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

2006－ARC－169 (20)
2006／8／2

CASによる最悪割込遅延解析の高速化

中島 浩[†,☆] 小西昌裕[†,☆☆] 中田尚[†]

本論文は、cycle accurate simulator (CAS) を用いた最悪割込性能解析法について、すでに我々が提案した方法を大幅に改善する手法について述べたものである．この方法は，個々の割込点に関するシミュレーションスレッドを全て実行するのではなく，スレッド間の「差分」だけを実行することで高速化することが特徴であり，この点については我々の従来手法を踏襲している．しかし具体的な実装法は大幅に改善されており，特に他のスレッドに実行を委譲して「休眠」状態にあるスレッドのサイクル数管理の手間を，二分木を用いることで従来の $O(N)$ から $O(\log N)$ に改善した効果が大きい．従来の実装法は，単純に全てのスレッドを完全に実行する方法で 15 日を要する解析を 9 時間に短縮するという効果を得ていたが，実装法改善により同じ解析を僅か 79 秒で完了するという，さらに高い効果が得られた．また SPEC CPU95 ベンチマークを用いた評価により，単純法では 200〜300 日を要する解析が 30 分以内で完了することも明らかになった．

Improvement of Worst Case Interruption Delay Analysis with CAS

HIROSHI NAKASHIMA,[†] MASAHIRO KONISHI[†,☆☆] and TAKASHI NAKADA[†]

This paper describes an improovement of our *worst case interruption delay* (WCID) analyzer, whose unique feature is *differential execution* of simulation threads correspoinding to all possible interruption points in a workload to be analyzed. The most important improvement given in this paper is the cycle count maintenace for *sleeping* threads whose execution is ommitted because it is *coherent* with other threads. That is, this maintenance took $O(N)$ time for N threads in our previous implemetation while the complexity is improved to $O(\log N)$ by our algorithm with binary tree. The previous implementation was fairly fast to complete an analysis in 9 hours while a straightforward method to execute all threads completely took 13 days. Our new imlementation is much faster than them because it finishes the same job in only 79 seconds. We also confirmed that workloads of SPEC CPU95 benchmark set are analyzed in a short time less than 30 minutes, while the straightforward method would take 200-300 days for the jobs.

1. はじめに

我々は，プログラムの最悪性能解析の一種である**最悪割込遅延**(Worst Case Interruption Delay: WCID) の解析を，cycle accurate simulator を用いて正確かつ高速に求める方法について研究している．図 1 に示すように 1 回の割込に関する WCID は，あるワークロードの実行過程のあらゆる割込可能な点で実行を中断し，キャッシュ，分岐予測器，パイプラインなどを割込後の実行にとって最悪となる状態にリセットした上で，残りの実行を継続して総サイクル数を得ることによって求めることができる．この「単純法」では，割込までの実行を命令レベルシミュレーション（図の細線）で行う工夫を施したとしても，cycle accurate な詳細シミュレーション（図の太線）には，ワークロードの総実行命令数 N に対して $O(N^2)$ の時間を要する．

そこで我々は，個々の割込に対応するシミュレーションスレッド（以下単に**スレッド**という）の実行過程でのマシン状態，すなわちキャッシュ，分岐予測器，パイプラインなどの状態が，他のスレッドと同じであるか，または違いが実行サイクル数に影響しない間は，スレッドのシミュレーションを省略する**差分実行**の方式を考案した[1]．またこの方式を実装して評価した結果，単純法では 15 日を要する解析を 9 時間で完了するという高い効果を得た．しかし，この約 43 倍の高速化は，詳細シミュレーションの対象命令数が差分実

† 豊橋技術科学大学
Toyohashi University of Technology
☆ 現在，京都大学
Presently with Kyoto University
☆☆ 現在，(株)PFU
Presently with PFU Limited

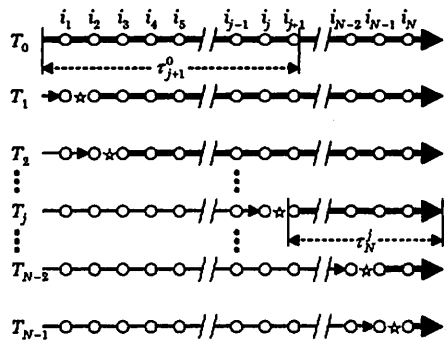


図 1 単純法

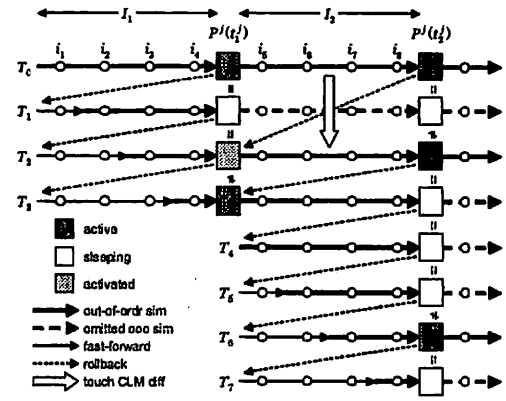


図 2 差分実行

行によって約 1/7000 になることを考えると，必ずしも十分な値であるとは言えない．

そこで本論文では，この差分実行に基づく WCID 解析器の実装を大幅に見直し，特にシミュレーションを省略する休眠状態のスレッドのサイクル数管理を改善することで，解析時間を大幅に削減する方法について述べる．

2. 設計と実装

2.1 差分実行

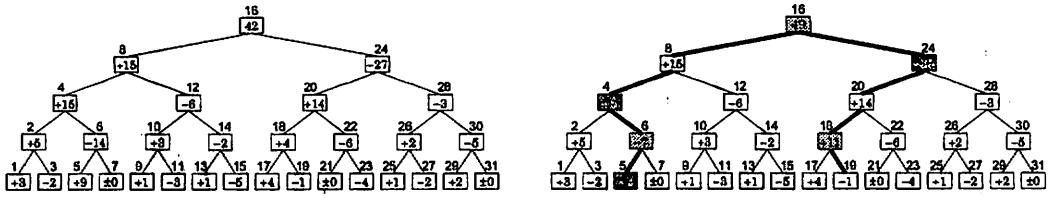
解析対象のワークロードを実行するプロセッサの状態は，実行した命令によってのみ定まるアーキテクチャ状態と，高速実行のための種々の機構が持つマイクロアーキテクチャ状態とに，二分して考えることができる．また後者は，命令の実行過程を表現するパイプライン状態と，キャッシュや分岐予測器などのキャッシュ様モジュール(Cache-Like Module: CLM) が保持する状態に分けることができる．

差分実行の基本的な考え方は，2 つのスレッドの実行過程でアーキテクチャ状態とマイクロアーキテクチャ状態が完全に一致すれば，それ以後は一方のシミュレーションを省略しても構わないという事実に基づく．また CLM の状態は，多数の部分状態（たとえばキャッシュのセットの状態）から構成されるが，パイプライン状態が一致する 2 つのスレッドについて，両者の間で異なる部分状態が参照されない間は，やはり一方のシミュレーションを省略することができる．

具体的な差分実行の手順を，図 2 に基づいて説明する．まず，実行中に割込が生じないスレッド T_0 を，一定の命令数からなる区間だけシミュレートする（図の太線）．すなわち図では区間長を 4 としているので（実際の実装では 8），4 命令の実行によるアーキテクチャ状態の更新が完了した時点で T_0 は中断する．続いて T_0 の実行によるアーキテクチャ状態の更新を全て元に戻す巻き戻しを行った後，最初の命令 i_1 の実行直後に割込が生じるスレッド T_1 のシミュレーションを行う．すなわち i_1 の命令レベルシミュレーション（図の細線）を行って割込点でのアーキテクチャ状態を求めた後，$i_2 \sim i_4$ の詳細シミュレーションを行って再び T_1 を中断する．

この時点で，あらかじめ保存しておいた T_0 のパイプライン状態と T_1 のパイプライン状態を比較し，それが意味的に一致すれば T_1 のシミュレーションは当面省略できるので，これを休眠状態にする．続いて T_3 と T_4 について同様のシミュレーションとパイプライン状態比較を行って，最初の区間 I_1 を終了する．図では T_3 も休眠状態となり，T_4 は T_0 とは異なるパイプライン状態を持つため活性状態のままであるとしている．

続いて次の区間 I_2 に移行し，インデックスが最も小さい最上位スレッド T_0 のシミュレーションを行う．この過程では，T_0 に支配されているスレッド，すなわち T_0 とパイプライン状態が一致したため休眠しているスレッドが存在するため，T_0 の CLM 状態と T_1，T_2 の CLM 状態の差異が後の実行過程に影響するか否かをチェックしなければならない．そこで，各々の CLM の個々の部分状態について，あるスレッドに固有の値，すなわち上位のスレッドと異なる値をスレッドの順に結合したリストを用い，差異の影響の有無を調べる．すなわち T_0 がある CLM のある部分状態を参照するたびに，その支配下スレッドの部分状態も参照し，その結果（キャッシュのヒット／ミスや分岐予測の結果）が T_0 のものと一致するか否かをチェックする．これが一致しなければ，I_2 のいずれかの時点でパイプライン状態が T_0 のものと一致しなくなると考



(a) before addition　　(b) after addition

図 3　二分木によるサイクル管理

えられるので，そのようなスレッドは**活性化**される．

図の例では，T_1 に固有の CLM 部分状態への参照は全て T_0 のものと同じ結果をもたらしたため，T_1 は I_2 の終了時点でも引き続き休眠状態となっている．このようなスレッドの CLM 部分状態は，区間内での更新により終了時点で上位の状態と一致する可能性もあり，もし一致すればリストから削除される．またこの結果，固有の部分状態を全く持たなくなった休眠スレッドは，全ての状態が上位スレッドと一致するため削除される．またこの部分状態の比較と一致時の削除は，活性スレッドに固有の部分状態についても行われる．

一方 T_2 については，参照された固有の CLM 部分状態の中に，T_0 の部分状態とは異なる結果をもたらすものがあったため活性化され，T_0 の中断後に I_2 の先頭からのシミュレーションが再開されている．この再開のために，T_0 の I_2 開始時点でのパイプライン状態が保存されており，これが T_2 の状態として利用される．続いて T_2 をシミュレートし，さらに I_1 終了時点で活性状態であった T_3 をシミュレートした後，新たなスレッド $T_4 \sim T_7$ の生成とシミュレーションを行って I_2 を終了する．

ここであるスレッド対 T_j と T_{j+1} について，両者の実行開始時のパイプライン状態と CLM 状態の差異は微小であり，前者についてはある定数サイクル以下の実行により，また後者についてはある定数回の参照により，それぞれ一致することが強く期待できる．したがって差分実行により，T_{j+1} の詳細シミュレーションが行われるサイクル数や命令数はある定数で抑えられることになる．この結果，サイクル数や命令数に関する限り，WCID 解析全体の手間は $O(N)$ となることが強く期待される．また巻き戻し，パイプライン状態の比較，支配下スレッドの CLM 部分状態の参照や一致比較など，差分実行に必要な処理のスレッドの区間シミュレーションあたりの実行コストも，やはりある定数で抑えられるものと期待される．

2.2　休眠スレッドのサイクル管理

WCID を求めるためには，各スレッドの実行サイクル数を求める必要があるが，休眠スレッドのサイクル数を管理するのは必ずしも容易ではない．すなわち，ある区間で活性状態であるスレッドがその区間に要したサイクル数を，その全ての支配下スレッドの総サイクル数に加算することは容易であるが，この処理は休眠スレッド数に比例する手間を要する．したがってこの方法では，サイクル数管理の手間が休眠スレッド数と区間数の積となるが，区間数はもちろん，休眠スレッド数も N に比例する可能性が高く，解析全体の手間が $O(N^2)$ となる恐れがある．

そこで休眠スレッドに対するサイクル数加算を，二分木を用いて $O\log N$ に比例する時間で実現するアルゴリズムを考案した．図 3(a) に示すように，T_0 を除くスレッド T_j の生成からのサイクル数は，二分木のノード n_j に親ノードとの差分の形で保持される．すなわち T_j のサイクル数を t_j，根ではないノード n_j の親ノードのインデックスを $p(j)$ とすると，n_j には $\delta_j = t_j - t_{p(j)}$ が保持される．また図では n_{16} である根ノードには，サイクル数の絶対値 t_{16} が保持される．したがって，ある t_j，たとえば t_6 の値を知る必要があれば，n_j から根に至る**祖先パス**上のノード値を全て加算すればよく，t_6 については $\delta_6+\delta_4+\delta_8+\delta_{16} = -14+15+15+42 = 48$ となる．

次に T_6 が $T_7 \sim T_{19}$ を支配している状況で，ある区間の実行を 7 サイクル要して完了したとする．したがって $\Delta = 7$ を $t_6, t_7, \ldots, t_{19}$ に加える必要があるが，この操作は図 3(b) の太線のパス，すなわち n_6 の左側の子 n_5 と，葉ノードである n_{19} から開始し，両者の共通祖先である n_{16} に至るパスに対して行われる．具体的には以下の手順によるが，ここで加算対象スレッドに対応するのノードの列を $S = \langle n_6, n_7, \ldots, n_{19} \rangle$ とし，また根にも仮想的な親（例では n_{32}）が存在してそのサイクル数は 0 であるとする．

(1)　$n_j \in S$ かつ $n_{p(j)} \notin S$ であれば，$\delta_j \leftarrow \delta_j + \Delta = (t_j + \Delta) - t_{p(j)}$ とする．例では n_6, n_{16}, n_{18} がこの条件を満たし，それぞれの値に $\Delta = 7$ が加えられている．

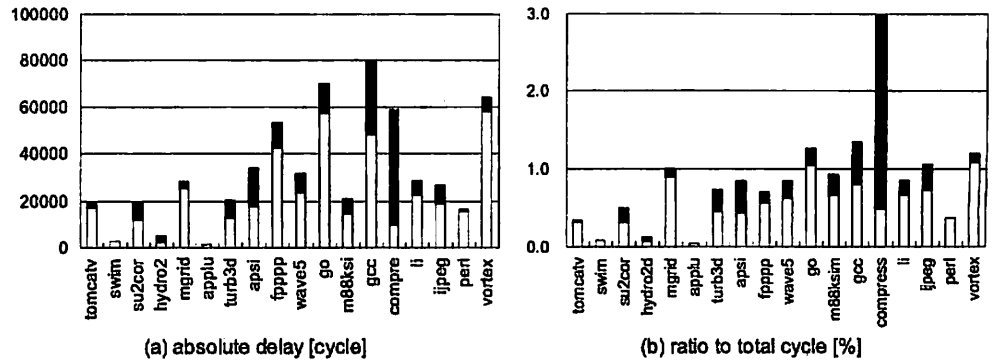


図 4 平均および最悪の割込遅延

(2) $n_j \notin S$ かつ $n_{p(j)} \in S$ であれば，$\delta_j \leftarrow \delta_j - \Delta = t_j - (t_{p(j)} + \Delta)$ とする．例では n_4, n_5, n_{24} がこの条件を満たし，それぞれの値から $\Delta = 7$ が差し引かれている．

(3) 上記のいずれでもない場合，すなわち $n_j \in S \leftrightarrow n_{p(j)} \in S$ であれば，t_j と $t_{p(j)}$ はともに加算対象であるか，あるいはともに対象外であるので，両者の差は変化しない．したがって δ_j は更新せず，図では n_8 ($n_8, n_{16} \in S$), n_{19} ($n_{19}, n_{18} \in S$), n_{20} ($n_{20}, n_{24} \notin S$) がこの条件を満たす．

このアルゴリズムの正当性は，共通祖先へのパス上にないノード n_k については，$n_k \in S \leftrightarrow n_{p(k)}$ が成り立つことに基づいている（証明は[2] 参照）．またこのアルゴリズムによる加算が $O(\log N)$ の手間で行われることは明らかであり，区間あたりの活性スレッド数が定数で抑えられるとすると，総管理コストは $O(N \log N)$ となる．またスレッドが活性化されるたびに $O(\log N)$ の手間でサイクル数の絶対値を知る必要があるが，同じ仮定に基づけばこの手間の総計も $O(N \log N)$ となる．

3. 評　価

3.1 評価環境

前節までに述べた新たな WCID 解析器を，SimpleScalar-3.0 をベースとして実装した．プログラムは C で記述し gcc 2.95.3 (-O2) でコンパイルし，3 GHz の Pentium-4 と 1 GB メモリからなる Linux PC で実行した．対象マシンの命令セットは PISA，またマイクロアーキテクチャの構成は SimpleScalar のデフォルトとした．ワークロードは SPEC CPU95 ベンチマークであり，m88ksim と compress 以外は，総実行区間

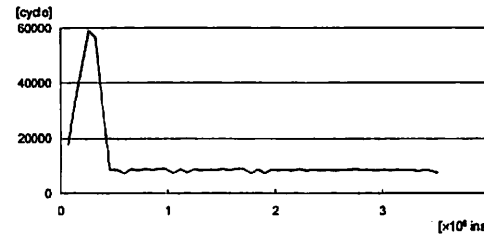


図 5 comprcss の 64K 命令区間ごとの最悪の割込遅延

中の中央の 500 万命令を解析対象とした．また WCID 解析器のスレッド実行制御の区間長は 8 命令とした．

3.2 WCID 解析結果

図 4 に各ワークロードの WCID を，(a) サイクル数の絶対値と，(b) 総サイクル数との比の，二つの指標で示す．また図の棒グラフの白い部分は，全ての割込点に関する割込遅延の平均値である．容易に予想できるように WCID はワークロードによって大きく異なり，サイクル数では 1,411 (applu) から 80,292 (gcc)，また比率では 0.03 % (applu) から 2.98 %の範囲に分布している．

これらの結果の中で特に注目すべきは compress の値であり，遅延の平均値は他のワークロードと同等であるが，最悪遅延の比率は突出して大きな値となっている．したがって，たとえば少数の候補点のみを対象として解析を行ったとすると，最悪値を著しく過小評価する恐れがあり，全ての候補点を対象とした解析の重要性を示唆している．実際，65,536 命令ごとにその中の割込がもたらした遅延の最大値を調べた結果，図 5 に示すように割込遅延の推移は非常に鋭いピークを持っており，大雑把な解析では最悪値を求めるのが困難であると予想される．

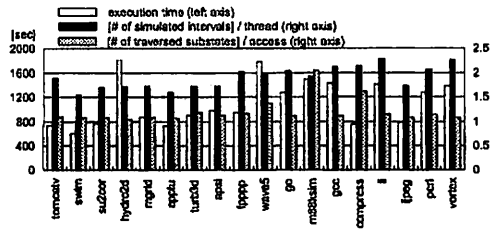


図 6 実行時間，スレッドあたりの実行区間数，および CLM 部分状態参照数

3.3 実行時間と差分実行の効果

本論文の重要なテーマである実行時間の削減については，まず文献 1) で用いた 9-queen プログラムを対象として評価した．このプログラムの総サイクル数は約 650 万であり，その中の 100 万〜110 万サイクルに割込を挿入して解析した．これを単純法で行うと約 15 日間を要し，また従来の方法では約 9 時間を要したが，今回の実装では僅か 79 秒にまで短縮された．これは従来は単純に行っていた休眠スレッドのサイクル管理を改善した効果のほか，区間の設定，パイプライン状態の比較法，CLM 部分状態の管理法など，様々な処理を改善した効果によるものである．

次に，SPEC CPU95 ベンチマークの実行時間を図 6 の白い棒グラフ（左軸）に示す．またこの図には実行時間を左右する重要な 2 つの指標として，スレッドあたりの実行区間数（黒）と，CLM へのアクセスごとに参照された部分状態数（灰色）も示している（右軸）．これらの値はいずれも極めて良好であり，まず実行時間は 608 秒 (swim) から 1,814 秒 (hydro2d) という非常に小さな値となっている．すなわち使用した評価環境での SimpleScalar の sim-outorder は 0.5 〜 0.7 MIPS の性能であり，単純法によって 12.5×10^{12} 命令を実行すると約 300 日を要するので，これを基準とすると約 15,000 倍の性能向上が達成されたことになる．

またスレッドあたりの実行区間数は 1.55 (swim) から 2.28 (li) の範囲であり，スレッド生成時に必ず 1 区間は実行されることを考えると，やはり極めて良好な値である．また CLM 部分状態参照数も 1.05 (hydro2d) から 2.14 (compress) の範囲であり，休眠スレッドの活性化判定のため余分に参照しなければならない部分状態は，最大でも 1 個程度であることがわかる．

さて，これらの性能指標はいずれも良好であるが，このことから解析の手間が $O(N \log N)$ であると結論することはできない．そこでこれらの区間ごとの値を計測して求めた一種の「微分値」を，図 7 と図 8 に示す．区間あたりの実行時間（図 7(a)）は概ね 1 〜 3 ms の範囲の定常値を示すが，m88ksim, li および wave5 では間歇的にピークが生じ，また hydro2d では高原状の部分が見られる．しかしこれらのピークや高原は 100 万命令程度の幅しかなく，実行した命令数に依存するというよりもプログラム中の実行フェーズの変化を反映しているものと考えられる．

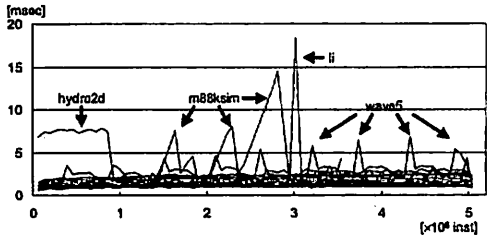
このフェーズ変化という仮説は，図 7(b)〜(f) に示す活性スレッド比，すなわち区間内で活性状態にあるスレッド数をその区間内で生成されたスレッド数（すなわち 8）で除したものと，図 8 に示す区間内での CLM 部分状態参照数の平均値によって裏付けられる．まず実行時間の推移にピークや高原がないワークロードでは，図 7(b) に示すように活性スレッド比はおよそ 1.5〜2.75 の範囲でほぼ定常している．一方 hydro2d の値 (c) には 2 つの高原があり，最初のものが (a) の高原に対応している．しかし (a) でピークが見られる 3 つのワークロードのグラフ (d)〜(f) には，周期的な実行フェーズ変化を反映していると思われる挙動が見られるものの，(a) のピークに対応はしていない．それに対し図 8 に示す CLM 参照数のグラフ中のピークは，図 7(a) のピークに非常によく対応しており，スレッド固有値の種類が多い CLM 部分状態が間歇的に参照されて一時的な実行時間増を招いていることがわかる．

最後に，休眠スレッド数の区間ごとの値を図 9 に示す．このグラフから少なくとも li, perl, tomcatv について，休眠スレッド数が実行命令数に比例して増加していることがわかる．したがって，仮に CLM も含めた全ての状態が一致することをスレッドのシミュレーション省略条件としたり，あるいは休眠スレッドのサイクル数管理をその数に比例する手間で行うと，全体的な実行時間が $O(N^2)$ となってしまうことが明らかになった．

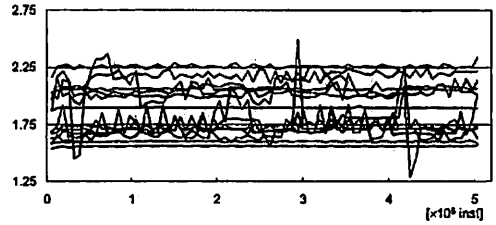
これらの結果から，WCID 解析器の実質的な時間計算量は $O(N(K + \log N))$ であり，かつ大きな定数 K が $\log N$ の効果をほとんど覆い隠していることがわかり，我々のアルゴリズムの高い効率性が実証された．

4. ま と め

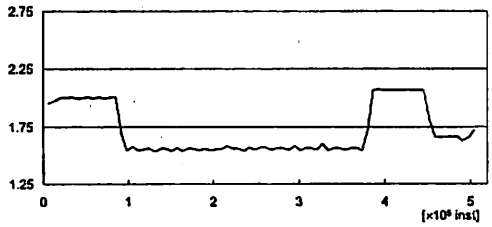
本論文では，差分実行に基づく WCID 解析器の改良とその性能評価について述べた．実装方式の改善は極めて効果的であり，従来の実装では 9 時間を要していた解析を僅か 79 秒で実施することができた．また SPEC CPU95 ベンチマークを用いた評価により，単純な $O(N^2)$ の方法では 1 年近く要するような解析が約 30



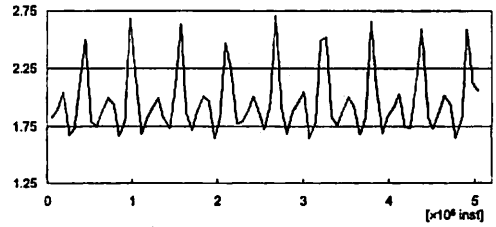
(a) 実行時間



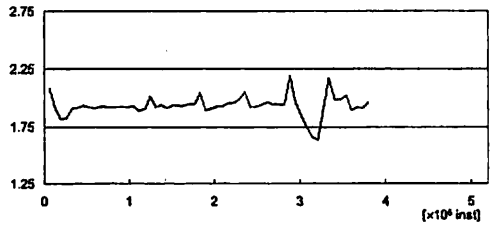
(b) 活性スレッド比（4 ワークロード以外）



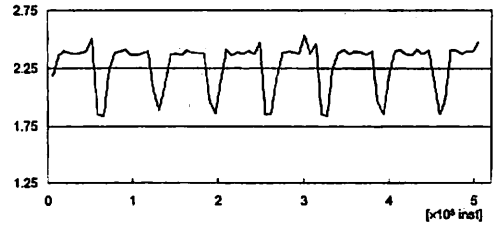
(c) hydro2d の活性スレッド比



(d) wave5 の活性スレッド比



(e) m88ksim の活性スレッド比



(f) li の活性スレッド比

図 7　区間ごとの実行時間と活性スレッド比

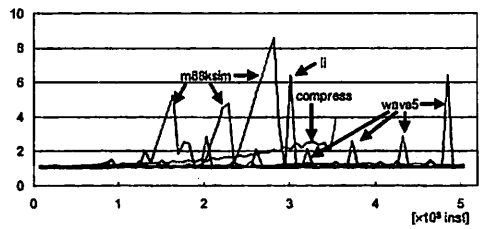


図 8　区間ごとの CLM 部分状態参照数

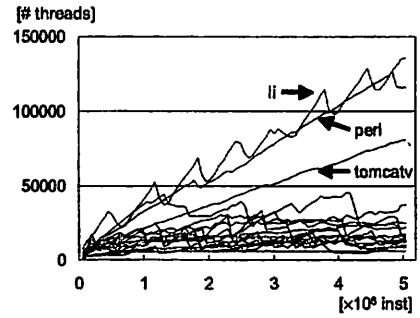


図 9　区間ごとの休眠スレッド数

分以内に完了し，かつ解析の手間が $O(N(K+\log N))$ であることも明らかになった．

今後の課題としては、現在 1 回に限定している割込を、複数回にするための効率的アルゴリズムの探求が挙げられる。

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) 小西昌裕，中田　尚，津邑公暁，中島　浩：重複実行省略を用いた割り込みによるマイクロプロセッサの最悪性能予測，*SACSIS 2006* (2006).

2) Nakashima, H.: An $O(\log N)$ Algorithm to Increment Subarray Members of an Array of N Elements, Technical Report http://www.para.tutics.tut.ac.jp/TR/tree-add.pdf, Toyohashi U. Tech. (2006).