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

2005－ARC－164 (16)
2005／8／4

割込みによるマイクロプロセッサの最悪性能予測

小西昌裕[†] 中田尚[†]
津邑公暁[†] 中島浩[†]

本論文では，命令の out-of-order 実行を行うプロセッサにおいて，割込みによるプリエンプションに起因する性能悪化の最大値を高速に取得する方法を論ずる．キャッシュや命令パイプライン機構を搭載するマイクロプロセッサでは，実効的な性能悪化量を得るためには割込みの実行をシミュレートする必要があるが，割込みが発生しうるすべての箇所においてこれを行えばその実行時間は膨大なものになる．そこで，割込みが発生する場所を変えた場合とプロセッサ状態の比較を行い，重複する実行であると判断した場合はその実行を省略することで実行時間の大幅な 短縮を図った．プラットフォームとして SimpleScalar を用いて，命令パイプラインとキャッシュメモリについてプロセッサ状態の比較による実行の省略を行った．その結果，連続する 10 万サイクルの割込み候補点について，各サイクルにおける割込みによる性能悪化量を状態比較を行わない場合に比べおよそ 38 分の 1 の処理時間で求めることができた．

An Estimation of the Worst-Case Performance caused by Interruptions for Microprocessors

Masahiro Konishi,[†] Takashi Nakada,[†] Tomoaki Tsumura[†]
and Hiroshi Nakashima[†]

In this paper, we describe a method to estimate the worst-case performance degradation by preemption of programs executed on out-of-order microprocessors. For modern microprocessors with complicated mechanisms, such as pipeline or cache memory, the only way for the precise estimation is to simulate the program repeatedly varying interruption points inserted in it. Since the straightforward implementation of this method requires a huge amount of execution time, we devised a fast measurement mechanism in which an interrupted execution is simulated *differentially*. That is, the simulation with an interruption at a time t completes when the machine state matches that derived from the execution interrupted at $t-1$. We implemented a SimpleScalar-based simulator with the differential interrupted execution. We evaluated the implementation using a 9-Queen solver program to find our simulator measures performance of a series of 100,000 interruptions 38 times as fast as the straightforward iterative simulation of interrupted execution.

1. はじめに

自動車のエンジン制御などといった組込みシステムにおいては，割込みにより生じるプリエンプションがもたらす性能悪化が，システムの性能見積りの際に重要な意味を持つ．このような組込みシステムでは，ある処理が定められた時間内に終了することが必要となることが多い．つまり，どのようなタイミングで割込みが発生した場合でも，その時間を越えないようにシステムを構成する必要がある．

一方，昨今のマイクロプロセッサは命令パイプライン，キャッシュ，分岐予測器などを搭載している．このため割込みの発生が，命令パイプラインの流れや命令の実行順序を大幅に乱すとともに，キャッシュおよび分岐予測器の状態にも変化を起こしうる．よって，割込みが性能悪化に及ぼす影響は，そのタイミングに大きく左右される．こういった理由から，in-order 実行などの単純な仮定を設ける場合以外では，割込みによる厳密な最悪性能悪化の予測は非常に困難である．

本稿では，さまざまなタイミングで実際に割込みを発生させてプログラムを実行し，得られた各性能悪化量を比較することで最悪性能を求める手法を提案する．しかし単純に各性能悪化量を測定すると，膨大な実行時間を要する．そこで，個々の割込み発生点に対する実行におけるプロセッサ状態を保存・比較し，その違いが命令の実行に影響を与えない間，あるいはプロセッサ状態が完全に一致した時点以降の処理を省略

† 豊橋技術科学大学
Toyohashi University of Technology

することで，最悪性能予測にかかる時間を大幅に削減する方法について論ずる．

以下，2章では割込みによるプリエンプションの影響と状態一致による実行省略の基本的な考え方を説明する．3章ではそのような実行省略を行うための具体的設計を述べる．4章では，実際に最悪性能予測を行うシミュレータの実装を，また5章ではその性能評価について，それぞれ述べる．最後に6章で本論文を総括し，今後の課題について概観する．

2. 最悪性能予測

この章では，プリエンプションによる性能の悪化の詳細とそれを求める手段，その過程において発生する不要な処理を省略する方法について述べる．

2.1 プリエンプションと性能悪化

一般的に，命令パイプラインを備えたマイクロプロセッサでは，割込みが発生した際にパイプラインに投入されている未完了の命令をどうするかが問題となる．パイプラインをフラッシュすることで未完了の命令を一旦破棄してしまうのが一般的な対処法であるが，この結果未完了命令の再実行が必要となるため性能が悪化する．また，キャッシュを備えている場合は，割込みやそれに起因するプリエンプションによってキャッシュに存在しているエントリの有効性が減少する．具体的には，割込みハンドラやプリエンプトしたプロセスがキャッシュメモリのエントリを置き換えてしまい，割込み元に処理が戻ってきたときには有効なエントリは減少している．

以上のように，プリエンプションはプログラムの性能に悪影響を及ぼすといえる．

2.2 プリエンプションの影響

プログラムを実行中にプリエンプションが発生することで，以下のようなことが起こると想定する．

- 命令パイプライン中のコミットされていない命令がフラッシュされる．フラッシュされた命令は実行完了していないので，再びプログラムに実行が戻ってきた際にパイプラインの最初のステージから実行がやり直される．
- キャッシュのエントリがすべて無効化される．実際にはいくつか有効なエントリが残る可能性もあるが，今回は最悪性能を求めるため，すべて無効化されるものとする．

なお，割込みハンドラやプリエンプトしたプロセスでどのような処理が行われるかは考えず，割込みの発生による影響は上で述べた2点のみとする．またTLBや分岐予測器への影響についてはキャッシュと同様に考えることができるので，本論文では議論を省略する．

2.3 最悪性能の検出方法

1章で述べたように，実際に各割込み候補点で割込みを起こすことで，割込みによる性能悪化量を検出する．従って，割込みによる最悪性能予測は以下のようにして行われることになる．

(1) 割込み候補点に到達するまでプログラムを実行する．
(2) 割込み候補点に到達したら割込み発生をシミュレートする．
(3) プログラムを最後まで実行する．
(4) 実行終了までに要した総CPUサイクル数を出力する．

これをすべての割込み候補点に対して実行すれば，割込み発生による最悪性能量が算出できることになる．しかしこの方法では，プログラムを割込み候補点の数だけ繰り返し実行することになるため，膨大な時間が必要となり非現実的である．そこで，次に述べるような状態比較による実行の省略を行うことで，総実行時間を削減する．

2.4 状態一致による実行の省略

プリエンプションが発生した直後のプロセッサ状態は，発生しなかった場合と異なっている可能性があるが，プログラムの実行を進めて行くにつれてこの差は小さくなり，やがては状態が一致すると考えられる．

プリエンプション発生後にプログラムの実行を継続し，ある地点においてプロセッサ状態がプリエンプション未発生の場合と等しくなったとすれば，これ以降もプロセッサ状態はすべて等しいままであると考えられる．従って，これ以降の実行に必要なCPUサイクル数はプリエンプション未発生時と等しくなると考えられ，プリエンプションによる割込みがもたらした性能悪化量は容易に算出することができる．つまり，ある地点Aでプリエンプションが発生した場合の性能悪化は，以下のようにして求められる．

(1) プリエンプションが発生しない場合のA地点以降のプロセッサ状態をすべて保存する．
(2) A地点でプリエンプションを発生させ，プロセッサ状態をプリエンプション未発生時のものと比較しながら実行を継続する．
(3) プロセッサ状態が一致したら，A地点から状態が一致した地点までに要した時間の差をとる．この差が性能悪化量である．

この様子を図1に示す．

プロセッサ状態の一致がプログラム全体の実行時間に比べてごく短い時間で起こるなら，この手法により

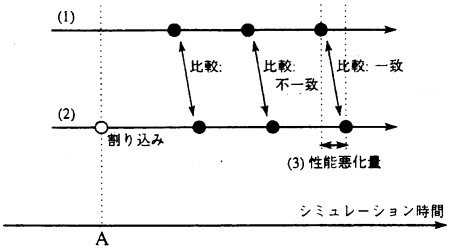


図 1　状態一致による実行の省略

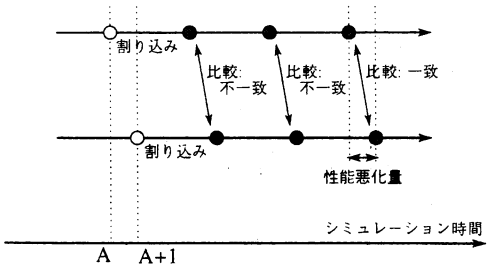


図 2　比較対象の変更

性能悪化量を本来必要な時間に比べてごく短い時間で検出することができる．

3. 設　　計

この章では，具体的にどのような処理を行うことで最悪性能を求め，不要な処理を省略するのかを述べる．

3.1 パイプラインの収束による一致

命令パイプラインはキャッシュに比べ，プリエンプション発生後により早期に収束する[1]．これは，パイプラインが保持しうるデータ量がキャッシュのそれと比べてごく小さいこと，分岐予測ミスが発生すると，投機的に実行されていたミスパスの命令がすべてフラッシュされ，パイプラインが空に近い状態になることが原因である．

3.2 キャッシュの比較

プリエンプションによってキャッシュがすべて無効化されると，それがプリエンプション未発生時のキャッシュの内容と一致するようになるまでには，最低でもプリエンプション前に有効だったキャッシュブロックをすべて有効にしなくてはならない．これはパイプラインと比べてかなり時間がかかると思われるので，プリエンプションが発生しない場合と発生した場合を比較するのではなく，図 2 のように，プリエンプションが地点 A で発生した場合と地点 A+1 で発生した場合の比較を考えることにする．ここで地点 A+1 とは地点 A の次の CPU サイクルとなる地点であり，例えば地点 A=100 サイクル目であれば A+1=101 サイクル目を指す．

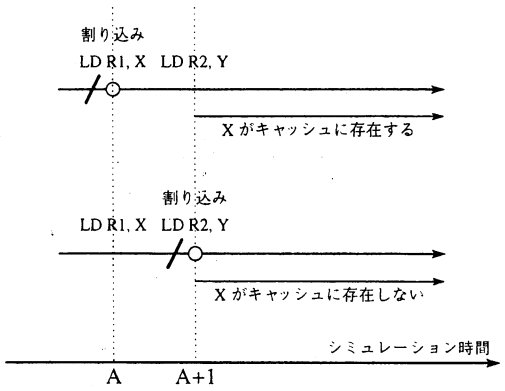


図 3　キャッシュの差異

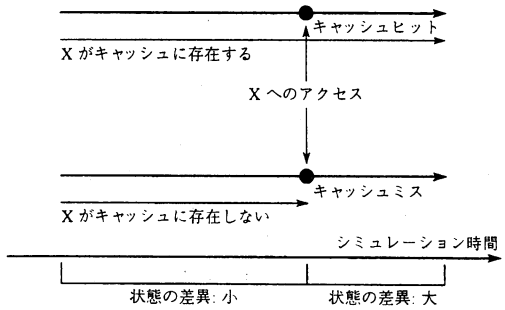
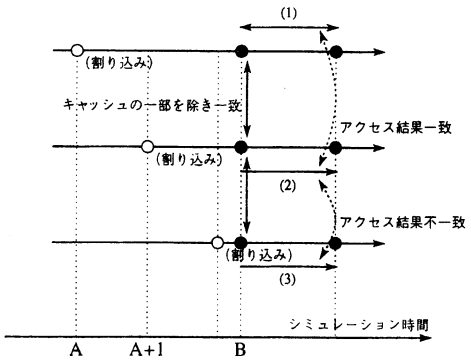


図 4　キャッシュの差異の発現

地点 A でプリエンプションが発生した場合と，地点 A+1 でプリエンプションが発生した場合の，キャッシュにおける違いは，地点 A で実行された命令によるメモリアクセスの結果がキャッシュに反映されているか否かである．図 3 の例では，プリエンプションが起こってから命令 LD R1，X が実行された場合と，LD R1，X が実行されてからプリエンプションが発生した場合を示している．前者では X の値がメモリからロードされた結果としてその内容がキャッシュに格納されているのに対し，後者では X の値は一旦キャッシュに取り込まれるものの，その後で発生したプリエンプションによってキャッシュから取り除かれる．

両者にはこの他に命令パイプラインの違いも存在する可能性があるが，3.1 節で述べたようにこれはごく早期に収束する．一方でキャッシュメモリの差異は，このキャッシュデータを利用するようなメモリアクセスが発生するまで潜在的に存在し続ける．図 4 は先



(1) A 地点でプリエンプトした場合のアクセスログを保存
(2) ログを用いてアクセス結果が同じになるか調べる
(3) アクセス結果が異なれば差異が発現したと判明する

図 5　メモリのアクセスログ

程述べた例の後，実行を継続している状態を示している．このときに X に対するメモリアクセスが発生すると，地点 A でプリエンプションが発生していた場合はキャッシュヒットとなるのに対し，地点 A+1 でプリエンプションが発生していた場合はキャッシュミスとなり，パイプラインが再び乱れることになる．しかしながらこれでキャッシュメモリの差異は解消され，パイプラインがじきに収束して 2 つの場合におけるプロセッサ状態は完全に一致する．

これで地点 A と地点 A+1 における性能悪化量の差が判明するので，次は地点 A+1 と地点 A+2，A+2 と A+3，…について同様の処理を行う．

しかし，キャッシュ状態の潜在的な差異を残したままパイプライン状態が一致した場合，その時点とキャッシュ状態の差異が発現する時点との距離は不明瞭である．もしこれがプログラム全体の実行時間に対して無視できないほど大きい場合，この手法により削減できる実行時間はごくわずかとなる．そこで以下では，このような場合に対応する手法について述べる．

キャッシュに潜在的な差異があったとしても，それが発現するまでの間は地点 A と地点 A+1 それぞれの場合におけるプロセッサ状態には前述の潜在的な差異を除いて何ら差はないものと考えられる．また，地点 A と地点 A+1 それぞれでプリエンプションが発生した場合のキャッシュの差は，実際にキャッシュメモリの比較を行うことで容易に判断することができる．そこで，図 5 のように，(1) 地点 A でプリエンプションが発生した場合の実行を先行させメモリアクセスログを保存し，(2) 地点 A と地点 A+1 それぞれの場合におけるキャッシュの差とメモリアクセスログを参照すれば，(3) この差異が発現したかどうかということも容易に確認できる．以上の点を利用し，図 6 のようにしてパイプラインの状態が一致してからキャッシュ状態の差異が発現するまでの実行を省略する．

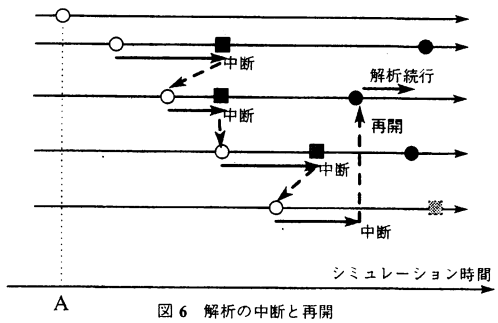


図 6　解析の中断と再開

まず，図中で下向き破線矢印で示しているように，地点 A と地点 A+1 のパイプラインの状態が一致した時点で一時的に解析を中断し待機状態として，次の地点 A+2 の解析に移ることとする．このとき地点 A の解析を先行させてメモリアクセスログを得ておく．このアクセスログを用いれば，待機状態とした地点についてそのキャッシュを実際に命令の実行を行わずにその先の状態へと更新することができる．待機状態となっている候補点のキャッシュの状態をどんどん先へ進めて行きながら次々に解析対象とする地点を進めていき，メモリアクセスログとの比較により差異が発現したことが判明した時点で待機状態となっていた解析を再開する．

このようにして，キャッシュを考慮した最悪性能予測を高速に行うことができる．

4. 実　　装

4.1 対象シミュレータ

今回，プラットフォームとして SimpleScalar Toolset 3.0c[2] の PISA target を選択した．SimpleScalar Toolset に含まれている *sim-outorder* を用いて，指定した区間の各サイクルを割込み候補点として最悪性能予測を行うプログラムを実装した．

4.2 比較対象の分離

最初の割込み候補点で割込みを発生させた後のプロセッサ状態と，それ以降の割込み候補点で割込みを発生させた場合のプロセッサ状態を比較する必要がある．これを容易に実現するために，プログラムを 2 プロセスに分けて動作させることにした．親プロセスは最初の割込み発生候補地点で割込みを発生させたまま実行を継続する．子プロセスはその次以降の地点で割込み

を発生させ，親プロセスからプロセッサ状態を受け取り比較を行う．

4.3 プロセッサ状態の比較タイミング

割込み発生後，パイプラインの状態が一致するまで実行を継続する．このとき，どのようなタイミングでパイプラインの状態比較を行うかが問題となる．頻繁に比較を行えばその分実行時間は遅くなり，また長期間比較を行わなければ無駄な実行が行われることとなってしまう．ここで，3.1 節で述べた，分岐予測ミスが発生することでパイプラインは疎な状態になるという点に着目し，分岐予測ミスの発生をプロセッサ状態の比較タイミングとすることにした．

4.4 データ構造

親プロセスは最初の割込み候補点で割込みを発生させた後，分岐予測ミスが発生した時の命令パイプラインやキャッシュなどのプロセッサ状態，キャッシュアクセスログを保存していく必要がある．また子プロセスは，割込み後待機状態となった地点から処理を再開する機能や，次の割込み候補点まで実行を巻き戻す機能を持たなくてはならない．

まず子プロセス向けに，メモリやレジスタへの変更を追跡・記録することによって，プログラムの実行を自由に巻き戻したり先送りする機能を実装した．このとき命令パイプラインやキャッシュについては，巻き戻したい地点でスナップショットとしてその状態を保存しておくこととした．次に親プロセスについて，プロセッサ状態を保存し子プロセスに送信する機能を実装した．

4.5 記憶領域の削減

前述のように，実行中に保存しなければならないデータは多岐に渡る．これを分別なく行っていたのでは非現実的な量の記憶領域が必要となってしまうので，不要なデータを必要に応じて解放するなどの工夫を行った．

次に、キャッシュの状態保存をより小さいデータサイズで行うことを考えた．各割込み発生候補地点ごとに，待機状態となった時点でのキャッシュメモリの状態をすべて保存しなければならないが，それぞれのキャッシュの差異はごく小さいと考えられる．そこで，完全な状態で保存されているキャッシュメモリは一部だけとし，他はそのキャッシュに対する差分という形で状態を保存することとした．これにより，プログラムの動作に必要な記憶領域を削減することができる．また，メモリアクセスログからキャッシュヒット・ミスを判定する処理についても，各差分についてのみ処理を行えばよいので，処理時間の削減にも繋がることが期待できる．

CPU	Intel Pentium4 Xeon 2.8GHz (2CPU)
主記憶	3GBytes
OS	Vine Linux 3.1 (Linux 2.4.27)
シミュレータ	SimpleScalar 3.0c
対象プログラム	9-Queen
総 CPU サイクル	6,476,959
割込み候補区間	1,000,000 ～ 1,100,000

表 1 評価環境

命令発行幅		4
RUU		16
LSQ		8
メモリポート		2
機能ユニット	INT-ALU	4
	INT-MUL/DIV	1
	FP-ALU	4
	FP-MUL/DIV	1

表 2 プロセッサの設定

1 次命令キャッシュ	16KB / 32B ライン / 1-way
1 次データキャッシュ	16KB / 32B ライン / 4-way
2 次統合キャッシュ	256KB / 64B ライン / 4-way
命令 TLB	16 エントリ / 4-way
データ TLB	32 エントリ / 4-way

表 3 キャッシュの設定

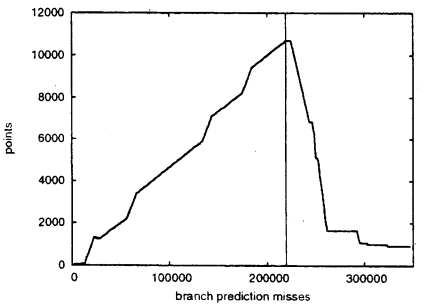


図 7 実行中の待機割込み候補点数の遷移

5. 評　　価

9-Queen 問題を解くプログラムに対し，最悪性能予測を行った．このときの環境は表 1，シミュレータの構成は表 2，表 3 の通りである．

このときの各割込み候補点の状態を図 7 に示す．ここで x 軸は分岐予測ミス回数，y 軸はその時点で待機状態となっている割込み候補点の数である．グラフ中の 210,000 付近の垂直線は，最後の割込み候補点である 1,100,000 サイクル目の処理を終えた地点を示している．1,100,000 サイクル目までは，待機状態になる

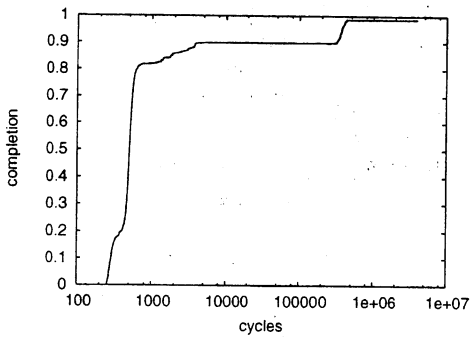


図 8　プロセッサ状態の一致に要する時間

割込み候補点が増えていくのと並行して，既に待機状態であった割込み候補点がキャッシュ中の差異を解消して，完全に他の状態と一致し消えていくため，待機状態となる割込み候補点の数はゆるやかに増加していく．1,100,000 サイクル目を終えると，待機状態のままとなっている各割込み候補点の解決のみを行うため，その数は急激に減少していくことがわかる．また，プログラム終了までキャッシュの状態が一致しなかった割り込み候補点も，全体の 100,000 個に対してわずかではあるものの存在することが確認できた．

また，割込みが起こってからキャッシュが完全に一致するまでに必要な時間と完全に一致した割込み候補点の全候補点に対する割合を図 **8** に示す．横軸は割込み発生からキャッシュが一致するまでに必要なサイクル数，縦軸は全体に対する一致完了率である．なお，ここにはプログラム終了までキャッシュが一致しなかった約 1,000 個ほどの割込み候補点は含まれていない．このグラフから，ほとんどの場合割込みが発生してから 10,000 サイクル以内にほぼ 9 割の候補点の処理が終了すること，残り 1 割もその後 1,000,000 サイクルほどで大部分が一致することがわかる．

今回の解析を行うのにかかった時間は 8 時間 46 分 39 秒，使用した記憶容量は約 900MB という結果になった．記憶領域のほとんどは待機状態の割込み候補点についてのデータ保持に用いられており，割込み候補点の終端である 1,100,000 サイクル以降の処理ではほとんど増加しなかった．

仮に，この解析をパイプラインの状態比較による実行省略なしに行ったとすると，各候補点ごとに残り 550 万～540 万サイクルの実行を行うことになる．650 万サイクルの実行に 14 秒かかっているので，550 万サイクルの実行には単純計算で 11.9 秒かかることになる．これを各候補点ごとに繰り返すわけであるから，$11.9 \times 100,000 = 1,190,000$ 秒，およそ 331 時間かかることになる．今回 9 時間程度で実行できたことから，実行省略による高速化という目的は十分達成できているということができる．

6. ま と め

今回，SimpleScalar を用いて，連続する特定区間においてプリエンプションが発生した場合の最悪性能を調査することができた．また，その実行に要する時間をプロセッサ状態の比較を行うことで，本手法を用いない場合に比べおよそ 38 分の 1 と大きく削減することに成功した．

本論文で述べた手法では，プリエンプションの発生回数は 1 回と限定している．これは最悪性能予測に本質的に存在する問題で，発生回数が複数回になればその発生パターンは指数的に増加していくために，計算量や必要な記憶容量もそれに比例する形で増加することになる．しかしキャッシュミス回数が最悪となるような複数の割込み発生箇所を求める研究[3] では，動的計画法を用いて極めて高速に複数回の割込みについてその最悪性能を求めることができるということがわかっている．この研究は in-order 実行を前提として行われたものであるが，この研究を今回の手法に適用すれば，out-of-order 実行についても複数回の割込みに対しその最悪性能を高速に求めることができると考えられる．

謝辞　本研究の一部は（株）半導体理工学研究センター（STARC）との共同研究「SpecC によるソフトウェア記述検証システム」，および文部科学省科学研究費補助金（基盤研究 (B)，研究課題番号 17300015，「高度情報機器開発のための高性能並列シミュレーションシステム」）による．

参 考 文 献

1) 高崎透, 中田尚, 津邑公暁, 中島浩: 時間軸分割並列化による高性能マイクロプロセッサシミュレーション, 先進的計算基盤システムシンポジウム SACSIS2005 論文集, pp. 339–348 (2005).
2) Austin, T., Larson, E. and Ernst, D.: SimpleScalar: An Infrastructure for Computer System Modeling, *Computer*, Vol. 35, No. 2, pp. 59–67 (2002).
3) Miyamoto, H., Iiyama, S., Tomiyama, H., Takada, H. and Nakashima, H.: An Efficient Search Algorithm of Worst-Case Cache Flush Timings, *Proc. 11th IEEE Conf. Embedded and Real-time Computing Systems and Applications* (2005).