ハイパフォーマンス
コンピューティング　93−21
（２００３．３．１２）

オブジェクトコードの配置アドレス改善による プログラム高速化と評価

高村 明裕 †　梅原 俊治 ‡　三木 良雄 †
akihirot@crl.hitachi.co.jp　shunji.umehara@itg.hitachi.co.jp　miki@crl.hitachi.co.jp

†(株) 日立製作所 中央研究所
〒 185-8601 東京都国分寺市東恋ヶ窪 1–280

‡(株) 日立製作所 エンタープライズサーバ事業部
〒 259-1392 神奈川県秦野市堀山下 1 番地

あらまし 高性能プロセッサではプログラムを変更した際に，変更を行なっていない箇所で性能変動が起こりうるという問題がある．この性能変動は，プログラム変更箇所と性能低下箇所が異なっているため原因の追及が困難で，性能チューニングの際に問題となっていた．本報告書は，使用頻度を考慮したメモリ配置とスペーサー挿入による命令とデータのアドレス調整を行なうことで，性能変動を抑え，安定して高い性能を得ることができることを示す．実際のプログラムを用いて評価したところ，POWER3-II プロセッサで性能変動を 7.3%から 1.1%に，POWER4 プロセッサで性能変動を 9.3%から 3.6%にすることができた．

Performance Enhancement Techniques based on Object Code Reallocation and their Evaluation

Akihiro TAKAMURA†　Shunji UMEHARA‡　Yoshio MIKI†
akihirot@crl.hitachi.co.jp　shunji.umehara@itg.hitachi.co.jp　miki@crl.hitachi.co.jp

†Hitachi, Ltd., Central Research Laboratory
1–280, Higashi-koigakubo Kokubunji-shi,
Tokyo 185–8601, Japan

‡Hitachi, Ltd., Enterprise Server Division
1 Horiyamashita, Hadano-shi,
Kanagawa 259–1392, Japan

Abstract This paper shows a new performance enhancement technique that stabilizes performance fluctuation in high-performance processors when a program is modified. High-performance processors have the performance fluctuation problem that when one part of a program is modified, the performance of another part of the program may change. Performance tuning is difficult in this case because the parts of the program modification and performance change are different. To stabilize the performance of a program, we introduce a method that sort the instructions and the data by usage frequency and adjusting the addresses of the instructions and the data. Experimental results show that the performance fluctuation is reduced from 7.3% to 1.1% on POWER3-II microprocessor and from 9.3% to 3.6% on POWER4 microprocessor.

1 はじめに

近年の高性能プロセッサは，キャッシュメモリに代表される “よくあるケースを高速化する機構” を搭載している．しかしながら，性能チューニングの為にプログラムを変更した際に，高速化機構が上手く働かず，変更を行なっていない箇所でも性能が変動するという問題 (性能変動問題) がある [1].

性能変動は，プログラムの変更箇所によらず，プログラムの変更毎に起きる可能性がある．このため，性能変動の原因箇所の特定が困難であること，処理性能が要求されるプログラムでは変更毎に対策が必要であることが問題である．また，プログラムを変更する際，“変更していない箇所の性能は変化しない” ことを暗黙のうちに仮定して性能チューニングを行なうことが多いが，性能変動はこの仮定から外れているため，性能チューニングを難しくしてしまうことも問題である．

本報告書では，新たに判明した性能変動の原因を明かにすると共に，対策方法とその評価を行なう．これにより，性能変動を抑え，プログラムを高速化する方法を示す．

2 性能変動問題

2.1 性能変動問題とは

本報告書では，プログラムの変更時に，変更していない箇所の処理性能が変動する問題を性能変動問題と呼ぶ．性能変動問題の一例として，プログラム起動時に一度だけ実行される初期化処理を変更した場合に，初期化処理後に実行されるメインループでの処理性能が低下することが挙げられる．

2.2 性能変動の原因

性能変動の原因は，プログラムを変更した箇所のオブジェクトコードサイズが変化することにより，命令及びデータが配置されるメモリアドレスが変化することと，アドレスが変化したことにより，高性能プロセッサ内の高速化機構である分岐予測機構とキャッシュがうまく動作しなくなることであることがわかっている [1]．また，次節で説明する store → load の擬似依存関係も性能変動の原因となることが新たに判明した．

プログラムを変更した際，分岐予測機構がうまく動作しなくなる原因は以下の通りである．プログラムを変更したことにより，変更箇所のオブジェクトコードサイズが変化すると，変更箇所より前のメモリアドレスに配置される分岐命令のアドレスは変化しないが，変更箇所より後のメモリアドレスに配置される分岐命令のアドレスが変化する．一般に分岐予測機構は分岐命令のアドレスを用いて分岐予測テーブルの添字を算出している．このため，異なる二つの分岐命令のアドレスの差分が変化すると，異なる分岐命令が同じ分岐予測情報を用いることがあり，分岐予測ミスによる性能低下の原因になる．

プログラムを変更した際，データキャッシュがうまく動作しなくなる原因は以下の通りである．データは属性ごとに連続するメモリ領域に配置されている．すなわち初期化が必要なデータ (DATA)，初期化が不要なデータ (BSS)，malloc() でメモリ割り当てが行われるデータ (heap)，ローカル変数を格納するデータ (STACK) のいずれかの領域に格納されている．このうち，STACK 領域以外は，命令が格納されている TEXT 領域に続いて順にアドレスが割り当てられるため，オブジェクトコードサイズが変化すると配置されるアドレスが変化する．したがって，オブジェクトコードサイズが変化すると，STACK 領域にあるデータと，その他の領域にあるデータが同一のキャッシュラインに割当たる可能性があり，キャッシュミスの原因となる．また，DATA 領域，BSS 領域，heap 領域が異なるアドレス境界で整列されていると，オブジェクトコードサイズが変化した時に領域毎に変化するアドレス量が変化するため，キャッシュミスの原因となる．

2.3 store → load の擬似依存関係による性能変動

POWER3-II プロセッサを搭載した IBM RS/6000 44P モデル 270 上でプログラムを動作させた際に，後に述べる第 4.1 節の図 5 のように性能変動問題により性能が約 7%低下するケースがあった．このケースではオブジェクトコードサイズを増やしていくと，4KB 周期で性能が低下していた．

このケースについて，原因調査を行なったところ，store x の後に load y を実行する際，POWER3-II プロセッサでは store x と load y が参照するアドレス x と y の下位 12 ビットが同じ (アドレスの差分が $4\times n$ KB) である時に，load の実行が遅れることがわかった．store x と load y が参照するアドレスは，オブジェクトコードサイズによって変化することがあるため，性能変動の原因となる．

store x を実行した後に load y を実行すると，load

y の実行が遅れることがある理由は，load y の結果が store x の結果に依存しないことを判定し，依存しないことが判明すれば store x と並行して load y を実行するが，そうではない時は store x の完了を待ってから load y を実行するためであると考えられる．

store x を実行した後に load y を実行するとアドレス x と y の下位 12 ビットが同じである時に load y が遅れるのは，論理アドレスで判定を行なうためであると考えられる．load y の結果が store x の結果に依存するのは，論理アドレス x と論理アドレス y をそれぞれ物理アドレスに変換した物理アドレス x' と物理アドレス y' が同じ時である．ところが，プロセッサ高速化の為に物理アドレスへ変換する前の論理アドレスで判定を行なうと，論理アドレスから物理アドレスへの変換によって変化しない下位 12 ビットが異なる場合は依存しないと判定できるが，下位 12 ビットが同一である場合は依存するか否かを判定することができない．そこで，POWER3-II では，下位 12 ビットが同一である場合は依存する可能性があると判定し，store x が完了するまで load y の実行を遅らせていると考えられる．

3 性能変動問題の対策

3.1 対策の方針

性能変動の原因は，プログラムを変更した箇所のオブジェクトコードサイズが変化すると，命令 (データ) が配置されるメモリアドレスが変化することである．この時，2 つの命令 (データ) の間でアドレス差分が変化すると，性能低下が起きることがある．

メモリアドレスの変化を防ぎ，性能変動を抑える方法として，スペーサーを挿入することでアドレスの変化を抑える方法が提案されている [1]．しかしながら，アドレスの変化を防ぐために，スペーサーを挿入する箇所を増していくと，キャッシュのヒット率が低下してしまう．そこで，本報告書では，使用頻度を考慮したメモリ配置とスペーサーの挿入を併用することでアドレスの変化を防ぐ方法を示す．

3.2 スペーサ挿入によるアドレス調整

スペーサ挿入によるアドレス調整では，プログラムの変更があっても 2 つの命令 (データ) の間でのアドレス差分が変化しないように，コンパイル後にアドレスを調節するためのスペーサを挿入する．アドレスを調節する箇所は，性能変化への影響の大きい (一般的には使用頻度が高い) 命令 (データ) が配置されているアドレスとする．

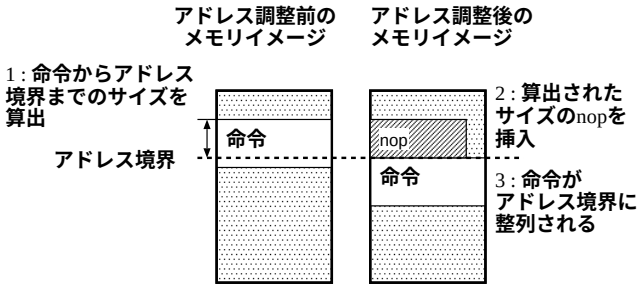

図 1: スペーサ挿入によるアドレスの調整

アドレスを調整する手順を図 1 に示す．まず，プログラムのコンパイルを行ない，アドレス調整を行なう命令 (データ) のアドレスを求める．次に，2^nByte のアドレス境界と，命令 (データ) とのアドレスとの差を求め，アドレス調整を行なう箇所に nop を挿入する．最後に，再度コンパイルを行なう．

スペーサー挿入によるアドレス調整では，アドレス調整する箇所を多くすると，スペーサーで挿入した nop もキャッシュされるために，実効的なキャッシュ容量が減少し，性能が低下することがある．また，単純に 2^nByte のアドレス境界に整列させると連想度不足によるキャッシュミスが発生しやすくなる．そこで，次節に示す使用頻度を考慮したメモリ配置を併用することにより，性能の低下と性能変動を抑える．

3.3 使用頻度を考慮したメモリ配置

使用頻度を考慮したメモリ配置では，性能変化への影響の大きい命令 (データ) をメモリ上に連続して配置する．命令 (データ) をメモリ上に連続して配置すると，連続する領域外のオブジェクトコードサイズが変化してもメモリ上に連続で配置された領域内の任意の二つのアドレスが同じだけ変化するため，性能は変化しない．たとえば，図 2 に示す様に，データを連続して配置すると，連続する領域以外のオブジェクトコードサイズが変化しても，キャッシュミスは起きない．しかしながら，連続する領域内のオブジェクトコードサイズが変化すると，性能が変化する可能性があるため，性能変化量の期待値を小さくするためには，使用頻度が高いプログラムやデータを選んでメモリ上に連続して配置する必要がある．

命令を連続したメモリ上に配置するには，命令列に対応するソースファイル上の関数を一つのソースファイルの中で順に記述すればよい．しかしながら，データは属性 (DATA : 初期化済みの静的データ，BSS :

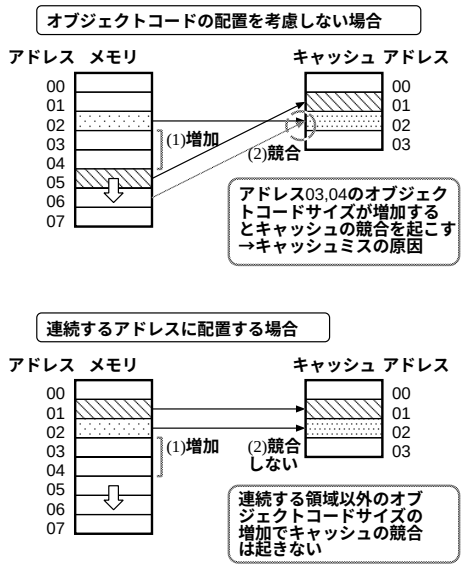

図 2: 連続するアドレスに配置する効果

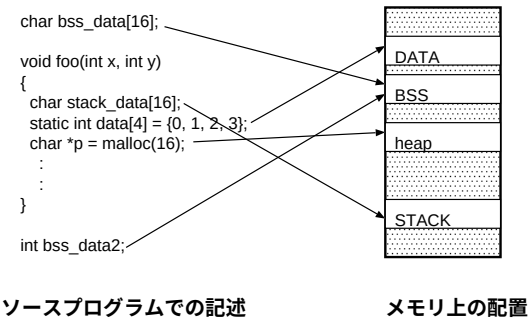

図 3: ソースプログラムでのデータの記述とメモリ上の配置

初期化不要な静的データ，heap：malloc() で動的に確保されるデータ，STACK：スタック上に配置されるデータ) ごとに連続する領域に配置される．このため，図 3 で示す様にソースファイル上で順に記述されていても，データの属性が異なっていればメモリ上では離れた場所に配置されてしまう．

そこで，データを連続するメモリ上に配置するために，データの属性をそろえた上でソースプログラム中に順に記述する．もし，データの属性が異なる領域が複数回参照される場合には，図 4 に示すようにデータの属性を同じにするためのデータ領域を確保しておき，複数回参照される前に属性が同じである領域へデータをコピーしておき，複数回参照する時には属性が同じデータを参照する．

これにより，使用頻度の高い命令，データをメモリ上の連続する領域に配置することができ，性能変動が起こる確率を下げることができる．

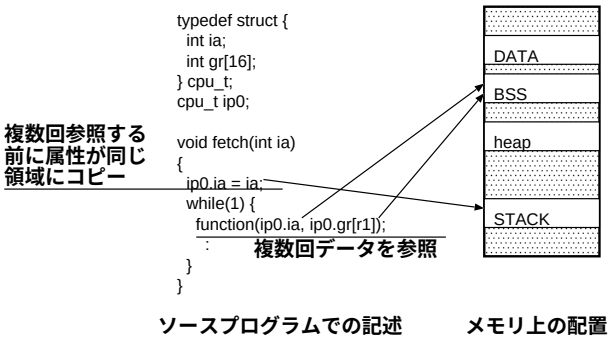

図 4: コピーによりデータを連続するメモリ上に配置する方法

4 性能変動対策の評価

4.1 POWER3-II プロセッサでの評価

POWER3-II プロセッサにおいて，提案する性能変動の対策方法の効果を定量的に評価するため，対策前のプログラムと，対策後のプログラムに対して，性能変動の原因を人為的に起こした際の実行時間を測定した．

POWER3-II プロセッサ [2] では，分岐予測ミスによる 1KB 周期での性能変動と store → load の擬似依存関係による 4KB 周期での性能変動が起こりうる．そこで，実行時間の測定と同時に，ロードストアユニットの BUSY サイクル数，分岐予測ミス回数をハードウェアモニタを用いて測定した．ロードストアユニットの BUSY サイクル数は store → load の擬似依存関係が発生すると増加することがわかっており，プロセッササイクル数とロードストアユニットの IDLE サイクル数から算出する．

対策後のプログラムは，実行頻度の高い 10 個の関数を連続配置した上で，メインループを含む関数の開始アドレスを 64KB 境界に整列，データの開始アドレスを 4KB 境界に整列させている．

性能変動の原因を人為的に発生させるため，nop をプログラム中に挿入した．挿入する nop の量は，POWER3-II プロセッサのキャッシュラインサイズの半分である 64Byte ごとに 0〜16KByte とした．

評価にはコンピュータシステムのエミュレーションを行なうプログラムを用いた．プログラムは POWER3-II プロセッサを搭載した IBM RS/6000 44P モデル 270 上で動作させた．OS は AIX 5L である．

対策前と対策後の実行時間 (プロセッササイクル数)，ロードストアユニットの BUSY サイクル数，分岐予測ミス回数を測定した結果を図 5 と図 6 に示す．

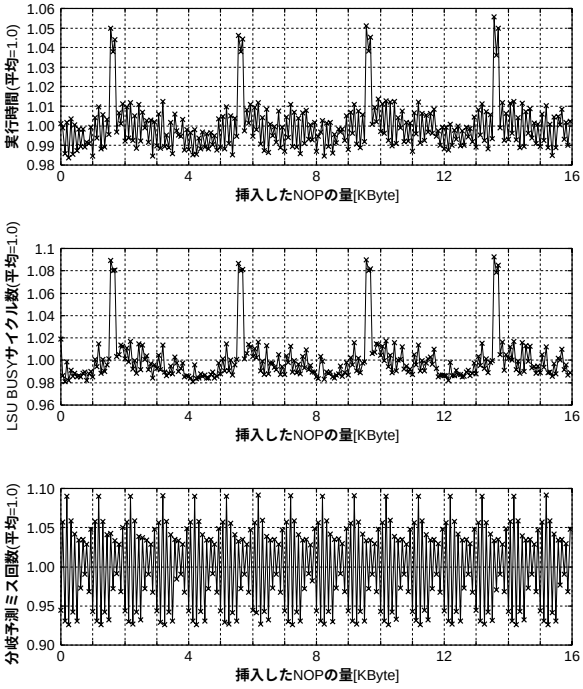

図 5: 対策前の変動 (POWER3-II)

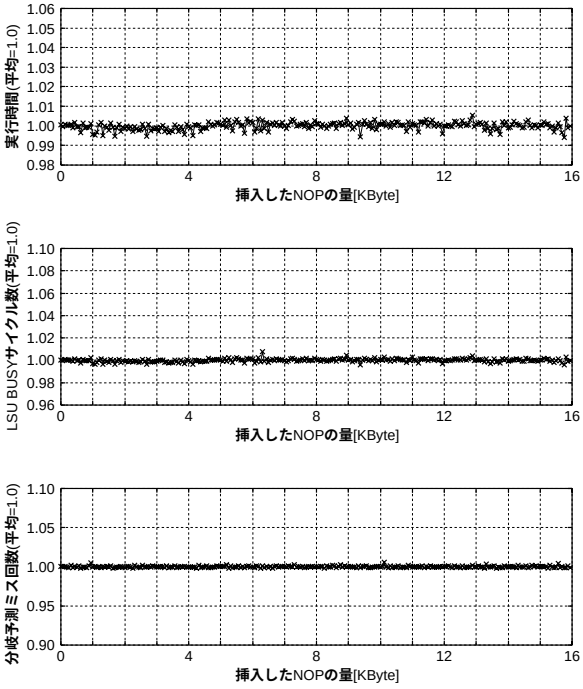

図 6: 対策後の変動 (POWER3-II)

全ての測定値は nop を挿入した全ケースの平均値からの相対値で表わしている．

対策前は，実行時間が最大 7.3%変動している．実行時間が平均値から 4%増加する箇所では，ロードストアユニットの BUSY サイクル数も増加していることと，実行時間の増加が 4KB 周期であることから，この箇所での実行時間の増加は store → load の擬似依存関係が原因と考えられる．一方，1〜2%程度の小さな性能変動は分岐予測ミス，あるいは，store → load の擬似依存だけの単純な原因ではなく，測定時の誤差も含めた複合的な原因によるものと考えられる．分岐予測ミスを原因とする実行時間の変動が小さいのは，分岐予測ミス数が少ないことと，POWER3-II プロセッサの分岐予測ミス時のペナルティが 3cyc と小さいためであると考えられる．

これに対して，対策後は実行時間の変動が最大 1.1%にまで抑えられており，提案する対策方法が有効であることが確認できる．また，ロードストアユニットの BUSY サイクル数，分岐予測ミス回数の変動も抑えられている．

4.2 POWER4 プロセッサでの評価

プロセッサによって，性能変動の原因となる高速化機構は異なる．そこで，POWER4 プロセッサについても性能変動と対策方法の効果を定量的に評価するため，対策前のプログラムと，対策後のプログラムに対して，性能変動の原因を人為的に起こした際の実行時間を測定した．

POWER4 プロセッサ [3] は，64KB，ダイレクトマップ方式の L1 命令キャッシュと 32KB，2way セットアソシアティブ方式の L1 データキャッシュの構成であるため，命令キャッシュの競合ミスによる 64KB 周期での性能変動，データキャッシュの競合ミスによる 16KB 周期での性能変動が起こりうる．しかしながら，L1 命令キャッシュミス回数と L1 データキャッシュミス回数は同時に測定できないため，連想度が小さく，性能変動への影響が大きいと考えられる L1 命令キャッシュミス回数を測定した．

性能変動の原因を擬似的に発生させるための nop の量は，L1 命令キャッシュにおける競合ミスの周期が 64KB で起きると考え，1KByte ごとに 0〜256KByte とした．

測定は POWER4 プロセッサを搭載した日立 EP8000 630 model 6C4 上で行なった．OS は AIX 5L である．

対策前と対策後の実行時間と L1 命令キャッシュミス回数の測定結果を図 7 と図 8 に示す．

対策前は，実行時間が最大 9.3%変動している．実行時間が平均値から 4〜6%増加する箇所では，L1 命令キャッシュミス回数が増加していることと，その際の周期が 64KB であることから，この箇所での実行時間の増加は L1 命令キャッシュミスが原因であると考えられる．一方，L1 命令キャッシュミスが増加し

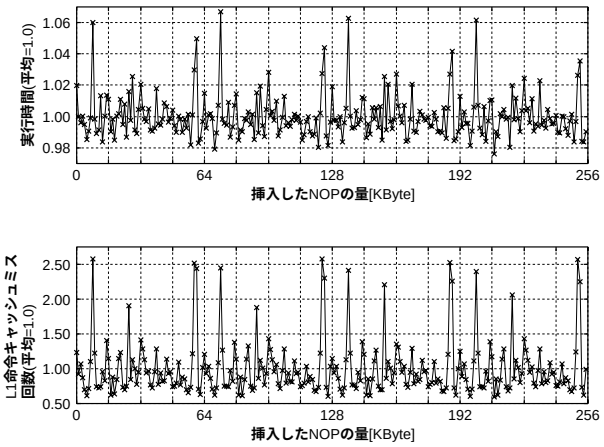

図 7: 対策前の変動 (POWER4)

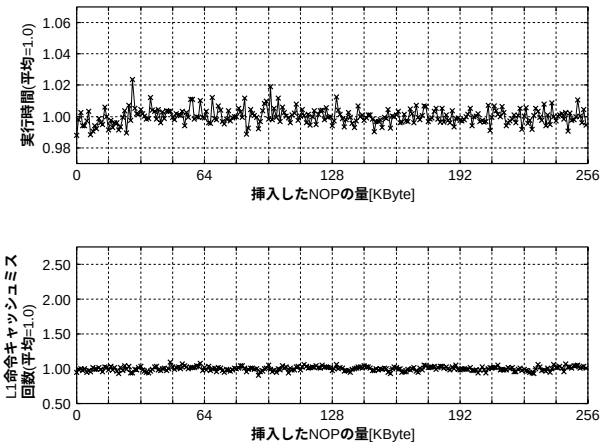

図 8: 対策後の変動 (POWER4)

ているのにもかかわらず，実行時間が増加していない箇所もある．実行時間が増加していない原因は調査中である．

これに対して，対策後は実行時間の変動が最大 3.6%程度に抑えられており，提案する対策方法が有効であることが確認できる．また，L1 命令キャッシュミス回数の変動でも対策前は 4.4 倍あったものが 1.2 倍にまで抑えられており，対策の効果が確認できる．

5 まとめ

高性能プロセッサはプログラムを変更した際に命令及びデータが配置されるメモリアドレスが変化することにより，変更を行なっていない箇所についても性能が変動するいう問題がある．この問題に対して，本報告書では性能変動の原因を明らかにし，対策方法を提案とその評価を行なった．

これまで知られているキャッシュメモリや分岐予測機構の他に store → load の擬似依存関係によっても性能変動が起きる．store → load の擬似依存関係のように，一般には公開されていない高速化機構も性能変動の原因となり得るため，性能変動の原因と性能低下を起こしている箇所を特定し，対策することは難しい．

そこで，プログラムの変更時の性能変動を防ぐ一手法として，使用頻度を考慮したメモリ配置とスペーサーの挿入を併用する方法を示した．本手法では，スペーサー挿入箇所を減らすことができるため，スペーサーがキャッシュされることによる性能低下を抑えつつ，性能変動の原因となる相対的なアドレス変化が起きる確率を減らすことができる．

本手法による性能変動の抑制効果を定量的に評価した．本手法を適用することで実行時間の変動を POWER3-II プロセッサでは 7.3%から 1.1%に，POWER4 プロセッサでは 9.3%から 3.6%に抑制できることを確認し，本手法が性能変動の抑制に有効であることを示した．

参考文献

[1] 高村明裕, 三木良雄. オブジェクトコードの配置アドレス改善によるプログラム高速化. 情処研報, Vol. 2003, No. 10, pp. 31–36, January 2003.

[2] F. P. O'Connell and S. W. White. Power3 : The next generation of powerpc processors. *IBM Journal of Research and Development*, Vol. 44, No. 6, pp. 873–884, November 2000.

[3] J. M. Tendler, J. S. Dodson, Jr. J. S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, Vol. 46, No. 1, pp. 5–25, January 2002.