改善が得るのが難しくなってしまう．

このLSへのアクセスを省略するために，1回だけ実行される分岐命令に対して分岐予測するのではなく，何度も繰り返し実行される分岐命令を対象として分岐予測を行う．対象としてまず考えられるのがループブロックである．ループブロック内に含まれる分岐を予測対象とすることで，同じPHTエントリを繰り返し利用する形となり，分岐予測処理毎にPHTエントリをロード／ストアを省略することができる．

ループブロックを前提とした分岐予測処理の流れを図2に示す．分岐予測処理を1)PHTエントリのロード（PHT load），2)分岐予測（Prediction），3)履歴更新（Update），4)PHTエントリのストア（PHT store）の4つのフェーズに分割する．PHT loadとPHT storeをループの外で行うことで，これらがループ全体に対して1回だけ実行されることとなり，LS

へのアクセスが減る．一方，ループブロック内にあるPrediction, Update(T/NT) はループ中毎回実行され，そのつど分岐予測を行うことができる．

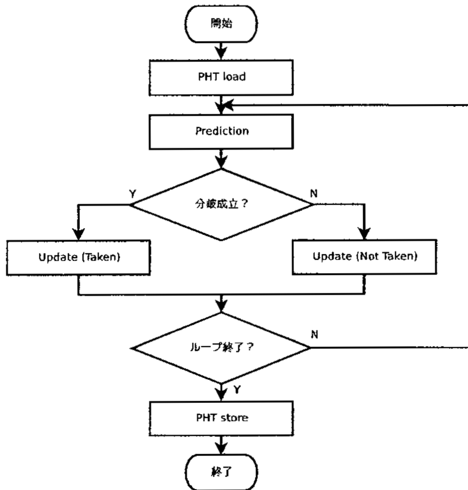

図 2　分岐予測処理の流れ

分岐予測コードを対象プログラムに対して適用する際，4 つの処理をそれぞれ適切な位置に挿入する必要がある．分岐予測処理コードを挿入する位置を図 3 に示す．この図は do-while ループのオブジェクトコード中に各フェーズの処理コードが挿入される位置を示している．分岐予測はループ中に含まれる if-then-else ブロックに対して行う．ループが始まる前に PHT load が一回だけ実行され，ループ内で Prediction, 分岐先で Update(T) もしくは Update(NT) が毎回実行され，ループ終了後に PHT store が一度だけ実行される．

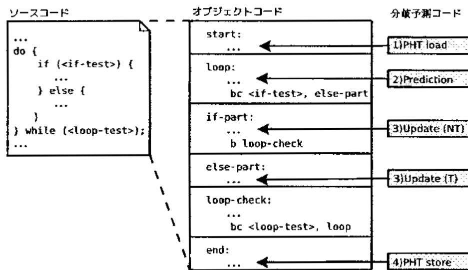

図 3　分岐予測処理コードの挿入位置

3.2　シフト命令を使った状態遷移

2 ビット予測器の状態遷移をそのまま実現する場合 2 ビットの飽和演算が必要となる．この演算を直接ソフトウェアで処理する場合，overflow/underflow のチェック，2 ビット以上のビットをマスクする処理を行う必要があり処理コストが高い．

そこで，状態を図 4 のような 3 ビットで表現し，シフト命令を使ってカウンタの更新を行う方法にすると，さらに少ない命令数で同じ機能を実現することができる．カウンタへのインクリメントは 1 ビット左シフトした後に "001" と論理和をとる，デクリメントは 1 ビット右シフトすればよいため，インクリメントは 2 命令，デクリメントは 1 命令で済む．Taken か Not Taken かの判断は 2 ビット目で判断することができる．

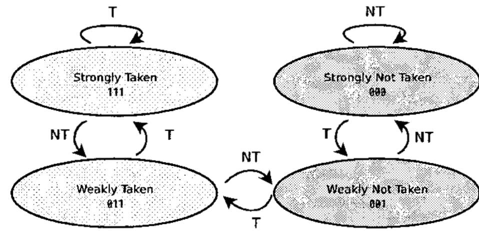

図 4　2 ビット予測器の状態遷移図

3.3　PHT のデータ構造

SPE ではレジスタ内の要素をバイト単位で扱う命令が用意されており，レジスタ内のローテート（グローバルヒストリによる PHT エントリ内のカウンタの選択に利用），shufb 命令による任意のバイト要素の更新（カウンタの更新に利用）といったことがそのまま可能となるため，図 4 の 3 ビットの状態値は，SPE レジスタのバイト要素に合わせて 8 ビットアラインで格納する（図 5）．ただし，3.2 で述べたように，シフト命令によるカウンタ更新で桁あふれを利用する必要があるため，8 ビットのうちの上位 3 ビットに状態値を格納する．SPE のレジスタ幅は 128 ビットであるので，1 つのレジスタ内で 128/8=16 個のカウンタを扱うことができる．$\log_2 16 = 4$ ビットあればカウンタを選択できるため，グローバルヒストリのビット数は 4 ビットとした．

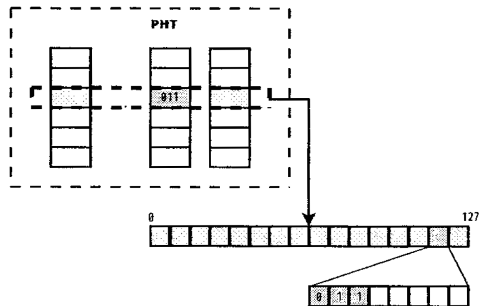

図 5　PHT エントリのデータレイアウト

3.4 分岐予測処理

分岐予測フェーズ (Prediction) では，以下の処理が行われる．

(1) レジスタにロードされた PHT エントリからグローバルヒストリの値に対応する 2 ビット予測器を選択

(2) その状態値を基に selb 命令のマスクを生成

(3) Taken か Not Taken のどちらかの分岐先アドレスを選択

(4) hbr 命令を使って分岐ヒントを対象の分岐命令に対して与える

マスク生成までのレジスタの遷移を図 6 に示す．PHT load フェーズでレジスタにロードされた PHT エントリ ($67) を rotqby 命令でグローバルヒストリが格納されているレジスタ ($64) の下位 4 ビットの値の分だけローテートすることで，該当する 2 ビット予測器の状態値が格納されているバイト要素を$68 の preferred slot に移動する．fsmh 命令で状態値の各ビットがそれぞれハーフワード要素に展開された後，さらに xshw 命令で状態値の 2 ビット目のビットが展開されたバイト要素を preferred slot に展開する．これにより，状態値の 2 ビット目が preferred slot に展開されたこととなり，以降で分岐先アドレスを選択する selb 命令のマスクとして使うことが出来る．

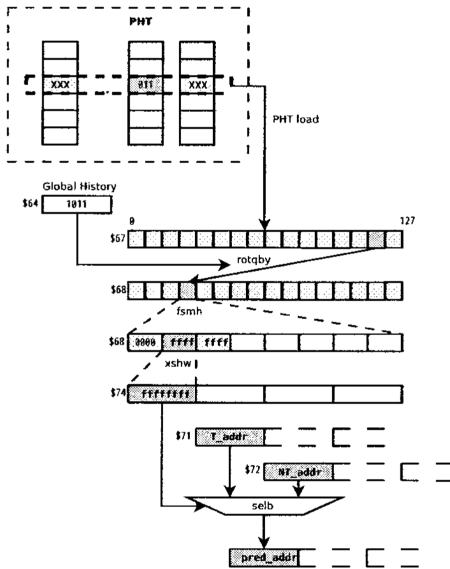

図 6 分岐予測処理

分岐ヒントを与えるために，PHT load フェーズでロード済みの 2 つの分岐先アドレスから予測先アドレスを選択する．今回は直接分岐命令を扱うため，分岐結果は分岐成立か分岐不成立のどちらかであるので，PHT load フェーズでロード済みの 2 つの分岐先アドレス ($71, $72) と，先ほど生成したマスク ($74) を引数に selb 命令を使い，予測先アドレスを選択する．最後に，選択された予測先アドレス ($70) を引数に，r-form の分岐ヒント命令である hbr 命令を使ってヒントを与える．

3.5 履歴更新処理

履歴更新フェーズ (Update) では，分岐結果に応じて 2 ビット予測器の状態を更新した後，分岐予測処理とほぼ逆の処理を行い，レジスタ内の PHT エントリ ($67) を更新する．最後に，分岐結果に応じてグローバルヒストリを更新する．

図 7 に分岐結果が Taken だった場合の履歴更新処理におけるレジスタの遷移を示す．分岐結果が Taken だった場合，2 ビット予測器の状態の各ビットがハーフワード要素に展開されているレジスタ ($68) の 2 バイト（1 ハーフワード要素）分を左シフトした後，2 番目のバイト要素が 0xffff となるよう論理和をとる．Not Taken だった場合は，2 バイト（1 ハーフワード要素）分を右シフトするだけでよい．その後，gbh 命令で各ハーフワード要素の最下位ビットが preferred slot のバイト要素に集められ，最後に shufb 命令でこの更新された状態だけを PHT エントリが格納されているレジスタ ($67) に書き戻す．

グローバルヒストリの更新は，分岐結果が Taken だった場合，1 ビット分左シフトした後に”0001”と論理和を取ることでグローバルヒストリが保持されているレジスタ ($64) の最下位ビットに”1”をセットする．分岐結果が Not Taken だった場合，1 ビット分左シフトすれば最下位ビットに”0”がセットされる．

このように，ループ内であれば PHT エントリ ($67) はそのまま次の分岐予測でも使うことができるので，ループ内はレジスタ処理だけで完結することができる．ループ内で行う分岐予測処理に必要な命令数は Prediction フェーズで 5 命令，Update フェーズでは Taken で 7 命令，Not Taken では 5 命令となり，合計 10〜12 命令で分岐予測を実現することができる．

3.6 レジスタ依存ストールを利用した分岐予測処理の埋め込み

プログラムが SPE 上で実行される際，レジスタ依存により多くのストールが発生することがある．この空きサイクル中に分岐予測処理を実行することで，オーバーヘッド無しで分岐予測を行うことができる．

図 8 は，mpyh 命令の前に fsmh 命令を埋め込んだ例で，実線の箱は各パイプラインステージを，点線の箱はレジスタ依存によるストールを示している．図 8 の (a) では，レジスタ$5,$6 の依存により mpyh 命令がストールしている．(b) で示すように，実処理と依存しないレジスタを使用する fsmh 命令を配置することで，mpyh 命令のストール中に処理をインタリーブすることができる．

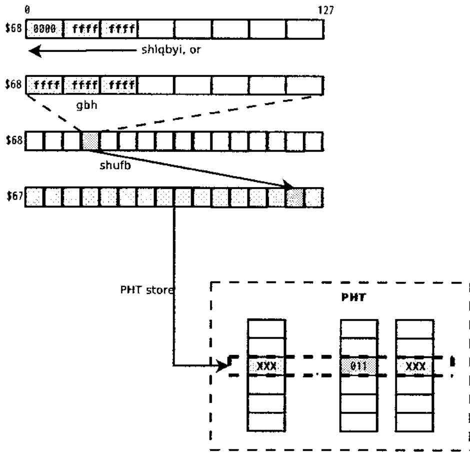

図 7　履歴更新処理

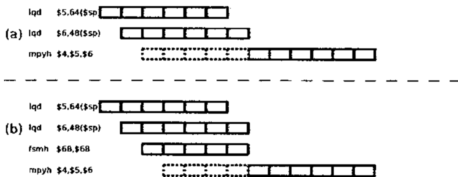

図 8　ストールにあわせた命令配置の例

4. 評　価

本方式の有効性を確認するため，2つのテストプログラムを作成し，評価を行った．

4.1 テストプログラム

使用したテストプログラムのソースコードの一部を図9に示す．図の四角部分が予測対象のブロックを示している．1つめのsimple関数では，予測対象if-then-elseブロックの分岐方向は"NNTTNNTT…"というように，2回ずつ分岐方向が変化するようになっている．2つめのsimple2関数では，予測対象の前に実行される2つのifブロックの結果によって変数aa,bbの結果が決まり，予測対象のif-then-elseブロックの分岐方向に影響をあたえる．両方とも，2ビット予測方式では分岐予測を失敗し，グローバルヒストリを持つ2レベル分岐予測方式が有効に働く形となっている．

4.2 測定環境

実行時間の測定には，東芝のCell Reference Setを使い，より詳細なストール数などの解析には，Cellシミュレータのsystemsim ver1.1(x86版)を使用した．プログラムのコンパイルは東芝のSDKに含まれているgccを用いて行った．測定方法は，実機ではSPEのデクリメンタを使い，シミュレータでは，関数の入り口と出口でsystemsimのbreak point機能を使って実行を停止させ，そのときに出力される統計情報から測定した．

```
int simple(void)
{
    ...
    do {
        do {
            if (<if-test>) {
                ...
            } else {
                ...
            }
        } while (j < COLS);
    } while (i < LINES);
}

int simple2(void)
{
    ...
    for (i = 0; i < N; i++) {
        if (aa == 2)  aa = 0;
        if (bb == 2)  bb = 0;
        if (aa != bb) {
            ...
        } else {
            ...
        }
    }
    ...
}
```

図 9　使用した2つのテストプログラム

4.3 測定結果

それぞれのテストプログラムに対して，比較のため以下の3つの方式について測定した．

- no_hint：分岐予測なしの実処理のみ
- sbr(no_sched)：2レベル分岐予測処理を実処理の前に配置
- sbr(sched)：2レベル分岐予測処理を実処理の空きサイクルに命令単位で配置

まず初めに，シミュレータから得られた全体の処理サイクル数を図10に示す．命令スケジューリングなしで分岐予測を適用したsbr(no_sched)では，分岐予測なしのno_hintに比べて処理サイクル数がsimpleで約11%増加，simple2で約5.4%増加した．一方，命令の配置を工夫したsbr(sched)では，no_hintに比べて処理サイクル数がsimpleで約4.9%減少，simple2で約1.2%減少した．

sbr(no_sched)で処理性能が悪化したのに対し，sbr(sched)で向上したことから，分岐予測処理自体のコストが分岐予測により削減された分岐ストールのコストを上回っていたと考えられる．さらに，命令単位の配置の工夫により，分岐予測処理が実処理の空いたサイクルやパイプラインで実行されるようになり，処理コストが隠蔽されることで，全体として性能が向上したと考えられる．

図11は全体のCPIを示している．sbr(no_sched)ではno_hintに比べてCPIがsimpleで約0.56向上，simple2で約0.21向上し，sbr(sched)ではno_hintに比べてsimpleで約1.02向上，約0.4向上した．CPIが向上した理由は，分岐ストールが削減されたことと，命令配置の工夫により空きサイクルが減少したことと考えられるが，後者は，実処理自体のCPI向上には寄与しないので，実処理の性能向上に含めることはで

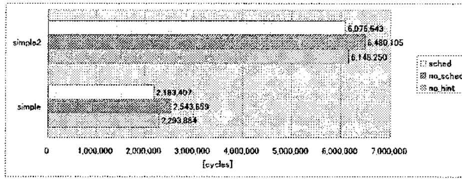

図 10 全体の処理サイクル数

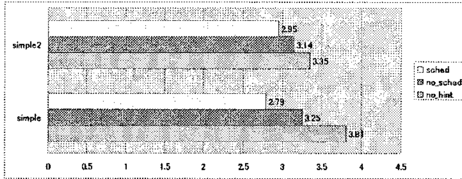

図 11 全体の CPI

きない.

図 12 は分岐予測ミスによるストールサイクル数を示している. これは, 分岐予測によるストール数の削減効果と見ることができ, それぞれ simple で 124,360[cycle] 減少, simple2 で 130,964[cycle] 減少した. なお, sbr(no_sched) と sbr(sched) の間では, 命令配置に差があるだけで, 分岐ストール数に差はない.

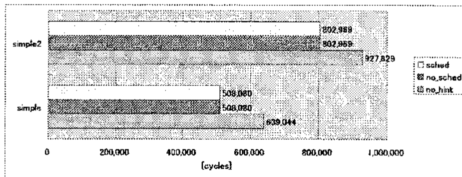

図 12 分岐予測ミスによるストールサイクル数

図 13 は依存によるストールサイクル数を示している. sbr(no_sched) では, no_hint に比べて simple で 221,186[cycle] 増加, simple2 で 227,330[cycle] 増加している. さらに, sbr(sched) では, no_hint に比べて simple で 122,880[cycle] 減少, simple2 で 137,218[cycle] 減少している. 命令配置の工夫によりレジスタ依存ストールが減少したものと考えられるが, CPI と同様に実処理のストール中に分岐予測処理を実行させることでストール数が減少したためで, 実処理自体の依存ストールが削減されたと言うことはできない.

図 12 と図 13 の増減から, 分岐予測によるストールの削減効果を依存によるストールが上回っていることが確認できる. つまり、ソフトウェア分岐予測のコストの多くを実処理の空きサイクルで隠蔽することができれば、分岐予測処理のコストを下げ、分岐予測による性能改善が期待できると言える。

最後に, 実機から得られた全体の処理時間を図 14

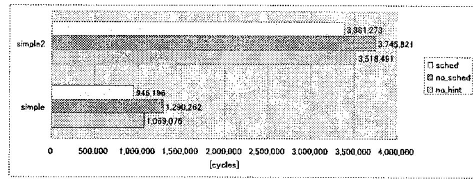

図 13 依存によるストールサイクル数

に示す. sbr(no_sched) では, no_hint に比べて処理時間が simple で約 10.9%増加, simple2 で約 5.4%増加した. 一方, sbr(sched) では, no_hint に比べて処理時間が simple で約 4.8%減少, simple2 で約 1.2%減少した. 概ね, シミュレータ上での結果と一致し, 実際に性能向上があることを確認することができた.

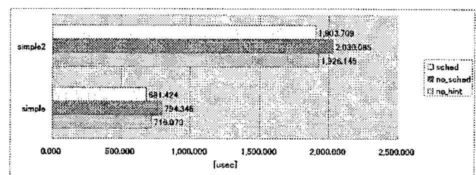

図 14 全体の処理時間 (実機)

5. おわりに

SPE の SIMD 命令と分岐ヒント命令を活用したソフトウェア分岐予測により, 性能向上の可能性があることが確認できた. また, 評価結果から, 性能向上を達成するには, 分岐予測処理を実処理の空きサイクルに隠蔽することが重要とわかった. 今後は, より多くのテストケースで評価を進めていくと同時に, さらに分岐予測方式の改善や適用方法の検討に取り組む予定である.

参考文献

1) Pham, D., Asano, S., Bolliger, M., Day, M., Hofstee, H., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y. et al.: The design and implementation of a first-generation CELL processor, *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pp.184–186 (2005).

2) Flachs, B. et al.: A streaming processing unit for a CELL processor, *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pp.134–135 (2005).

3) *Synergistic Processor Unit Instruction Set Architecture.*