重複実行省略を用いた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り込まれたプログラムの実行をシミュレートする必要があるが，これを割込みが発生しうるすべての箇所について行うと，それに要する時間は膨大なものになる．そこで，プロセッサ状態の比較を行うことで重複するシミュレーションを省略し，実行時間の大幅な短縮を図った．また，割込みの発生回数の増加によって可能な割込み箇所の組合せは指数的に増加するが，これを現実的な処理時間に抑えるアルゴリズムの検討を行った．プラットフォームとして SimpleScalar を用いて，割込み回数を 1 回に限定し，命令パイプラインやキャッシュメモリなど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対し，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 43 倍の速度で求めることができた．

Measuring Worst-case Performance of Microprocessor by Interruption with Omitting Redundant Execution

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. 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. Additionally, we show the outline of an algorithm to estimate the worst-case performance with two or more interrupts in a reasonable time rather than taking a huge time exponential with the number of interrupts. We evaluated our implementation with one interrupt using a 9-queen solver program to find our simulator measures performance of a series of 100,000 interruptions 43 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み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．このような組み込みシステムでは，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，その時間を超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おり，このようなプロセッサが組み込みシステムにも徐々に浸透しつつある．このようなプロセッサでは，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

† 豊橋技術科学大学
Toyohashi University of Technology
☆ 現在，株式会社 PFU
Presently with PFU Ltd.
☆☆ 現在，名古屋工業大学
Presently with Nagoya Institute of Technology
☆☆☆ 現在，京都大学
Presently with Kyoto University

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大きな変化を起こしうる．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．

割込みによる最悪性能予測を行う手法としては，プログラムにおけるキャッシュへの影響を静的に解析する手法[8),10),13)]が知られている．たとえば Lee らは，割込みが発生する各地点において，よくアクセスされるキャッシュブロックの数を求め，それにより割込みによる性能悪化の最悪値を予測している[8)]．また宮本らは，プログラムの in-order 実行によって得られたメモリアクセスログを解析し，キャッシュミス回数が最大となる割込みタイミングを求めている[9)]．しかしこれらの手法では，キャッシュへの悪影響を過大に評価したり[8)]，あるいはミス回数の見積りが正確であったとしても実際の最悪性能が求められなかったりする[9)]，といった問題点が存在する．

そこで本論文では，様々なタイミングで実際に割込みを発生させたプログラムの実行をシミュレートし，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．実際に割込みが発生した状況をシミュレートするため，この方法によって得られる性能の悪化量は既存手法とは異なり，シミュレータが正確である限り誤差を含まない．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，個々の割込みに対応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サイクル数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．なお最終的には任意回数の割込みを扱うことを目標とするが，本論文では 1 回の割込みについてのみ評価を行い，複数回の割込みについてはその実現への道筋を述べるにとどめる．

以下，2 章では割込みによる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，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．その後，6 章で割込みの回数を複数回へと拡張する方法について論じ，7 章で関連研究について述べた後，最後に 8 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，割込み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 割込みと性能悪化

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いったん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，割込みはその処理に時間を要するだけでなく，割り込まれたプログラムに固有の実行時間にも悪影響を及ぼすといえる．

2.2 割込みの影響

本論文では，プログラムを実行中に割込みが発生することで，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 や分岐予測器の BTB 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．分岐予測器が備える 2 ビット飽和カウンタについては後述する．

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測定することができる．すなわち，割込みによる最悪性能予測は以下のように行うことができる．

(1) 割込み候補点に到達するまでプログラム実行をシミュレートする．
(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) プログラムの実行を最後までシミュレートする．
(4) 実行終了までに要した総 CPU サイクル数を出力する．

上記の (1)〜(3) のシミュレーション過程を，ある割込み候補点に関する**スレッド**と呼ぶこととすると，すべての候補点に関するスレッドを実行してサイクル数の最大値を求めれば，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，割込み候補点の数に比例する膨大なシミュレーション時間を要すること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シミュレーションの省略を行うことで，総実行時間を削減する．

2.4 状態一致によるシミュレーションの省略

プロセッサは内部のメモリやレジスタなどに膨大な状態を保持し，それらはプログラムの実行に従って刻々と変化する．一般にプロセッサの内部状態は，レジスタファイル，主記憶などのアーキテクチャ状態と，命令パイプラインやキャッシュなどの非アーキテクチャ状態に分けることができる．ここで前節で述べた方針によるシミュレーションでは，割込みの前後でアーキテクチャ状態は変化しないため，割込みの影響は非アーキテクチャ状態に対してのみ生じると考えることができる．そこで以下では，これらの非アーキテクチャ状態を単に**プロセッサ状態**と呼び，また特に命令パイプラインとキャッシュの状態をそれぞれ**パイプライン状態**，**キャッシュ状態**と呼ぶ．また割込みの有無や割込み候補点の位置が異なる 2 つのスレッドについて，両者のアーキテクチャ状態が一致しているような各時点，具体的には両者においてコミットされた命令数が等しいような各時点の中から，適当に選択したものを（プロセッサ状態の）**比較点**と呼ぶ．

割込みが発生した直後のプロセッサ状態は，一般に発生しなかった場合とは大きく異なる．しかし命令パイプラインやキャッシュは，後述するように初期状態によらず一定の状態に収束する傾向を持つため，プログラムの実行を進めていくにつれて両者の差は小さくなり，やがてプロセッサ状態は一致すると考えられる．

一方，ある割込み候補点に関するスレッドのプロセッサ状態が，ある比較点において割込みなしのスレッドのものと等しくなったとすれば，この比較点以降の両者のプロセッサ状態はつねに一致することが保証される．この結果，この比較点からプログラムの終了までに要する両者の CPU サイクル数が一致することも保証され，割込みがもたらした性能悪化量を容易に算出することができる．つまり，ある割込み候補点 A に関する性能悪化量は，以下のようにして求められる（**図 1**）．

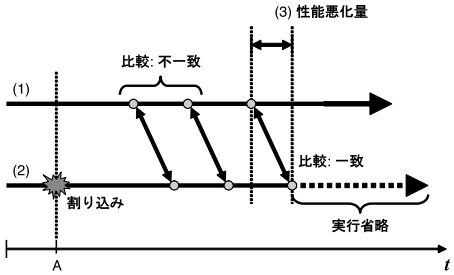


図 1 プロセッサ状態一致によるシミュレーションの省略
Fig. 1 Omitting simulation when two threads have the same processor states.

(1) 割込みが発生しないスレッドを実行し，候補点 A 以降のすべての比較点におけるプロセッサ状態を保存する．
(2) 候補点 A に関するスレッドを，各比較点でプロセッサ状態を割込みなしのスレッドのものと比較しながら実行する．
(3) プロセッサ状態が一致したら，候補点 A から状態が一致した比較点までに要したサイクル数の差をとる．この差が性能悪化量である．

プロセッサ状態がプログラム全体の実行時間に比べてごく短い時間で一致するならば，この手法により性能悪化量を本来必要な時間に比べてごく短い時間で算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，不要なシミュレーションを省略するのかを述べる．

3.1 パイプライン状態の一致

パイプライン状態は割込み発生後，比較的短時間で割込みなしのものと一致することが知られている[12]．これは，命令パイプラインが保持しうるデータ量がキャッシュなどと比べてごく小さいこと，および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 キャッシュ状態の比較

キャッシュの特定のセットの状態は，連想度に等しい数の異なるブロックに対するアクセスが生じると，それ以前の状態にかかわらず一定の状態に収束する．

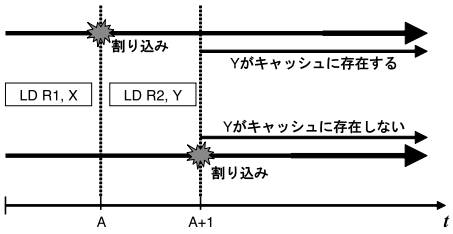


図 2 キャッシュの差異
Fig. 2 Difference in caches.

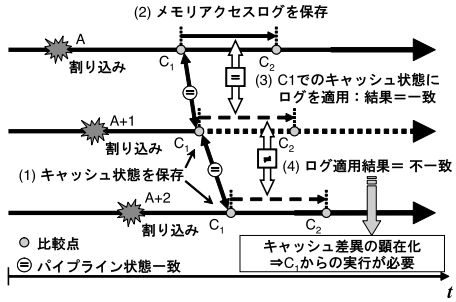


図 3 メモリのアクセスログ
Fig. 3 Memory access log.

したがって様々なブロックが高頻度でアクセスされるセットについては，割込みによる無効化後も比較的短時間で割込みなしの状態に一致することが期待できる．しかしキャッシュ全体を考えると，すべてのセットについてこのようなアクセスが生じるまでには，パイプライン状態の一致よりも長い時間がかかるものと予想される．

そこで，割込みなしと割込みありのスレッドのキャッシュ状態を比較するのではなく，割込み候補点 A に関するスレッド（以下「スレッド A」）と，A の次の CPU サイクルの割込み候補点 A+1 に関するスレッド A+1 のキャッシュ状態の比較を考える．スレッド A とスレッド A+1 の，割込み発生直後のキャッシュ状態の違いは，候補点 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 2 の例では，割込みが起こってからロード命令 `LD R2, Y` が実行された場合と，`LD R2, Y` が実行されてから割込みが発生した場合を示している．前者では `Y` の値がデータキャッシュに格納されているのに対し，後者では `Y` の値はいったんキャッシュに取り込まれるものの，その後で発生した割込みによってキャッシュから取り除かれる．またロード命令自身が命令キャッシュに存在するか否かについても同様である．スレッド A と A+1 の間にはこのほかに，当然パイプライン状態の違いも存在するが，前節で述べたようにこれはごく早期に一致する．一方でキャッシュ状態の差異は，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 2 の例で示した状態のままシミュレーションを継続したとき，`Y` に対するメモリアクセス（あるいは当該ロード命令の実行）が行われると，スレッド A ではキャッシュヒットとなるのに対しスレッド A+1 ではキャッシュミスとなり，両者のパイプライン状態が再び不一致となる．しかしながらこれでキャッシュ状態の差異は解消され，パイプラインはその後短時間で一致して，両者のプロセッサ状態は完全に一致する．この結果，実行 A と A+1 の性能差が判明するので，次は実行 A+1 と 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，いったんパイプライン状態が一致してからキャッシュ状態の差異によって再び不一致となるまでのスレッド実行を省略する手法について述べる．

キャッシュ状態に潜在的な差異があったとしても，差異がある部分にアクセスして顕在化するまでの間は，スレッド A と A+1 のプロセッサ状態は問題となる潜在的な差異を除いて一致している．また両者の割込み直後のキャッシュ状態の差は，実際にキャッシュメモリの比較を行うことで容易に求めることができる．そこで，**図 3** に示すように，キャッシュ状態の差異が顕在化するかどうかを，メモリアクセスログを用いて調べることができる．

(1) スレッド A と比較点 C_1 でパイプライン状態が一致したスレッド A+1 と A+2 の，C_1 におけるキャッシュ状態を保存する．
(2) スレッド A ではメモリアクセスが生じるたびに，そのアクセスログ（アドレス，操作，結果）を保存する．
(3) C_1 で保存したスレッド A+1 のキャッシュ状態に (2) のアクセスログを適用する．すなわち記録されたアドレスと操作に基づくキャッシュのみのシミュレーションを行い，その結果（ヒット/ミス，遅延）をログに記録されたものと比較する．すべてのアクセスについて結果が一致すれば，次の比較点 C_2 でのスレッド A+1 のパイプライン状態は A のものと等しいことが

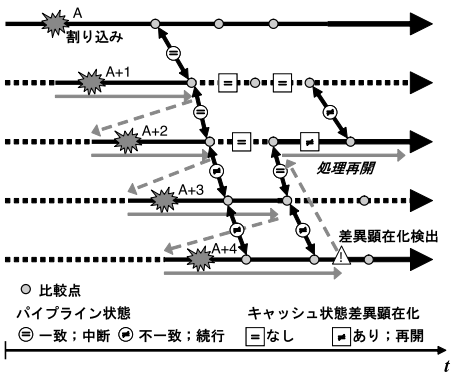


図 4 スレッドの中断と再開
Fig. 4 Suspension and resumption of threads.

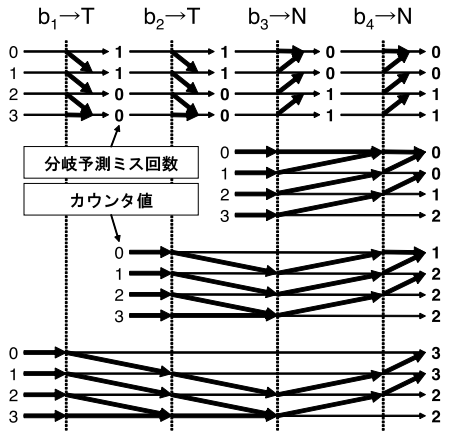


図 5 分岐予測器の最悪性能解析
Fig. 5 Analysis of worst-case performance of branch predictor.

保証される．またログの適用によって C_2 におけるスレッド A+1 のキャッシュ状態を求めることができる．

(4) (3) と同様にスレッド A+2 のキャッシュ状態にアクセスログを適用した結果，いずれかのアクセスの結果が異なれば，C_1 から C_2 に至る過程で A+2 のパイプライン状態が A と一致しなくなる．このようにキャッシュ状態の差異が顕在化した場合は，スレッド A+2 の C_1 からのシミュレーションが必要となる．

以上に基づき，**図 4** のようにしてパイプライン状態が一致してからキャッシュ状態の差異が顕在化するまでのシミュレーションを省略する．まず，図中の下向き破線矢印で示しているように，スレッド A と A+1 のパイプライン状態がある比較点で一致すると，スレッド A+1 を一時的に中断して待機状態とし，次のスレッド A+2 に移ることとする．なおスレッド A は A+1 以降よりもつねに先行させ，各比較点間でのメモリアクセスログを得ておく．またスレッド A+2 やそれ以降のスレッドが，スレッド A+1 を中断した比較点を越えて行われる際に，このアクセスログをスレッド A+1 のキャッシュ状態に適用してスレッド A との差異顕在化の検出と状態更新を行う．その後あるスレッド（図ではスレッド A+4）の実行過程で，待機状態のスレッド（図ではスレッド A+2）のキャッシュ状態の差異へのアクセスがログの適用で検出されると，実行中であったもの（A+4）を中止して待機状態のスレッド（A+2）を直前の比較点から再開する．

なおスレッドが待機中である間のサイクル数，たとえば図のスレッド A+2 の破線部分のサイクル数は，スレッド A の相当部分のサイクル数に等しいことが保証される．したがって待機中のシミュレーションを省略しても，スレッドの全実行サイクル数を正しく算出することができる．

3.3 分岐予測器の最悪性能

割込みによって 2 ビットカウンタ分岐予測器が受ける影響は，基本的にはキャッシュと同様であるが，分岐予測器特有の問題点として，割込み後に最悪性能をもたらすような状態を容易に決定できないということがあげられる．分岐予測器が持つ 2 ビット飽和カウンタは 0 から 3 のいずれかの値を持つ．最悪性能予測を行うためには，割込み発生後にこのカウンタの値を今後最も分岐予測ミスが多くなる値に設定する必要があるが，実際にプログラムの実行を進めなければ，どの値が最悪性能をもたらすかが決定できない．

この問題を解決するため，最悪性能予測を行うのに先立って，in-order 実行の命令トレースを用いた分岐予測器の解析を行うこととした．全条件分岐命令とその結果から，全割込み候補点について，そこで割込みが発生した場合における分岐予測器にとっての最悪状態を調べる．そして最悪性能予測を行う際，割込み後の分岐予測器の状態として，あらかじめ調べておいた最悪状態を適用する．以上のようにして分岐予測器の最悪性能を決定できる．

図 5 の例を用いて説明する．今，図に示したような分岐結果（T が taken，N が not-taken）となる 4 つの分岐命令 $b_1, \ldots, b_4$ がある．まずそれぞれの分岐命令について，2 ビット飽和カウンタが 0 から 3 それぞれの値だった場合についての分岐予測ミス回数および分岐命令実行後のカウンタの値を調べる．図中で矢印の後ろに記されている数がミス回数である．次に後ろから順に選択する分岐命令を増やしながら調査結果を連結していく．図では $b_3 - b_4$，$b_2 - b_3 - b_4$ という

ように連結を行っており，それぞれ b_3 の直前，b_2 の直前で割込みが発生した場合の最悪分岐予測ミス回数を調べている．たとえば $b_2 - b_3 - b_4$ について調べるときには $b_3 - b_4$ の調査は終了しているので，b_2 の結果をそれに連結するだけでよい．ここではカウンタの値が 1，2，3 のとき最悪となり，その場合の分岐予測ミス回数は 2 回である．またこの結果をさらに b_1 に適用すると，カウンタ値が 0 または 1 のときにミス回数が最大値の 3 になることが分かる．

このようにして，あらゆる場所で割込みが発生した場合の分岐予測器の最悪状態を調べることができる．

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 [1] 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．割込みの発生回数は 1 回とした．

4.2 実行プロセスの分離

最初の割込み候補点に関するスレッドを，**第 1 スレッド**と呼ぶ．2 番目以降の割込み候補点に関するスレッドの実行では，各々のプロセッサ状態と第 1 スレッドのプロセッサ状態と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．親プロセスは第 1 スレッドの実行のみを担当し，子プロセスはその他のスレッドを担当する．また子プロセスは親プロセスから第 1 スレッドのプロセッサ状態を受け取り，実行しているスレッドのプロセッサ状態との比較を行う．

4.3 プロセッサ状態の比較点

各スレッドは，第 1 スレッドとパイプライン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状態の比較を行うかが問題となる．頻繁に比較を行えばそのオーバヘッドによる実行時間を要し，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．そこで，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点とすることにした．

4.4 スレッド実行の制御

親プロセスが担当する第 1 スレッドの実行中は，比較点である分岐予測ミスの発生点で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と，キャッシュアクセスログを保存する．またこれらは子プロセスへプロセス間通信により送られ，子プロセスで実行するスレッドでのプロセッサ状態の比較に用いられる．

また子プロセスでのスレッド実行については，以下の機能を実装した．

(1) 第 1 スレッドのプロセッサ状態と自身が担当するスレッドのプロセッサ状態との比較．
(2) パイプライン状態が一致した際のシミュレーション完了（キャッシュ状態も一致）あるいは待機状態への移行（キャッシュ状態は不一致）．
(3) 待機中スレッドのキャッシュへのメモリアクセスログの適用と差異の検出．
(4) キャッシュ状態の差異が検出された場合の実行再開．
(5) スレッドが完了あるいは待機状態に移行した場合，次のスレッドの実行のためにアーキテクチャ状態とプロセッサ状態を，スレッドに対応する候補点まで巻き戻す機能．

なお (4) と (5) を実現するために，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，プロセッサのアーキテクチャ状態を割込み候補点や比較点へ，自由に巻き戻したり先送りする機能を実装した．パイプライン状態とキャッシュ状態については，巻き戻す可能性のある比較点でのスナップショットを保存しておくこととした．

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わたる．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，キャッシュ状態の保存をより小さいデータサイズで行うことを考えた．各スレッドごとに待機状態となった時点でのキャッシュ状態を保存しなければならないが，スレッド間のキャッシュ状態の差異はごく小さいと考えられる．そこで，完全に保存するキャッシュ状態は一部のもののみとし，他はその状態に対する差分という形で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アクセス結果の一致・不一致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つながることが期待できる．

5. 評　　価

9-Queen 問題を解くプログラムをワークロードとし，その総実行サイクル 6,476,959 中の中の 1,000,000〜

表 1 評価環境
Table 1 Environment of performance evaluation.

CPU	Intel Xeon 2.8 GHz（2CPU）
主記憶	3 GBytes
OS	Vine Linux 3.1 (Linux 2.4.31)
シミュレータ	SimpleScalar 3.0c

表 2 ターゲットマシンの構成
Table 2 Configuration of target machine.

命令発行幅	4
RUU	16
LSQ	8
メモリポート	2
INT-ALU	4
INT-MUL/DIV	1
1 次命令キャッシュ	16 KB / 32 B ライン / 1-way
1 次データキャッシュ	16 KB / 32 B ライン / 4-way
2 次統合キャッシュ	256 KB / 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
分岐予測器	2 ビット飽和カウンタ / 2,048 エントリ

1,100,000 サイクルの区間について，各サイクルを割込み候補点として最悪性能予測を行った．なお後述するように単純な方法での予測には膨大な時間を要し，また本研究の方法を用いてもかなり長い時間を要するため，本論文での性能評価は 1 例のみとした．したがって他のワークロード，特に組み込みシステムで用いられるアプリケーションによる評価は今後の課題であるが，9-Quees は時間的局所性と非局所性をあわせ持つため評価対象として適切であると考えられる．すなわち探索木の葉に近い部分は局所性が高いため割込みの悪影響が大きく現れ，根に近い部分や解発見時の出力部分はアクセス間隔が大きいためキャッシュ状態の一致に時間を要するという特徴がある．

評価の環境は**表 1**，ターゲットマシンの構成は**表 2** のとおりである．最悪性能予測の結果，最大の性能悪化量は 5,459 サイクル，最小は 5,123 サイクル，平均は 5,302 サイクルとなった．

まずスレッドごとの，実際にシミュレーションを行ったサイクル数を**図 6** に示す．横軸がシミュレーションを行ったサイクル数，縦軸があるサイクル数のシミュレーションを行ったスレッドの数である．なおこのグラフの範囲外に，約 12,000 サイクルのものと約 5,400,000 サイクルのものがそれぞれ 1 つずつ存在している．また 1 スレッドあたりのシミュレーションサイクル数の平均値は 791 であった．

この結果は，大部分のスレッドについて 1,000 サイクル未満のシミュレーションしか行う必要がなかったことを示している．本手法によるシミュレーションの省略を行わなかった場合，1 スレッドにつき少なくとも割込み候補点以降の 5,400,000 サイクル程度をシミュレートすることとなるので，サイクル数に関する限り 7,000 分の 1 程度に削減できたこととなる．

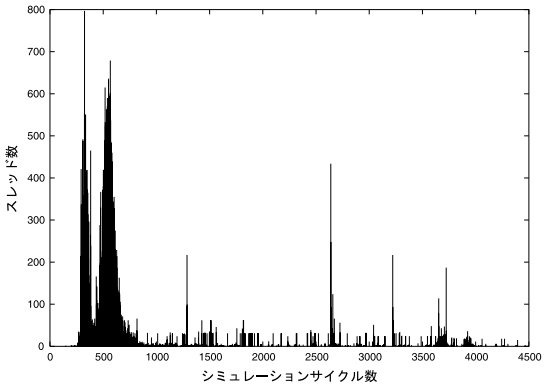


図 6 スレッドごとのシミュレーションサイクル数
Fig. 6 Simulated cycles for threads.

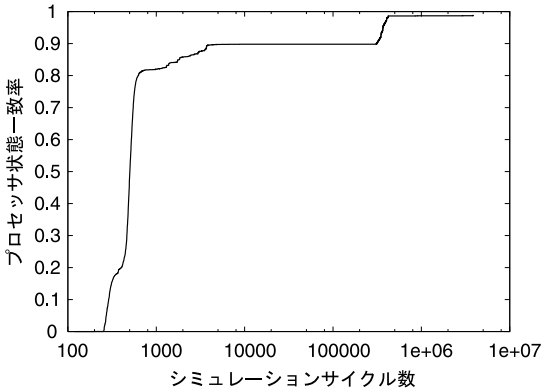


図 7 割込み後のサイクル数とプロセッサ状態の一致率
Fig. 7 Cycles after interruptions versus processor state agreement ratio.

このように大きな削減ができた理由を詳しく調べるために，割込み候補点から何サイクルをシミュレートすると，第 1 スレッドとプロセッサ状態が完全に一致するかを計測した．**図 7** は，あるサイクルまでに第 1 スレッドのプロセッサ状態と一致したスレッド数の累計を，全スレッドに占める比率で示したものである．なお，この図にはプログラム終了までキャッシュ状態が一致しなかったため，プロセッサ状態が完全に一致しなかった 1,171 個のスレッドは含まれていない．

このグラフから，仮にプロセッサ状態の完全一致のみをシミュレーションの省略条件とした場合，約 90%が 10,000 サイクル未満のシミュレーションを行えばよいのに対し，残りの 10%には 400,000 サイクル程度を要することが分かる．これに図示していない 1%程度

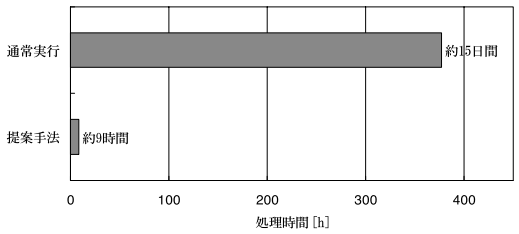


図 8 実行時間
Fig. 8 Execution time.

に 5,400,000 サイクル程度を要することを勘案すると，平均サイクル数は 40,000〜50,000 程度となり，前述の平均サイクル数 791 を大幅に上回ることとなる．このことから，3.2 節で述べたキャッシュの差異が顕在化するまでスレッドを待機状態とする手法が，シミュレーションサイクル数の削減に大きな効果をもたらすことが示された．

最後に，最悪性能予測を単純な方法，すなわち 2.3 節で述べたすべてのスレッドについて，対応する割込み候補点以降のシミュレーションを実施する方法（通常実行）による処理時間と，提案手法の処理時間を **図 8** に示す．前者の方法ではおよそ 15 日間という時間が必要であったのに対し，提案手法ではその 1/43 の約 9 時間で解析が終了していることから，実行省略による高速化は十分に達成できているということができる．

ただしこの 1/43 という比率と，シミュレートしたサイクル数の比である 1/7,000 との間には，かなり大きな開きがある．この事実は，パイプライン状態の比較や，待機状態のスレッドのキャッシュ状態の更新と差異の顕在化検出などのオーバヘッドが大きく，1 サイクルあたりのシミュレーション時間が SimpleScalar の 160 倍程度になっていることを意味している．この主な要因は，第 1 スレッドを担当する親プロセスと，その他のスレッドを担当する子プロセスの間のプロセス間通信であり，この問題を中心に性能改善を図ることが今後の課題となっている．

6. 複数回の割込み

本章では，これまで 1 回としてきた割込みの発生回数を複数回に拡張する方法について論ずる．なお，今後アルゴリズムの計算量を検討するにあたり，解析対象とするプログラムの総命令数を N，割込み回数を F で表す．また，特定の割込み候補点 F 個の組合せを**割込み発生パターン**と定義する．

6.1 動的計画法を用いた方法

最も単純な方法は，考えられる割込み発生パターンすべてを列挙し，それらすべてについて 3 章で述べた方法で性能悪化量を算出するというものである．しかし，この方法は計算量が $O(N^F)$ となってしまい，検討に値しない．

これよりもやや現実的な方法として，動的計画法を用いるものがある．まず対象プログラム中の i（$i > 0$）番目の命令がコミットされた直後に割込みが生じてから（以下 i での割込みという），j（$j > i$）番目の命令がコミットされるまでに要するクロック数を $C(i, j)$ とする．また $C(0, j)$ をプログラム実行開始から j 番目の命令がコミットされるまでに要するクロック数とすると，あらゆる j に関する $C(0, j)$ は割込みなしのシミュレーションを 1 回行うことによって求めることができる．

次にある $i-1$ とすべての $j \geq i$ について $C(i-1, j)$ が既知であるとする．また $i-1$ での割込みと i での割込みでパイプライン状態が一致していない区間を，当該区間内にコミットされた命令の番号で表現したものを $[h_i^1, t_i^1]$, $[h_i^2, t_i^2]$, ..., $[h_i^{n_i}, t_i^{n_i}]$ とすると，これらの区間中の $C(i, j)$（$h_i^k \leq j \leq t_i^k$）は，3 章で述べたようにこれらの区間だけをシミュレーションすることにより求めることができる．また区間外の $C(i, j)$（$t_i^k < j < h_i^{k+1}$）については

$$C(i, j) = C(i-1, j) + (C(i, t_i^k) - C(i-1, t_i^k))$$

となる．

ここで 5 章の評価結果から，パイプライン状態不一致区間の長さの総和 $\sum_{k=1}^{n_i}(t_i^k - h_i^k + 1)$ が総実行命令数 N によらない定数 K で抑えられるとすると，すなわち 3 章のアルゴリズムの時間計算量が $O(KN)$ であるとすると，あらゆる $C(i, j)$ を保持するためのデータ構造に要するメモリ量は $O(KN)$ となる．また任意の i と j について，このデータ構造を参照して $C(i, j)$ を求める手間はたかだか定数時間とすることができる．

一方 F 回の割込みによる最悪性能は，連続する 2 つの割込みに挟まれた区間の性能が他の区間とは独立に定まることから

$$\Gamma(i, f) = \max\left\{ \sum_{l=0}^{f} C(j_l, j_{l+1}) \,\middle|\, i = j_0 \leq j_1 \leq \cdots \leq j_f \leq j_{f+1} = N \right\}$$

としたとき $\Gamma(0, F)$ となる．ここで上式は

$$\Gamma(i, f) = \max_{i \leq j \leq N} \{C(i, j) + \Gamma(j, f-1)\}$$

と変形することができるので，文献 9) に示された手

法により $O(N^2F)$ の時間計算量で $\Gamma(0,F)$ を求めることができる．

さてこの $O(N^2F)$ の時間計算量は $O(N^F)$ に比べてはるかに優れてはいるが，N がプログラムの総実行命令数であって，たとえば 1 GIPS のプロセッサによる 1 秒間の実行で $N=10^9$ となることを考えると，$N^2=10^{18}$ に比例した計算時間はやはり非現実的なものとなってしまう．

6.2 最悪性能をもたらす割込み発生パターンの絞り込み

これまでに述べた方法では，いずれも割込み候補点としてプログラムのすべてを対象としていた．しかし，実際には，最悪性能をもたらすような割込み発生パターンは，各割込みが互いにある程度の間隔をおいていると考えられる．これは次のような理由からである．

割込みが発生すると，キャッシュや分岐予測器中にある「有用な情報」は失われてしまう．その後の実行はそれら「有用な情報」がないままで行われるため，割込みが発生しない場合と比べてキャッシュミスなどが頻発し，実行にかかる時間は増加する．そのような割込みの影響を乗り越え十分な時間実行を継続し，再びキャッシュメモリや分岐予測器が「有用な情報」で満たされるようになったときに割込みが再び発生すれば，これまでに起こったキャッシュミスなどの結果として得られた「有用な情報」は再び失われ，またキャッシュなどが空の状態から実行を行わなければならない．一方で，一度割込みが発生した直後に再び割込みが発生した場合，キャッシュメモリや分岐予測器から失われる「有用な情報」はごく少なく，それが発生しない場合に比べ性能の悪化量は少ない．以上から，最悪性能をもたらすような割込みは互いに距離を置いているといえる．

そこで，任意の回数の割込みを解析するにあたって，事前に in-order 実行の結果に基づく解析を行うことで，キャッシュや分岐予測器にとって最悪となるような割込み発生パターンを求め，それに基づき 3 章で述べた手法によるシミュレータを用いた解析を行うことを提案する．すなわち解析により求められる最悪の F 回割込み発生パターンを，前節と同様に命令番号で表記したものを $i_1,\ldots,i_F$ としたとき，各 i_k について $i\in S_k=(i-M/2,i+M/2]$ なる M 個の命令のコミット直後のタイミングを割込み候補点としてシミュレーションを行う．

ここで前節で述べたように，連続する 2 つの割込みに挟まれた区間の性能は他の区間と独立に定まるので，個々の k に対してすべての $i\in S_k$ と $j\in S_{k+1}$ についての $C(i,j)$ を求めれば，真の最悪割込み発生パターン $i_1^w,\ldots,i_F^w$ がすべての k について $i_k^w\in S_k$ である限り $\Gamma(0,F)$ を求めることができる．なお $\Gamma(0,F)$ を求めるための時間計算量は前節と同じ方法によれば $O(M^2F)$ となるが，M が十分小さく，たとえば 10,000 程度であれば現実的な時間で求めることができる．一方，ある k に関するすべての $i\in S_k$ と $j\in S_{k+1}$ について $C(i,j)$ を求めるには，割込み候補点をすべての $i\in S_k$ として M 個のスレッドのシミュレーションを $i_{k+1}+M/2$ まで行えばよく，5 章の評価結果からすべての k に関するシミュレーションに要する時間計算量は

$$O\left(\sum_k\left\{(i_{k+1}+M/2)-(i_k-M/2)\right\}\right)=O(N+MF)$$

となる．

さてこの手法が成立するためには，各候補点の相互干渉が存在しないこと，in-order 実行による解析時間が十分短いことの 2 つの条件が満たされなくてはならない．

まず最初の条件を検討する．命令パイプラインは前述のとおり早期に収束するため，ある地点で発生した割込みの影響がその次の割込み候補点に波及することはないと考えられる．一方キャッシュはより長い期間が必要となるが，数十万サイクル程度の時間が経過すればほとんどの差異は解消され，たかだか 1 個程度の差異が残るだけだと考えられる．このような差異が最悪性能に及ぼす影響はごくわずかなので，キャッシュについてもそれほど長くない期間で割込みの影響が消え，相互干渉は無視できる．

次に 2 番目の条件について検討する．in-order 実行による最悪性能の解析は大きく分けるとキャッシュの解析と分岐予測器の解析という 2 つに分けられる．したがって 2 番目の条件についてはこの 2 つについてそれぞれ別々に考える必要がある．キャッシュの解析については，動的計画法を改良した方法により $O(NF)$ の時間計算量で実現できることが明らかになっている[9]．一方で 2 ビットカウンタ分岐予測器については，$O(N^2F)$ よりも良いアルゴリズムは知られていなかったが，我々の研究により実効的に $O(NF)$ で求められることが明らかになっている[7]．

以上により，最悪性能をもたらす割込み発生パターン候補を in-order 実行で求めるための手間は $O(NF)$，これに基づき必要な $C(i,j)$ を求めるためのシミュレーションは $O(N+MF)$，この結果から最終的に $\Gamma(0,F)$

を求める手間は $O(M^2F)$ となる．したがって，M が 10,000 程度であれば現実的な時間で F 個の割込みに関する最悪性能を算出できると考えられる．

7. 関連研究

本研究の対象である，割込みによる性能劣化の最悪値の予測は，最悪実行時間（WCET: Worst Case Execution Time）解析[11] の一種ととらえることができる．WCET 解析には大きく分けて，プログラムを実行せずに静的に解析するものと，プログラムを何らかの方法で実行した結果を解析するものとがある．前者は主に，ある定義域で定められるプログラム入力の集合から，最悪実行時間をもたらすものを求めるものである．また求められた最悪入力から実際の最悪実行時間を求める方法としては，単純にプログラムを実行するもののほか，キャッシュ[5),6)]，分岐予測器[2),3)]，パイプライン[4)] の影響を加味するために，部分的にターゲットマシンのシミュレーションを行うものもある．

一方，割込みによる性能劣化の予測を完全に静的に行うことは困難であり，本研究と同様に何らかの実行に基づき解析する方法に限定される．また既存研究の多くは，割込みによる悪影響が顕著に生じるキャッシュに関するものである．たとえば Lee らは，プログラムの基本ブロックごとにキャッシュ中の「有用なデータ」の数を求め，それに基づき割込みによって失われる有用データの最悪値を算出する方法を提案している[8)]．この方法は複数回の割込みに対しても適用可能であるが，有用データ数が最大となる箇所がループ中にある場合，非常に悲観的な見積りをするという欠点がある．また Lee らの方法の変形として，プリエンプトしたプロセスによるキャッシュの汚染状況を勘案するものも提案されているが[10),13)]，最悪値の見積りが悲観的になるのは同様である．

一方宮本らは，メモリアクセストレースを解析して，キャッシュがこうむる割込みの悪影響をより厳密に解析する手法を提案している[9)]．この方法は，特定の入力データセットに対してキャッシュミスの最悪値をもたらす割込み発生パターンを正確に求めることができるという点で，Lee らの方法にはない特徴を持っている．また解析に要する時間計算量が，トレースの長さ N と割込みの数 F に対して実効的に $O(NF)$ であるということも，この研究の特質である．

さて，上記の割込みによって増加するキャッシュミスの最悪値を求める研究は，いずれも真の性能劣化を求めるものではない．すなわち，あるタイミングでの割込みがキャッシュミスの増分を最大化することが求められても，それによって生じる性能劣化は別途シミュレーションなどで求める必要がある．また Lee らの手法は最悪値の上限を与えるのみであって，その最悪値をもたらす割込み発生パターンを示すものではないので，シミュレーションで性能劣化を見積もること自体が困難である．またいずれの方法でも，キャッシュミスの増分が最大となる割込み発生パターンが最悪の性能劣化をもたらす保証はなく，安全な解析値が得られないことがありうる．さらに，分岐予測器などキャッシュ以外の機構への悪影響は考慮されておらず，これらを別途求めることができたとしても，すべての機構への悪影響の総合効果として生じる最悪性能を求めるのは困難である．

これらの既存研究に対して，本研究は 1 回の割込みに関する限り，それによる性能劣化を直接的かつ精密に求めることができるという点で優れている．複数回の割込みについては前章で述べたように既存研究の結果を活用することとなるが，解析的に求められた最悪の割込み発生パターンの周辺を効率的に幅広く探索する方法を提供することで，既存研究よりも高い精度の結果が得られるものと考えられる．一方，本研究の手法は長いシミュレーション時間を要することが問題となる．たとえば 5 章で用いたワークロードの解析には，本研究では 9 時間という長い時間を要するのに対し，宮本らの解析は同等のメモリアクセス数に対して数分で完了する．したがって本研究の方法を真に実用的なものとするために，処理時間の大幅な改善が今後の課題となっている．

8. まとめ

本論文では，ワークロードの実行中に生じる割込みがもたらす性能劣化の最悪値を，割込みが生じるタイミングを変化させながらシミュレーションすることにより，精密に求める方法を提案した．提案手法の特徴は，異なる割込み候補点に関するシミュレーション過程が，まったく同じ結果をもたらす区間が存在することに着目し，そのような区間のシミュレーションを省略することにある．この結果，単純に全割込み候補点に関するシミュレーションを行う方法と比べ，処理時間を約 1/43 に短縮するという高い効果が得られた．なおシミュレーションの省略は同一結果が保証される区間に対してのみ行うため，単純な方法とまったく同じ結果が得られることが保証される．

今後は，より実際的なプログラムを用いた場合でも同様の性能が得られるかを調査することが必要であり，同時に処理時間を短縮することも重要な課題となって

いる．また，本論文で行った実装では，割込みの発生回数は 1 回と限定している．これを複数回に拡張するための手法について検討を行ったが，キャッシュや分岐予測器の解析で求められる最悪の割込み発生パターン候補に対して，各候補点の周辺をどの程度の幅で探索すれば精密な結果が得られるかなど，検討すべき課題は数多く残されている．

謝辞 本研究の一部は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) Austin, T., Larson, E. and Ernst, D.: SimpleScalar: An Infrastructure for Computer System Modeling, *Computer*, Vol.35, No.2, pp.59–67 (2002).
2) Bate, I. and Reutemann, R.: Worst-Case Execution Time Analysis for Dynamic Branch Predictors, *ECRTS'04*, pp.215–222 (2004).
3) Colin, A. and Puaut, I.: Worst Case Execution Time Analysis for a Processor with Branch Prediction, *Real-Time Systems*, Vol.18, No.2/3, pp.249–274 (2000).
4) Engblom, J. and Ermedahl, A.: Pipeline Timing Analysis Using a Trace-Driven Simulator, *RTCSA'99*, pp.88–95 (1999).
5) Healy, C.A., Arnold, R.D., Mueller, F., Whalley, D.B. and Harmon, M.G.: Bounding Pipeline and Instruction Cache Performance, *IEEE Trans. Comput.*, Vol.49, No.1, pp.53–70 (1999).
6) Kim, S.-K., Min, S.L. and Ha, R.: Efficient Worst Case Timing Analysis of Data Caching, *RTAS'96*, pp.230–240 (1996).
7) 小西昌裕，中島 浩，中田 尚，津邑公暁，高田広章：分岐予測器の最悪フラッシュタイミングの効率的解析手法，情報処理学会研究報告 2006-EMB-001, pp.1–6 (2006).
8) Lee, C.G., et al.: Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling, *IEEE Trans. Comput.*, Vol.47, No.6, pp.700–713 (1998).
9) 宮本寛史，飯山真一，冨山宏之，高田広章，中島浩：キャッシュフラッシュの最悪タイミングの効率的な探索手法，情報処理学会論文誌：コンピューティングシステム，Vol.46, No.SIG 16 (ACS12), pp.85–94 (2005).
10) Negi, H.S., Mitra, T. and Roychoudhury, A.: Accurate Estimation of Cache-Related Preemption Delay, *CODES+ISSS 2003*, pp.201–206 (2003).
11) Puschner, P. and Burns, A.: A Review of Worst-Case Execution-Time Analysis, *Real-Time Systems*, Vol.18, No.2/3, pp.115–128 (2000).
12) 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速マイクロプロセッサシミュレーション，情報処理学会論文誌：コンピューティングシステム，Vol.46, No.SIG12 (ACS11), pp.84–97 (2005).
13) Tan, Y. and Mooney, V.: Integrated Intra- and Inter-Task Cache Analysis for Preemptive Multi-tasking Real-Time Systems, *SCOPES 2004*, LNCS 3199, pp.182–199 (2004).

(平成 18 年 1 月 27 日受付)
(平成 18 年 6 月 13 日採録)

小西 昌裕

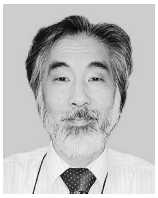
2006 年豊橋技術科学大学大学院工学研究科情報工学専攻修士課程修了．同年（株）PFU 入社．在学中はマイクロプロセッサシミュレータに関する研究に従事．

中田 尚（学生会員）

2004 年豊橋技術科学大学大学院工学研究科情報工学専攻修士課程修了．同年同大学院工学研究科電子・情報工学専攻博士後期課程入学．計算機アーキテクチャとシミュレーションに関する研究に従事．

津邑 公暁（正会員）

1998 年京都大学大学院工学研究科情報工学専攻修士課程修了．2001 年同大学院情報学研究科博士後期課程学修認定退学．同年同大学院経済学研究科助手．2004 年豊橋技術科学大学工学部助手．2006 年名古屋工業大学工学研究科助教授．博士（情報学）．計算機アーキテクチャ，並列処理応用，脳型情報処理等に関する研究に従事．日本神経回路学会，電子情報通信学会，ACM 各会員．



中島　浩（正会員）

1981 年京都大学大学院工学研究科情報工学専攻修士課程修了．同年三菱電機（株）入社．推論マシンの研究開発に従事．1992 年京都大学工学部助教授．1997 年豊橋技術科学大学教授．2006 年京都大学教授．並列計算機のアーキテクチャ等並列処理に関する研究に従事．工学博士．1988 年元岡賞，1993 年坂井記念特別賞受賞．情報処理学会計算機アーキテクチャ研究会主査，同論文誌：コンピューティングシステム編集委員長，同理事等を歴任．IEEE-CS，ACM，ALP，TUG 各会員．