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

2004－ARC－159 (18)
2004／7／31

共有メモリ型並列計算機の分散シミュレータの設計

久野和樹† 中田尚† 中島浩†

本論文では，我々が提案している共有メモリ型並列計算機の分散シミュレータ Shaman を out-of-order 実行可能なプロセッサで構成された並列マシンの分散シミュレーションを行うことができるよう拡張する方法を検討した．Shaman は in-order 実行する要素プロセッサを想定しており，プログラムの論理挙動をシミュレートするフロントエンドが生成した参照履歴をフィルタリングし，フィルタを通過した参照履歴のみを対象システムの物理的挙動をシミュレートするバックエンドに渡すことによって，効率的な並列処理を実現している．一方，out-of-order 実行可能なプロセッサは，メモリ参照遅延によってそれ以降のプロセッサの挙動が変化してしまう．このため，参照履歴を記録しながらシミュレーションを続けるのではなく，参照フィルタをとおりぬけたアクセスについては逐一バックエンドに対して通信して競合を解決することで対応させる．簡単な実験によってその性能を予測した結果，フロントエンドとバックエンド間の通信オーバヘッドは高々20%程度と見積もられ，本方式によって高い台数効果が得られる見通しが得られた．

Design of Distributed Simulator for Shared Memory Multiprocessors

KAZUKI KUNO,† TAKASHI NAKADA† and HIROSHI NAKASHIMA†

To simulate shared memory multiprocessors that consist of out-of-order microprocessors, this paper discusses an extension of our Shaman distributed simulator. Shaman assumes its target system has in-order microprocessors. This assumption made it possible that its front-end simulates logical behavior of a workload generating a memory access trace filtered by a cache and sends the trace to its back-end that simulates physical behavior. On the other hand, the instruction scheduling of an out-of-order microprocessor depends on the latency of memory accesses. Thus in our new simulator, a front-end communicates with the back-end each time it finds a memory access passing through the filtering cache to determine the access latency. We estimated the performance of the new simulator by a simple test-bed. As a result, we found the communication overhead will be less than 20% and thus sufficiently high parallel speed-up will be obtained.

1. はじめに

アーキテクチャの研究において，新しいアイディアの検証にシミュレータは重要なツールである．しかし，並列計算機のような大規模システムでは，そのシミュレーション時間が問題であり，高速化のための研究が数多くなされている．

並列計算機のシミュレーションでは，対象マシンの並列性を利用したシミュレーションの並列化が有効である．しかし，各プロセッサ間には相互作用があるため，対象システムを単純に分散シミュレータの１つのノードにマップして独立にシミュレーションすることはできない．相互作用を満たすために，たとえばクロック毎に全ノードが同期するといった方法が必要であるが，これを分散環境で実現すると，禁則的なオーバヘッドが生じることが予想される．

一般的な共有メモリ環境では，全ての load, store におけるメモリ参照が潜在的なプロセッサ間通信であり，これら全ての参照の履歴を記録しなければならないため，履歴の大きさが問題となる．そこで我々は共有メモリ型並列計算機 Shaman を開発した[1]．Shaman では，以下の３点に着目してこの問題を解決している．

(1) 正しく同期が行われる決定的なワークロード・プログラムに対しては，ソフトウェア分散共有メモリの技法を用いたシミュレーションにより，個々のプロセッサに関する限り正しく順序づけられたメモリ参照履歴を生成することができる．

(2) 対象システムのコヒーレント・キャッシュを部分的にシミュレートすることにより，対象キャッシュで必ずヒットするようなメモリ参照を履歴から除去するフィルタ操作を行う．

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

(3) フィルタを通り抜けた履歴には, 対象システムにおける全てのキャッシュミスが含まれており, これらによってメモリ・システムの動作タイミングの正確な, あるいは十分な精度を保った, 再現が可能である.

近年, プロセッサは高速化のために様々な方式が考案され, 採り入れられている. 代表的な方式として out-of-order 実行がある. この方式は, メモリ参照遅延によって, 後続命令の実行順が変化するという特徴を持つ. これに対し, Shaman が対象にしているのは in-order 実行のプロセッサである. in-order 実行のプロセッサでは, 後続の命令の実行順がメモリの参照遅延によらず独立である. このため正確な遅延時間は, 後に履歴を元に検証を行えばよいので, メモリ参照が発生した際, 正確な遅延時間を決定することなくシミュレーションを継続させることができる. Shaman はこの特性を用いて, メモリの物理的動作とプロセッサの論理的動作の依存性を解消し, ほぼ独立してシミュレーションを進めることができる. 一方, 命令の実行順が変化する out-of-order 実行型のプロセッサでは, 遅延を決定しなければ以後の動作が決定しないため, シミュレーションを継続できない.

本稿では, Shaman で用いている Software-DSM や, メモリ参照履歴のフィルタリング手法を用いて, out-of-order 実行可能なプロセッサで構成された並列計算機の分散シミュレータを設計し, シミュレータ完成時の性能を予測する. Shaman の概要について 2 章で述べ, 3 章で, out-of-order 実行可能なプロセッサにより構成された分散シミュレータの構成手法を述べる. 4 章では, 簡単なシミュレータの実装を行い, 5 章で分散シミュレータ完成時の性能予測を行う. 6 章でまとめを述べる.

2. 分散シミュレータ Shamen

2.1 概　　要

Shaman では, 論理的な動作と物理的な動作にシミュレータを分離することでシミュレーションを改善する方法を提案した. フロントエンドがプロセッサが実行する命令の論理的な動作をシミュレートする. バックエンドがキャッシュミスとコヒーレンス維持操作に伴うプロセッサ間の物理的な通信をシミュレートする.

フロントエンドにおけるシミュレーションではプログラムの意味を保てばよい. Lazy Release Consistency に基づく Virtual Shared Memory (LRC-VSM)[2] の技法を用いた Software DSM によってページ単位のコヒーレンス制御を仮想的に行うことで, 分散環境でも効率よくシミュレーションできる. このとき正確な性能を検証するための履歴を記録する. 生成する参照履歴を削減するために, 我々は参照フィルタを提案した. 対象システムのキャッシュを部分的にシミュレートすることで実行タイミングによらず確実にヒットするような参照履歴を削減する. 一方, バックエンドではプロセッサ間ネットワークにおける競合などをシミュレーションするために緻密な時刻管理が必要であるが, フロントエンドで生成した履歴のみを対象とすればよく, フロントエンドに比べて論理構造が単純であるため, 逐次シミュレーションでも十分な性能が期待できる.

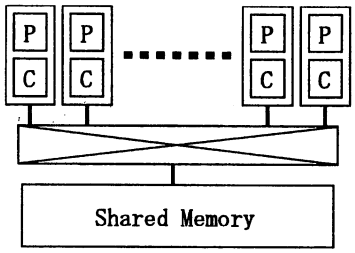

図 1　対象システム

2.2 対象システム

Shaman が対象とするシステムは, 一般に図 1 に示すような集中型あるいは分散型の共有メモリ型の並列計算機である.

ワークロード・プログラムは, Solaris または POSIX のスレッド・ライブラリによって書かれたマルチスレッドプログラムである. また, バリア操作など, 独自の同期プリミティブも用意されている.

実行方式は, Keleher らによる lazy release consistency (LRC) の機構に基づいており, ワークロード・プログラムは data-race-free (DRF)[3][4] の性質を満たしていなければならないが, 有用な並列プログラムの大部分は DRF であることが知られている.

2.3 構　　成

対象システムの Shaman へのマッピングは図 2 のように行われる. なお, P はプロセッサ, C はキャッシュを示す. Shaman は複数のフロントエンド・ノードと単一のバックエンド・ノードによって構成される.

フロントエンドには対象システムの各プロセッサがマッピングされ, 論理的な動作のシミュレーションを行う.

バックエンドにはキャッシュ, ネットワーク, メモリがマッピングされる. ただし, バックエンドでは, メモリシステムの動作タイミングなどの物理挙動のみをシミュレートし, 共有メモリを介したデータ送受信は Software-DSM によってフロントエンドがシミュレーションする.

2.4 参照フィルタ

フロントエンドが生成した参照履歴はバックエンドに送られるが, 全ての参照履歴を送信すると極めて大きな通信オーバーヘッドが生じる. そこで, 個々のフロントエンド・ノードはプロセッサ毎にフィルタ・キャッシュ(C_f) を持ち, このフィルタをミスした参照のみ履

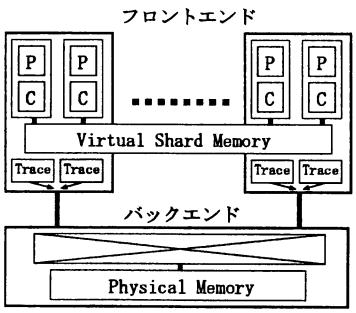

図 2 Shaman の構成

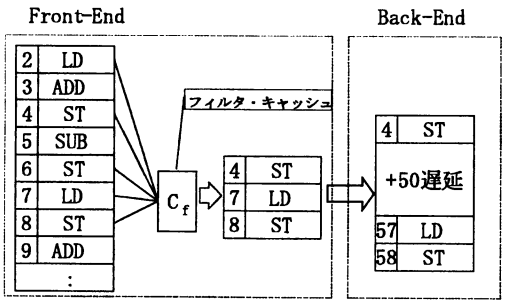

図 3 動作タイミングの再現例

歴に残し, バックエンドに対して送信する. スピンロックのような非同期性をもつ同期プリミティブは, スピン回数を再現するためにプリミティブ全体がバックエンドに送られる. 送信される参照やプリミティブには, 個々のフロントエンドに局所的なタイムスタンプが付加される.

バックエンドでは, 対象システムのメモリやネットワークだけでなく, 対象システムのキャッシュと同じ構成の対象キャッシュ(C_t) のシミュレーションも行う. フロントエンドから送られた参照に対し, 対象キャッシュにより再度ヒット/ミスの判定を行い, 真のミスだけが抽出されてメモリやネットワークシミュレーションに用いる. 動作タイミングの再現例を**図 3** に示す. 以下, このフィルタ・キャッシュを用いて行われる参照フィルタ操作の具体的な方式について説明する.

2.4.1 フィルタ・キャッシュ

フィルタ・キャッシュは, 以下のようにコヒーレント・キャッシュを部分的にシミュレートする. なお, 説明を簡単にするために, write-invalidate プロトコルと {M,S,I} の 3 状態キャッシュを仮定する.

(1) 対象システムのキャッシュ(C_t) が容量 $\gamma_t = \gamma \cdot w$ の w-way set associative であるとき, フロントエンドでシミュレートするキャッシュ C_fをブロックサイズが同一で容量 γ の direct map とする. この結果 C_tで容量性あるいは競合性のミスが生じるならば, 必ず C_fでもミスが生じる. なおブロックサイズは LRC-VSM のページサイズよりも小さく, 両者の境界は整合しているものとする.

(2) プロセッサのアクセスによるブロックの状態遷移は C_t, C_fで同一とする.

(3) 他のプロセッサのアクセスによるコヒーレンス維持操作では, C_tはページの更新情報である diff が送信された場合, diff に含まれる全てのラインについて以下のような状態遷移をする.
- 送信元: $\mathbf{M} \Rightarrow \mathbf{S}$
- 送信先: $\{\mathbf{M}, \mathbf{S}\} \Rightarrow \mathbf{I}$

なお, 他のコヒーレンスプロトコルや状態追加については, 状態や状態遷移を MSI プロトコルに写像することで対応することができる. 例えば, MESI プロトコルは, exclusive-clean な状態 (E) を shared-clean な状態 (S) に写像することによって, 正しいフィルタ操作を実現することができる.

2.4.2 DRF ブロックのアクセス履歴

あるメモリブロックに属する任意の要素に対するアクセスが DRF であるとき, そのブロックを DRF ブロックと言う.

フィルタ・キャッシュを用いて行われる参照フィルタ操作は DRF ブロックについて, C_fにおける広義のミスのみを参照履歴として残せばよい. すなわち, DRF である操作に関しては, C_tでコヒーレンス・ミス (S のブロックへの書き込みを含む) が生じるならば, 必ず C_fでもミスが生じる[5].

2.4.3 非 DRF ブロックの検出

DRF でないブロックを無視してしまうと, 広義のミスであるにもかかわらず, 履歴から除去されてしまう. そこで我々が既に提案している非 DRF ブロックの検出アルゴリズム[5] を利用して, プログラムを一度実行すれば非 DRF なブロックを検出することができる. シミュレーションは次の 2 フェーズにわけて, 必要なアクセス履歴を生成する.

フェーズ 1 全てのブロックの操作は DRF であるとの仮定で, 参照履歴を生成しながら非 DRF なブロック操作のチェックとマーキングを行う. 非 DRF なブロックが存在しなければ, ここでシミュレーションが終了する. 非 DRF ブロックは, ブロックに対する局所的な参照時刻と, ブロックを更新する diff の生成時刻を比較することにより発見することができる. そこで, 非 DRF ブロックを発見したら, そのノードは各ノードにその旨を伝える. 以降, 全てのノードは参照履歴の生成を中止して, 非 DRF ブロックの検出のみを行うようにする.

フェーズ 2 フェーズ 1 が完了して全ての非 DRF ブロックが発見されると, プログラムを再実行する. フェーズ 2 では, 非 DRF ブロックについては全てのアクセスに付いて履歴を生成する. バック

エンドはこの履歴に基づいてシミュレーションを行う.

このときフェーズ1で起こった通信を記憶しておくことで, フェーズ2では, 各ノードが独立にシミュレーション可能である.

3. 対象マシンのOut-of-order実行サポート

3.1 メモリ参照遅延の逆依存性の解決

図3で示したように, in-order実行型のプロセッサにより構成された並列計算機を対象としているShamanのフロントエンドは, フロントエンドで計算された局所時刻でシミュレーションを行っている. フロントエンドでのシミュレーションが終了した後, 参照履歴をバックエンドに送信してシミュレーションしたアクセス競合を局所時刻に加算することで, 局所時刻を対象システム全体における大域時刻に変換可能である.

一方, 図4に示すように, out-of-order実行型のプロセッサでは, キャッシュミスによる遅延が生じると, 後続命令が先行命令を追い越してスケジューリングされるため, 遅延が分からなければ局所時刻の算出は元より, 命令の実行順すら決定することができない. このため, アクセス履歴を利用する設計では, C_tでミスが発生してメモリ参照が発生する度にシミュレーションをやり直さなければならず, 現実的ではない. C_fでミスが発生した場合, あるいは非DRFブロックにアクセスする場合に, バックエンドに対して逐一, 参照遅延を問い合わせる必要がある. このとき, バックエンドは次のような動作をする.

(1) C_tでヒットした場合, バックエンドはスケジューラに対して, その時のシミュレーション時刻を通知し, 直ちにフロントエンドにおけるシミュレーションを再開させる.

(2) C_tでミスした場合, メモリへの参照競合が発生し, 解決しなければ, シミュレーションを再開できない. ミスであることが分かると, シミュレータはスケジューラに対して値が取れる時刻の問い合わせを行う. この問い合わせと同時に, シミュレーション時刻もスケジューラに対して通知する. スケジューラは, ミスが発生した時刻と, 他のプロセッサのシミュレーション時刻が等しくなるか追い越すまで競合状態が決定しないため待機する. その後, アクセス競合をスケジューリングして参照遅延を報告する.

上記の方法では, C_fをミスするたびにバックエンドとの通信が発生するため, 参照履歴を用いる方式に比べてオーバヘッドが大きくなるが, out-of-order実行をサポートするシミュレータは一般に低速であるため, 通信によるオーバーヘッドの占める割合は少なくなるものと考えられる.

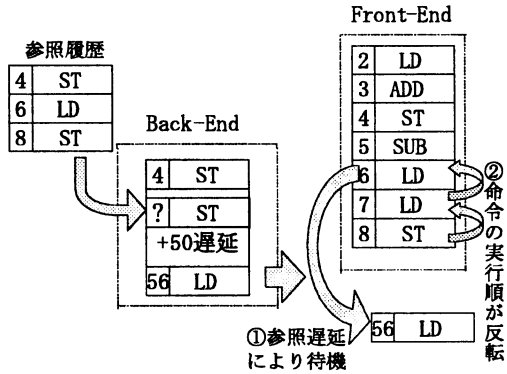

図4 out-of-order実行型プロセッサにおける動作タイミング

3.2 同期命令のクロックレベルシミュレーション

スピンロックやバリア同期などの非同期性を持つ同期プリミティブのシミュレーションは, クロック毎の同期が必要である. 先に述べたように, 分散環境でクロック毎の同期を行えば, 禁則的なオーバーヘッドを生じることが容易に予測されるため, バックエンドでまとめてシミュレーションを行う. out-of-order実行プロセッサの同期プリミティブの動作をクロックレベルで正確にシミュレーションするためには, 正確なマシン状態を再現しなければならない. バックエンドが保持しているマシン状態は, メモリとキャッシュである. そのため, 状態を保持しているフロントエンドと通信を行い, シミュレーションを行うためのマシン状態を構築する必要がある. バックエンドが命令をシミュレーションするために必要なデータは, 分岐予測器, パイプライン状態, レジスタ, レジスタの依存表等である. パイプライン状態などは状態数が少ないが, 分岐予測器は大きな状態数を持ち, 分岐予測器をバックエンドで再現するための通信がボトルネックとなる可能性がある. この通信の削減は今後の課題である.

同期プリミティブを実現するために以下の命令を用意してスピンロックにより実装する.

- LL (Load Linked)
- SC (Store Conditional)
- SYNC

同期プリミティブのシミュレーションが終わった後, フロントエンドでシミュレーションを再開させるためには, フロントエンドのマシン状態を同期プリミティブが終了した状態まで進める必要がある. このとき同期プリミティブのシミュレーション開始時にバックエンドに対して送信したものと同じ情報をバックエンドから送信すれば良いが, 分岐予測ミス時の投機実行で触れたキャッシュを更新している可能性があるので, キャッシュの状態もバックエンドとフロントエンドで同期を取り, C_fを更新する必要がある. キャッシュは状態数が多く, 効率的に実装しなければならない. 代

替案として, スピンロックの回転数と参照遅延の履歴を送信して, フロントエンドで同期プリミティブの振る舞いを再現する方法が考えられる. この方法は, 同期プリミティブの競合が少なく, スピンロックの回転数が少ない場合には有効に働くが, 回転数が多くなると履歴が肥大化してしまうという欠点がある.

4. 性能予測用シミュレータの実装

本シミュレータは, メモリ参照が発生した際, C_fにヒットしなければ, バックエンドに対して逐一通信を行わなければならない.

out-of-order 実行をサポートするプロセッサのシミュレータは一般に低速である. 実際の対象マシンと比較した場合のスローダウンが 1,000〜10,000 と言われており, これら通信によるオーバヘッドが問題にならない可能性がある. また, 逆に通信によるオーバーヘッドが非常に大きく, シミュレータの分散化による高速性が得られない可能性も考えられる. そこで, 簡単なシミュレータを実装して実験を行い, シミュレータ完成時の性能を予測した.

4.1 実装したシミュレータの概要

参照フィルタの履歴削減効率については, 先の研究で結果が出ている.

そこで, C_fにミスしたメモリ参照が逐一バックエンドに対して行われたときのオーバーヘッドを予測するために, メモリのデータを 1 ノードに集めて全ての load, store 命令をバックエンドに対しておこなうようなシミュレータを実装する. **図 5** に, 実装したシミュレータの構成を示す. 分散環境で複数のフロントエンドと単一のバックエンドからなる. フロントエンドはプロセッサのシミュレーションを行う. SimpleScalar の sim-outorder をもとに実装を行った. バックエンドはメモリのデータについてシミュレーションを行う. いずれも通信量と通信回数の再現が目的であるため, 物理的な挙動については考慮していない.

これを用いて SimpleScalar の sim-outorder のメモリ参照がネットワーク越しに行われたときの通信によるオーバーヘッドを測定する.

4.2 同期プリミティブ

同期のメカニズムを SimpleScalar で実現するために, 以下の命令を追加した.

- LL (Load Linked)
- SC (Store Conditional)

また, これらの命令を用いて, POSIX threads に含まれる mutex_lock および mutex_unlock の同期プリミティブをライブラリ関数として実装した.

ロックの管理は 4Byte のメモリ領域を用いて行う. この領域が 0 のとき, ロックが開放されている状態である. この領域が 0 以外に設定されているとき, いずれかのスレッドがロックを所持していることを示している.

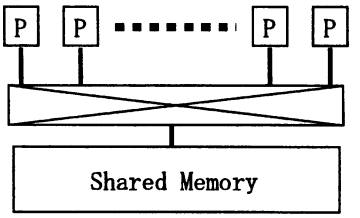

図 5 試験用シミュレータの構成

mutex_lock は, ロックの取得を行う関数である. ロックを管理する, メモリ領域のポインタを引数として 1 つ指定する. ロックが取得できないときは, 現在ロックを持っているスレッドがロックを開放するのを待ち, 再度ロックの取得を試みる動作をする. 以下のように実装をした.

```
$L7:
  lw $2,0($4)
  bne $2,$0,$L7       #ロックが開放されるまで待機
$L11:
  addu $2,$0,1
  ll $3,0($4)
  sc $2,0($4)
  beq $2,$0,$L11      #SC 命令が成功するまでループ
  bne $3,$0,$L7       #ロックが取得できるまでループ
```

mutex_unlock は, ロックの開放を行う. これは単にロックを管理する領域に対して 0 の代入を行う.

4.3 スレッドの生成と終了

スレッドの生成は, 生成関数の呼出をシミュレータが検知して行う. 検知は, 始めに対象プログラムのシンボル情報からスレッド生成関数のスタートアドレスを取得し, Jump and Link (JAL) 命令でジャンプ先のアドレス一致で行う.

スタックが空になったとき, あるいは exit システムコールによってスレッドを終了する.

5. 性能予測

5.1 ワークロード

ワークロードとして 100 行 100 列の正方行列の行列積の計算を, スレッド数を変えて (1, 2, 4, 8) 行うプログラムを動作させ, その計算時間を計測する. また, オリジナルの SimpleScalar で同様なワークロードを動作させて, その計算時間を計測する.

5.2 計測環境

実行時間計測には CPU が Mobile Intel Pentium III 866 MHz で, メモリを 512MB 搭載し, Redhat Linux 7.3 (kernel 2.4.7-10) をインストールした PC を GigabitEthernet で接続した PC クラスタを使用した.

5.3 計測結果

計測結果を**表 1** に示す. また, そのグラフを**図 6** に示す.

スレッド数 1 のときの計算時間と, オリジナルの

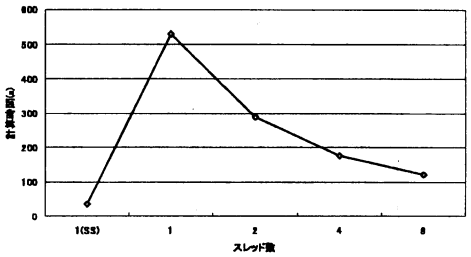

図 6 測定結果

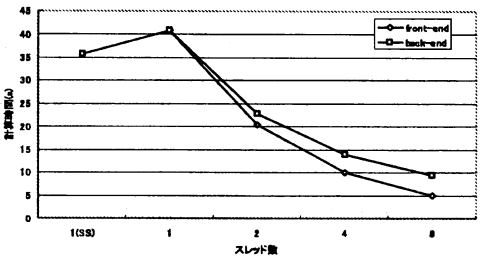

図 7 予測計算時間

SimpleScalar の計算時間との差は, 通信に要している時間と考えられる. 表 1 より, 通信に要した時間 t_c は下式のようになる. ただし SimpleScalar の計算時間を t_{SS}, n スレッドのときの計算時間を t_n とする.

$$t_c = t_1 - t_{SS} = 495.0[s] \quad (1)$$

オリジナルの SimpleScalar と比較する.

$$\frac{t_c}{t_{SS}} = 13.8 \quad (2)$$

これにより, オリジナルの SimpleScalar の計算時間と比較して, およそ 14 倍の時間を通信に使用していることが分かる.

5.4 フィルタ・キャッシュ使用時の予測計算時間

測定によって得られた値を用いて, Shaman の研究結果を元に, シミュレータ完成時の性能を予測する.

Shaman の研究では, 参照フィルタによって, 参照履歴がおよそ 1/100 に削減されるという結果を得ている. 本シミュレーターでは, バックエンドとの通信量が 1/100 になることに相当する. したがって, 上記の例では t_c/t_{SS} は 0.138 となり, 通信オーバーヘッドは 14% 程度になると見積もられる. なお, この評価では Software DSM による影響を考慮していないが, 対象のワークロードでは, 同期回数が 1 回で影響が小さいため, これを含めたオーバヘッドも高々 20% 程度になると予想される.

スレッド数を増やしたときの計算時間も同様に予測した. グラフを図 7 に示す. ボトルネックがバックエンドにある場合と, フロントエンドにある場合についてプロットした. 実際にはそれらの中間のいずれかの値を取るものと考えられる.

6. ま と め

本論文では我々が提案している共有メモリ型並列計算機の分散シミュレータ Shaman を out-of-order 化する方法を検討し, その性能を予測した. このシミュレータでは, 逐一メモリの参照遅延を決定しながらシミュレーションを行うことで, 命令の論理挙動に対する参照遅延の依存性を解決する.

簡単な実験によって, 性能を予測した結果, 僅かなオーバーヘッドで十分な台数効果が期待できる結果が得られた.

現在, 本論文の提案に基づく分散シミュレータの実装を行っており, 実装完了後に提案方式の有効性を評価して報告する予定である.

表 1 計測結果 (s)

スレッド数	1(SS)	1	2	4	8
計算時間	35.8	530.8	289.2	174.8	121.4

※ SS は オリジナルの SimpleScalar の計測値

謝辞 本研究の一部は (株) 半導体理工学研究センターとの共同研究「SpecC によるソフトウェア記述の性能検証システム」による.

参 考 文 献

1) 松尾治幸, 大野和彦, 中島浩: 共有メモリマルチプロセッサの分散シミュレータ Shaman の設計と実装, 情報処理学会論文誌ハイパフォーマンスコンピューティングシステム, Vol.44,No.SIG1(HPS6) pp.114-127 (2003).
2) Keleher, P.: *Lazy Release Consistency for Distributed Shared Memory*, PhD Thesis, Department of Computer Science, Rice University (1994).
3) Adve, S. V. and Hill, M. D.: Weak Ordering—A New Definition, *Proc. 17th Intl. Symp. Computer Architecture*, pp. 2–14 (1990).
4) Adve, S. V. and Hill, M. D.: A Unified Formalization of Four Shared-Memory Models, *IEEE Trans. Parallel and Distributed Systems*, Vol. 4, No. 6, pp. 613–624 (1993).
5) 今福茂, 大野和彦, 中島浩: 共有メモリ・マルチプロセッサの分散シミュレータのための参照フィルタ方式, 情報処理学会論文誌ハイパフォーマンスコンピューティングシステム, Vol.42,No.SIG9(HPS3),pp.93-105 (2001)
6) Austin, T., Larson, E. and Ernst, D.: SimpleScalar: An Infrastructure for Computer System Modeling, *Computer*, Vol. 35, No. 2, pp. 59–67 (2002).