平成 27 年度 修士論文

共有キャッシュメモリを有効活用するスレッドスケジューリング機構

電気通信大学 大学院情報システム学研究科
情報システム基盤学専攻
1153021　藤田 竜一

主任指導教員　多田 好克　教授
指導教員　小宮 常康　准教授
指導教員　新谷 隆彦　准教授

提出日　平成 28 年 2 月 10 日

目次

図目次

表目次

プログラムコード一覧

第 1 章

背景と目的

1.1 CPU の進歩による並列処理の重要性向上

現代の CPU は消費電力あたりの性能の問題からクロックの上昇による 1 コアあたりの性能向上が鈍化しており，CPU1 パッケージ内に搭載するコアの数を増やすマルチコア CPU によって性能の向上を続けている．このようなマルチコア CPU の持つ計算能力を活かすため，プログラムの並列処理の重要度が増大している．プログラムが並列処理を行う方法として，1 つのプロセスのメモリ空間を共有しながらも独立した実行の単位を持つ「スレッド」を複数作成するマルチスレッドプログラムが一般的に用いられている．

1.2 メモリアクセス速度問題によるキャッシュメモリの重要性の増大

CPU の性能向上に対してメインメモリの性能向上が追い付かず，メインメモリにアクセスしてデータを読み書きする時間が CPU にとって長大な待ち時間となり CPU の利用効率が悪化してしまう [1] [2] [12]．そこで近代の CPU では小容量ながら高速にアクセスできるキャッシュメモリを設置し，頻繁にアクセスする一部のデータについてはキャッシュメモリ上で高速に読み書きすることでメインメモリへのアクセス速度を隠ぺいしている．

キャッシュメモリはメインメモリに対して容量が非常に少ないことから，あるプログラムが利用するメインメモリ上の全てのデータを格納できることは稀である．そのためメインメモリ上の一部のデータのみをキャッシュメモリにコピーして使用するという状態が恒常的となる．キャッシュメモリも複数の階層を持ち，2016 年現在では CPU に近い順に L1 から L3 までの 3 階層のキャッシュメモリが設置されていることが多い．CPU に近い L1 は高速なアクセスが可能だが容量は少なく，CPU から 1 番離れたラストレベルキャッシュ (LastLevelCache) とも呼ばれる L3 はアクセスに時間がかかるが容量が大きくなっている．本論文では特別な説明なくキャッシュ・キャッシュメモリと書かれている場合は，ラストレベルキャッシュを指すこととする，

マルチコアプロセッサでは，コアごとに独立した L1,L2 キャッシュを持ち，L3 キャッシュを全コアで共有する設計となっている製品が多い．

1.3 マルチスレッドプログラムにおけるキャッシュミスによる悪影響

キャッシュメモリ上のデータは，メインメモリ上の一部のデータのみコピーして保持していることから，CPU がデータを使用する際にはキャッシュメモリ上にデータのコピーが見つからず低速なメインメモリへのアクセスを余儀なくされるデータも発生する．この現象をキャッシュミスと呼ぶ．逆に，CPU がキャッシュメモリ上のデータにアクセスできた場合をキャッシュヒットと呼ぶ．

キャッシュミスには以下のような原因に応じた分類が存在する．

- 初回参照ミス：CPU がデータに初めてアクセスする場合，この時点ではデータはキャッシュメモリにコピーされていないためにキャッシュミスとなりメインメモリにアクセスする必要がある．これを初回参照ミス (Cold Start Miss とも) と呼ぶ．2 回目以降のアクセス時にはキャッシュメモリにデータのコピーが存在する可能性があるためキャッシュヒットが期待できる．初回参照ミスを回避するため，将来使用すると予測されるデータをあらかじめメインメモリからキャッシュメモリにコピーしておくプリフェッチ技術が存在する．プログラム中で明示的に特定データのプリフェッチを指示するソフトウェアプリフェッチや，次に使用するであろうデータを空間局所性やメモリアクセスパターンの観点からプログラムの実行中に CPU が自動的に予測し投機的にプリフェッチを行うハードウェアプリフェッチが存在する．
- 容量性ミス：あるプログラムが繰り返しアクセスするデータのサイズがキャッシュメモリの容量よりも多い場合，一定のアルゴリズムに従ってキャッシュメモリ上のデータが削除され，別のデータに使用されることがある．この場合，過去にアクセスしたことがあるデータであってもキャッシュメモリ上にコピーがないためキャッシュミスとなる．これを容量性ミス (Capacity Miss) と呼ぶ．容量性ミスを回避する対策として，実行するプログラムを書き換えてあるデータがキャッシュメモリ上から追い出される前に次のアクセスができるように時間局所性を高める方法や，搭載されているキャッシュメモリのサイズがより大きい CPU を使用する方法がある．
- 競合性ミス：キャッシュメモリとメインメモリ上のデータのコピーはメインメモリの空間を固定長で区切ったキャッシュラインと呼ばれる単位で行われる．キャッ

シュメモリ上でのデータの配置は，そのデータのメインメモリ上での物理アドレスにしたがって一定のアルゴリズムで決定される．キャッシュメモリはメインメモリよりもはるかに容量が小さいことから，キャッシュメモリ上にデータをコピーする場合キャッシュメモリ上の同じ場所に配置され上書きされてしまい，次回アクセス時にキャッシュミスとなることがある．これを競合性ミス (Conflict Miss) と呼ぶ．MMU(Memory Management Unit) によって物理アドレス空間と仮想アドレス空間が分断されている現代の OS の場合，プログラムは物理アドレス空間のどこに展開されるかは実行時ごとに変化するため，再現性が乏しく対策も難しいが，キャッシュライン配置アルゴリズムであるセットアソシアティブ方式の way 数を増やす方法や，プログラム中でアドレスが連続するデータを同一のキャッシュラインとならないようにパディングしキャッシュラインを分離するなどの対策がある．

マルチコアプロセッサ上で複数のプロセスが実行されている場合，実行の単位が複数あるためその分だけこれらのキャッシュミスが発生する頻度は高まる．同一の仮想アドレス空間上で複数の実行単位を持つマルチスレッドプログラムの場合，物理アドレスが近いデータを操作することが多くなるため，競合性ミスの発生確率は更に高くなってしまう．

以上のように，キャッシュミスによって高速なキャッシュメモリが利用できず，低速なメインメモリにアクセスしなくてはならない状況が発生し，メインメモリへのアクセス待ち時間によって CPU の本来の処理性能が発揮できなくなり，プログラムの実行速度が低下する．

1.4 キャッシュメモリの状態を無視したスレッドスケジューリングの問題

複数あるプロセス・スレッドの中からどれを実際に CPU に割り当てて実行するかを決定することをスケジューリングと呼ぶ．これを実際に計算機上で実行しているのが OS のプロセス・スレッドスケジューラ (以下．スケジューラと呼ぶ) である．マルチスレッドプログラムの重要度が増したことで．効率的なスケジューラの重要度も同様に増している．

現在の Linux や FreeBSD といった OS は．スケジューリングする際に各プロセスの状態（各種待ち状態・実行可能状態）とプロセスに割り振られたタイムスライスの消費状況といった値をスケジューリングの判断の指標として使用している．この方法はスケジューラそのものの処理を軽減し．少しでも多くの時間をユーザプログラムの実行に使用するためである．一方，この実装では 1.3 節で述べたようなキャッシュミスの多発によるプログラムの実行速度の低下といった状況を検出することができず．キャッシュミスが大量発生するスケジューリングを行ってしまい実行効率が悪くなってしまう場合がある．

1.5 本研究の目的

本研究は，スケジューリングで使用する指標として新たな情報を利用できるようにし，スケジューリングに介入できるインターフェースを構築することで，従来のスケジューラで発生していた問題を回避し柔軟で効率の良いスケジューリングを実現するための仕組みを提案，実装する．

スレッドごとのキャッシュミスの発生状況を監視してスケジューリングで使用する指標とし，1.3 節で述べた競合性ミスを抑えるスケジューリングを行うことでマルチスレッドプログラムの効率的な実行を目指す仕組みを提案する．

第 2 章

設計

本章では，提案するシステムの設計方針，システム全体の構成，制御の流れについて述べる．

2.1 設計方針

提案するシステムでは以下のような設計方針とする．

- Linux 上で動作する．
- 特定のプロセスで実行されているスレッドのみをスケジューリング介入の対象とする．スケジューリング介入の対象とするプログラム (以下，対象プログラムと呼ぶ) を限定することで，スケジューリングの介入結果に問題があったとしてもその影響を対象プロセスのみに封じ込めることができ，同一 OS 上で稼働している他のプログラムへの影響を抑えられる．
- スケジューリング介入の判断をユーザモードのプログラムとして実装する．スケジューリングに使用したい情報をより柔軟・手軽に追加できるようにユーザモードのプログラムとして実装する．同様に，スケジューリング介入の判断をユーザモードプログラムとして実装する事でカーネルに手を加えることなく対象プログラムに合わせたチューニングが可能となる．
- スケジューリングで使用する情報としてスレッドごとのキャッシュミス値を使用する．
- 対象プログラムはスレッドライブラリとして pthread の使用を前提とする．スレッドを作成する際に広く使用されている pthread に対応することで，対象プログラムとすることができるプログラムを多くできる．
- 対象プログラムには一切変更を加えないこと．対象プログラムへの変更を不要とすることで，バイナリ形式の実行ファイルしかないような既存のプログラムであっても提案システムの効果が得られることとなる．

2.2 システムの全体構成

提案するシステムは大きく分けて 3 つのモジュールから構築される．対象プログラムは pthread を使用している任意のプログラムとする．イメージを図 2.1 に示す．

- スナッチャ：スレッド作成を監視するモジュール．対象プログラムのプロセスのスレッドの作成情報を収集し，スケジューリング介入の判断を行うホストプログラムに伝達する．
- ホストプログラム：スケジューリング介入の判断を行うモジュール．スケジューリング介入に必要な情報を収集し，キャッシュミスが大量発生している場合には特定のスレッドの実行を一時的に停止する判断を行い，カーネルに伝達する．
- ブリッジモジュール：カーネル内のデータに対してスケジューリング介入を実際に反映するモジュール．ホストプログラムからの指示をカーネル内のデータに反映し，実際にスレッドの一時停止を実行する．

2.3 制御やデータの流れ

実際に本システムを使用する場合のモジュール間の制御やデータの流れについて述べる．

1. ターゲットプログラムの実行ファイルのパスを引数にホストプログラムを起動する
2. ホストプログラムがターゲットプログラムを fork-exec で起動する．
3. ターゲットプログラムの実行が進み，pthread によるスレッドが作成される
4. 作成されたスレッドの情報がホストプログラムに報告され，それに合わせて perf を設定する
5. スレッドのキャッシュミス値を監視し，一定の条件にしたがって一時停止するスレッドを決定する．
6. (一時停止するスレッドがある場合) ホストプログラムがブリッジモジュールに一時停止したいスレッドの PID を送信する
7. ブリッジモジュールがカーネル内の該当する PID のプロセス構造体を探し，そのプロセスの状態を待ち状態に変更する
8. ブリッジモジュールが強制的にカーネルのスケジューラに対して再スケジューリングを実行することで，一時停止したいスレッドの実行が実際に停止する．
9. スレッドが停止したのち，このプロセスの状態を再度実行可能状態に変更する．

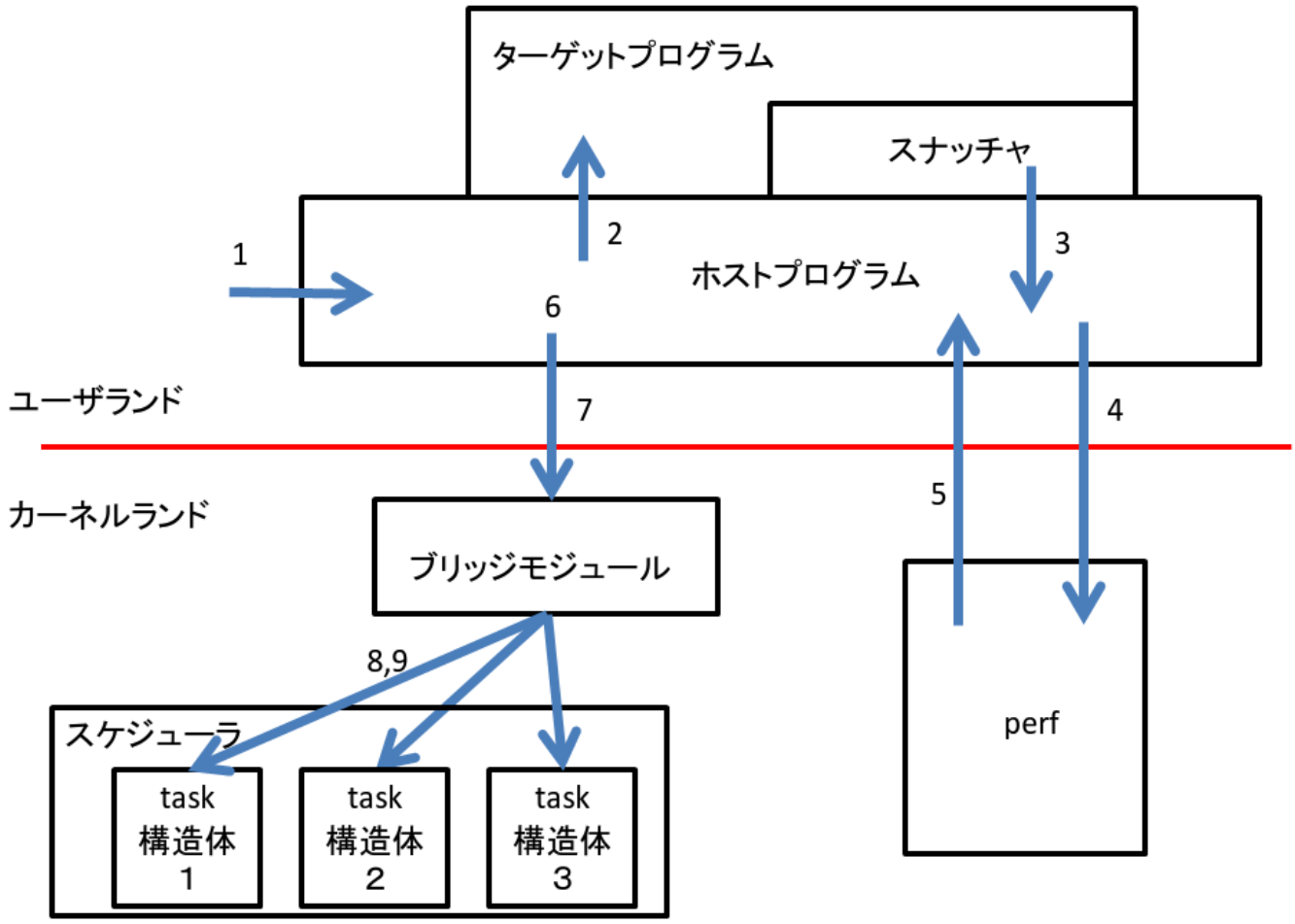

図 2.1　システムの全体構成図

ホストプログラムは，ターゲットプログラムが停止するまで上記 5〜9 の処理を繰り返し実行し，また，スレッドが作成されるたび 3〜4 を実行し 5〜9 処理の対象に加える．

2.4　カーネル内の既存スケジューラを直接拡張する実装の問題点

スケジューラはカーネル内の機能として実装されているため，これを拡張するならば同じカーネルモードのプログラムとして実装するのが自然である．さらに本実装のようなカーネルモードとユーザーモードとのデータのやりとりもコンテキストスイッチの発生によるオーバーヘッドとなりうるため，これを回避するためにもカーネルモードでの実装が

パフォーマンスに有利である．一方で，カーネルモードのプログラムは一般的に実装の難易度が高い．万一プログラムに不具合がありカーネルのメモリを破壊するような場合には容易にカーネルパニックに陥り，システムの安定性を損ねてしまう．また，カーネル内の既存スケジューラはシステム全体のプロセス・スレッドに対して作用しているため，今後実装される独自のスケジューリング手法次第ではスケジューリングの偏りが発生してしまう可能性があり，これもシステムの安定性を損ねる要因となってしまう．

今回提案した手法では 2.1 節で述べたようにユーザーモードのプログラムとして独自のスケジューラを実装し，この結果をカーネル内の既存スケジューラに反映する手法とすることで，難易度の高いカーネルモードのプログラムの実装範囲を減らし，さらにカーネルがユーザモードプログラムに提供するメモリ保護機構等の恩恵によりシステム全体としての安定性は維持される．また，ユーザモードの一般的なプログラムであるため任意のプログラミング言語で容易にプログラムを変更，拡張できる点も優位と考える．また，ユーザモードのプログラムとしてカーネルモードと頻繁な通信が行われることから前述したパフォーマンス上のオーバーヘッドは確かにあるが，今後本提案システムによって実装されるスケジューリング手法によってはパフォーマンス以外の効果が目的となる場合には問題とならない．

これらの理由から，本研究ではスケジューラをユーザモードで行うこととした．

第 3 章

実装

2.2 節で述べた各モジュールについて，それぞれの実装について使用した既存技術を含めて解説する．

3.1 スナッチャ

スナッチャは対象プログラムのメモリ空間に動的リンクによって結合して動作する．そのため作成されたスレッド情報の抜き出しと別プロセスであるホストプログラムとのプロセス間通信が主要な役割となる．

3.1.1 スレッドの作成を検出する

perf は特定のプロセス・スレッドのみのパフォーマンスカウンタの値を参照することが可能となっている．この時のプロセス・スレッドの指定には PID(Process Identifier) と呼ばれるプロセスごとにユニークな数値で指定する． Linux においてスレッドはメモリ空間を他のプロセスと共有している特殊なプロセスとして実装されており，スレッドとプロセスは同列に扱われるため，スレッドであっても PID が存在する．本システムで対象とするスレッドライブラリは pthread なので，スレッド生成関数は pthread_create である．pthread_create の第一引数である pthread_t は，スレッド作成後には様々なデータが格納されており，この中に実際に PID も含まれている．

perf とは

perf とは，Linux カーネル 2.6.31 から提供されている，パフォーマンスカウンタを抽象化して提供するツールである．パフォーマンスカウンタとはプログラムの実行に際して CPU 内部の実行サイクル数や分岐予測の成否，キャッシュアクセスの状況といったメタな統計情報を記録する特殊なレジスタである．パフォーマンスカウンタは MSR(Model Specific Register，CPU 固有レジスタ) であるため，CPU のモデルごとに利用方法が異なっている扱いづらいものであるが，これを抽象化してユーザモードから使用できるようにした機能である．

パフォーマンスカウンタは現在どのプロセスが動いているかについては無視して情報を収集するため，特定のプロセスを対象とした値の取得はできない．perf はカーネル内の情報を利用して特定のプロセスのみの値の取得が可能となっている．また，パフォーマンスカウンタはハードウェアの実装であるため，同時に監視できるイベントの数に上限があるが，perf では時分割多重でパフォーマンスカウンタを切り替えて実行するため上限なく利用することが可能である．Linux-4.3 現在，perf で利用できるパフォーマンスカウンタのイベントの代表例を表 3.1 にまとめる．

表 3.1　perf で監視できる代表的なイベント

イベント名	内容
cpu-cycles	CPU の実行サイクル数
bus-cycles	バスサイクル数
instructions	CPU の実行命令数
branch-instructions or branch-misses	分岐予測の成功，失敗回数
alignment-faults	アラインメントフォルトの回数
page-faults	ページフォルトの回数
cache-refeerence	キャッシュメモリ参照回数
cache-misses	キャッシュミス回数

対象のプログラムを書き換えることなくスレッド作成後の pthread_t 構造体を読み取るために，Linux に固有の LD_PRELOAD 機能を利用する．pthread_create 関数をラップした pthread_t 構造体を読み取る独自の関数を定義し使用する方法もあったが，これは対象プログラムの再コンパイルが必要となってしまうため，2.1 節で述べた「対象プログラムには一切変更を加えないこと」に反するため採用しなかった．

LD_PRELOAD とは

LD_PRELOAD とは，ダイナミックリンク時に任意の動的ライブラリファイルを指定し，合致するシンボルがあればリンク先を上書きする機能である．今回フックする pthread_create 関数は動的にリンクする関数であるため，関数のシグネチャを一致させた独自の pthread_create 関数を定義しておくことで，対象プログラムを書き換えることなく独自の pthread_create を実行することが可能となった．

独自の関数

独自に定義した pthread_create 関数の処理では，本来リンクするはずだった正規の pthread_create を実行してスレッドの作成を行いたいが，この関数はすでに自分自身へリンクされてしまっているため，別の方法で正規の pthread_create 関数のアドレスを取得し呼び出す必要がある．そこでダイナミックリンカに対してシンボル名からアドレスを問い合わせる dlsym 関数を使用する．この際に dlsym の第一引数のシンボルの探索範囲に RTLD_NEXT を指定することで LD_PRELOAD によるシンボルの上書きを無視してシンボルの探索が行われる．これを gcc 拡張機能である constructor アトリビュートを利用してプログラムの起動時に main よりも先に実行することで，本来の pthread_create 関数の呼び出しができつつも LD_PRELOAD による pthread_create の上書きが可能となった．この部分の実際の実装コードをプログラム 3.1 に示す．

```
1 void __attribute__((constructor)) save_original_pthread_create(void){
2   pthread_create_original = (int (*)(pthread_t *thread,
3                                      const pthread_attr_t *attr,
4                                      void* (*start_routine) (void*), void *arg)
5                                      )dlsym(RTLD_NEXT, "pthread_create");
6 }
```

プログラム 3.1　正規の pthread_create 関数のアドレスを取得するコード

これにより，対象プログラムから pthread_create が呼び出されるとリンクが上書きされた独自の pthread_create が実行され，その中では正規の pthread_create を実行し，情報が書き込まれた pthread_t 構造体から新しく生成されたスレッドの PID を取得することが可能となった．

3.1.2　ホストプログラムに伝達するプロセス間通信

このスナッチャは対象プロセスの一部として動作するため，ホストプログラムに PID を渡すにはプロセス間通信が必要となる．本システムではホストプログラムから対象プログラムを fork-exec によって起動するので，事前にパイプを作成しておくことでスナッチャからホストプログラムへ送信する経路を確保した．

3.2　ホストプログラム

ホストプログラムは本提案システムにおいて全体を管理するモジュールとなるため，スレッドのスケジューリングに関連する役割以外にも対象プログラムの起動や終了の管理も行うこととなる。

3.2.1 対象プログラムを起動する

ホストプログラム内から対象プログラムを fork-exec することで起動する．
実際に fork システムコールを実行する前に，

- 前述した LD_PRELOAD を使用するためスナッチャを環境変数に事前に登録
- pipe システムコールによってプロセス間通信を行うためのパイプを作成

している．

3.2.2 スナッチャから送信される新しく起動されたスレッドの PID を受信する

対象プログラムで新しく作成されたスレッドの PID をスナッチャから受け取り，「パフォーマンスカウンタでどのイベントの値を記録するか？」といった設定と一緒に PID を perf_event_open システムコールに渡すと，ファイルディスクリプタが返される．PID とそれに対応する perf のファイルディスクリプタ，この後実際に取得したデータを保存するリングバッファをまとめた構造体に保存する．対象プログラムがスレッドを作成するタイミングはホストプログラムからは関知することができないため，PID の受信と perf の準備はホストプログラムの中の別スレッドによって非同期に処理され，スケジューリングの対象に加えられる．

3.2.3 スケジューリング

スケジューリング手法に関しては，本論文では提案に留め，実装および評価は行わない．以下に現在検討しているスケジューリング手法の詳細と実装を述べる．

受信した PID からキャッシュミス値を取得する

前述した perf_event_open システムコールの返り値のファイルディスクリプタを 64bit 符号無し整数として read システムコールを実行すると設定した PID の設定したパフォーマンスカウンタの値を得ることができる．各スレッドごとのキャッシュミス値を取得し構造体内のリングバッファに記録し，随時移動平均を計算する．perf_event_open システムコールの引数の設定から実際の呼び出し，read システムコールでキャッシュミスの値を取得するコードをプログラム 3.2 に示す．

```
1 // perf_event_openに引数として渡す
2 // 「どのパフォーマンスカウンタの値を取得するか？」を
3 // 設定する構造体を作る関数
```

```
 4 struct perf_event_attr *getPerfEventAttr_cacheMiss(){
 5   struct perf_event_attr *event_cachemiss;
 6   event_cachemiss = calloc(1, sizeof(struct perf_event_attr));
 7
 8   event_cachemiss->inherit = 1; // 対象プロセスの子スレッドを監視対象にしない
 9   event_cachemiss->type = PERF_TYPE_HW_CACHE;
10   event_cachemiss->config = PERF_COUNT_HW_CACHE_LL << 0 |
11     PERF_COUNT_HW_CACHE_OP_READ << 8|
12     PERF_COUNT_HW_CACHE_RESULT_MISS << 16;
13   // ここまででラストレベルキャッシュの参照時の
14   // キャッシュミスの回数を指定している
15
16   return event_cachemiss;
17 }
18
19 // perf_event_openシステムコールはC言語ラッパー関数が用意されていないため,
20 // C言語から呼び出すためにはsyscall関数を利用する必要がある.
21 int sys_perf_event_open_wrapper(struct perf_event_attr *attr,
22                                 pid_t pid,
23                                 int cpu,
24                                 int group_fd,
25                                 unsigned long flags){
26   return syscall(298, attr, pid, cpu, group_fd, flags);
27 }
28
29 int fd = sys_perf_event_open_wrapper(getPerfEventAttr_cacheMiss(),
30                                      pid, -1, -1, 0);
31
32 unsigned long int cachmiss_value;
33 // perf_event_openで取得したファイルディスクリプタを
34 // 64bit符号無し整数としてreadすると
35 // ラストレベルキャッシュの参照時のキャッシュミスの回数を取得できる
36 read(fd, &cachemiss_vlaue,sizeof(unsigned long int));
```

プログラム 3.2　perf_event_open システムコールを呼び出しているコード

キャッシュミス値が増大したスレッドがあった場合，スレッドを一時停止させる

キャッシュミス値の移動平均の値が 10 %以上上昇した場合，そのスレッドは他スレッドとキャッシュの競合を起こしていると判断し，スレッドの実行を一時停止して他のプロセスやスレッドに CPU を明け渡すことを決定する．特定のスレッドを一時停止させる際は，次節で解説するデバイスファイルに対して ioctl システムコールで停止したいスレッドの PID を伝達する．

3.3 ブリッジモジュール

ブリッジモジュールはカーネルモジュールとして本システムの中で唯一カーネルモードで動作するプログラムとなる．ブリッジモジュールはホストプログラムからのスケジューリング命令の受信とそれを実際のスケジューラに反映する役割がある．

3.3.1 ホストプログラムからのスケジューリング命令の受信

このカーネルモジュールでは，カーネルモジュールのロード時に/dev に独自のスペシャルファイルを作成する．このスペシャルファイルに対してホストプログラムが ioctl システムコールを実行することで実行を一時停止したいスレッドの PID を受信する．

3.3.2 スケジューリング命令のスケジューラへの反映

カーネルは自身が管理しているプロセス・スレッド全てにそれぞれの task 構造体を作成して管理しており，この中に各プロセス・スレッドのタイムスライスや実行状態といった情報が保存されている．受信した PID から対応する task 構造体を探し，その実行状態を実行可能状態から待ち状態に書き換え，強制的に再スケジューリングさせることでそのプロセス・スレッドの実行を停止する．再スケジューリングが完了した段階ですぐに待ち状態から実行可能状態に戻すことで，ほかに実行したいプロセス・スレッドがない場合等の理由からスケジューラが停止したスレッドを再度動かす必要があると判断したらすぐに動き出すことも可能となる．

第 4 章

評価

本章では，提案したシステムの評価を行う．

本論文ではスケジューラについては実装は行わなかったことから，本提案システムが適切に動作していることの確認として，プログラムの起動から終了までの間にキャッシュメモリへのアクセス状況がどのように推移するかを測定を行う．

4.1 実験環境

評価に用いた計算機環境を表 4.1 に示す．

表 4.1 評価に用いた計算機環境

OS	Linux 4.3.0
CPU	Intel Xeon E5-2640 v3 @ 2.60GHz 物理 8 コア 論理 16 コア
LastLevelCache	全コア共有 20 MB
メインメモリ	16 GB

4.2 実験

提案システムのうち，スケジューリングのかわりにスレッドごとに取得したキャッシュヒット回数，キャッシュミス回数をダンプすることで，あるプログラムの起動から終了までの間のキャッシュミス量の推移を可視化を行う．対象とするプログラムは評価に使用した CPU のキャッシュメモリに乗り切る程度の小さいワーキングセットを持つプログラム (以下，小さなワーキングセットのプログラムと呼ぶ) と明らかにキャッシュメモリに絶対に乗り切らない巨大なワーキングセットを持つプログラム (以下，巨大なワーキングセットのプログラムと呼ぶ) を作成し計測した．

4.2.1 小さなワーキングセットのプログラムでのキャッシュミスの推移

図 4.1 に小さなワーキングセットのプログラムでプログラムのキャッシュヒット・ミスの回数の 1 ミリ秒ごとの推移のグラフを表す．横軸は時間の経過を表し，1 ミリ秒ごとに記録されている．縦軸は左右で別の数値を持ち，左の縦軸はある時点での回数の数値を表し，右の縦軸はある時点までの累積の回数を表す．緑の十字記号はある 1 ミリ秒でのキャッシュヒットの回数を，赤いバツ記号はある 1 ミリ秒でのキャッシュミスの回数を表す．これらは左の縦軸に対応している．青いアスタリスク記号はその時点までのキャッシュヒット回数の累積を，紫の四角記号はその時点までのキャッシュミス回数の累積を表す．これらは右の縦軸に対応している．

キャッシュメモリに乗り切るワーキングセットのプログラムであることから，ほとんどのメモリアクセスでキャッシュメモリ上のデータがヒットする形となる．そのためキャッシュヒット回数を表す緑の十字記号が全域で高い数値を示している一方で，キャッシュミス回数を表す赤いバツ記号数値はほとんど 0 に近い低い数値を示し続けている．この 2 つのプロットの高さの差が大きいほどキャッシュヒット率が高いと言える．したがって，キャッシュヒットの累積回数を表す青いアスタリスク記号のプロットとキャッシュミスの累積回数を表す紫の四角形のプロットの傾きの差が大きくなる．

4.2.2 巨大なワーキングセットのプログラムでのキャッシュミスの推移

図 4.2 に巨大なワーキングセットのプログラムでプログラムのキャッシュヒット・ミスの回数の 1 ミリ秒ごとの推移のグラフを示す．グラフの見方は 4.2.1 と同様である．

4.2.1 とは違い，キャッシュメモリに絶対に乗り切らない巨大なワーキングセットであることからキャッシュヒット回数を表す緑の十字記号のプロットと赤のバツ記号のプロットが全域にわたって非常に近い高さにあり，キャッシュヒットとキャッシュミスがほぼ同数発生しキャッシュメモリを有効活用できていないことを示している．それぞれの累積回数を表す青いアスタリスク記号のプロットと紫の四角形のプロットの傾きもほぼ同一となっている．

4.2.3 実験まとめ

以上のように，スレッドのワーキングセットの大小によってキャッシュミス回数も大きく変化する形を可視化できた．これによって，スケジューリングを除いた本提案システムが正常に動作していることが確認できた．

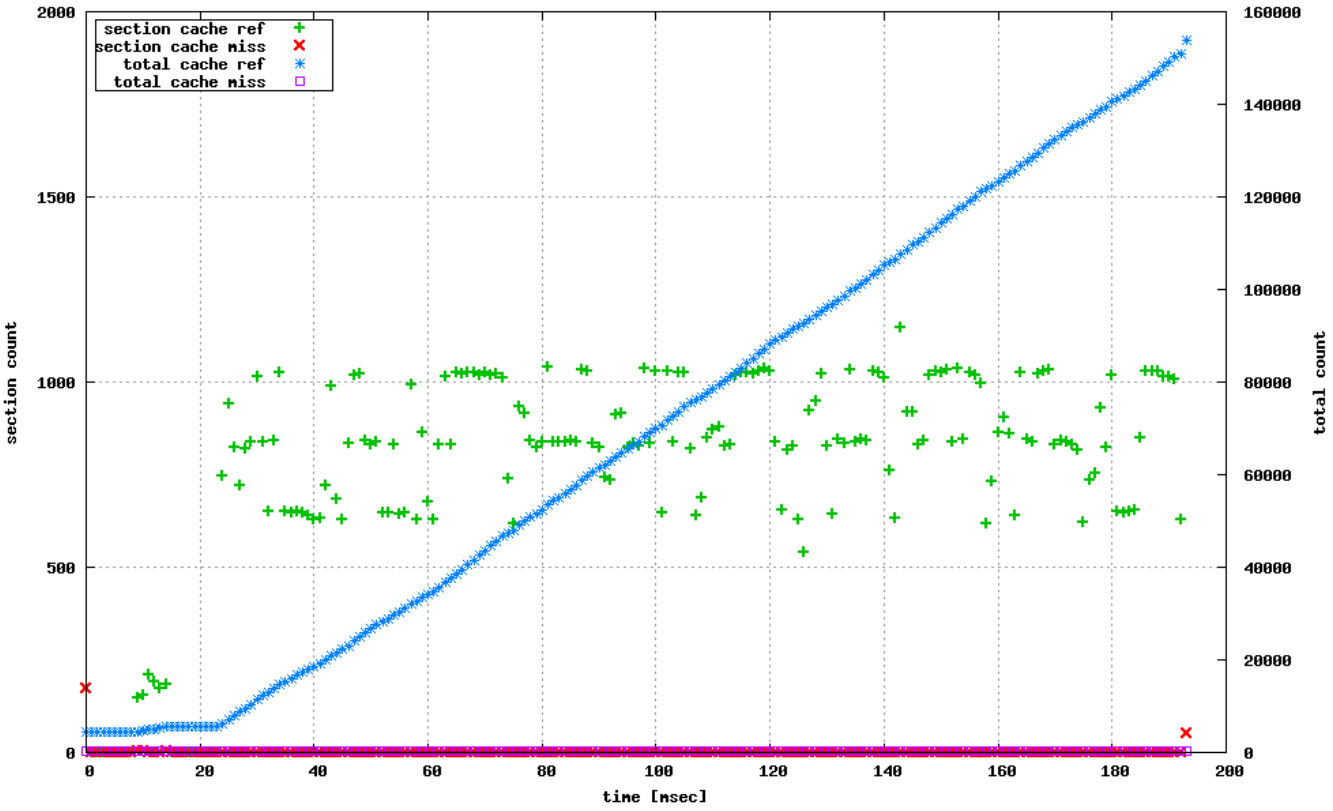

図 4.1　小さなワーキングセットのプログラムでのキャッシュミスの推移のグラフ

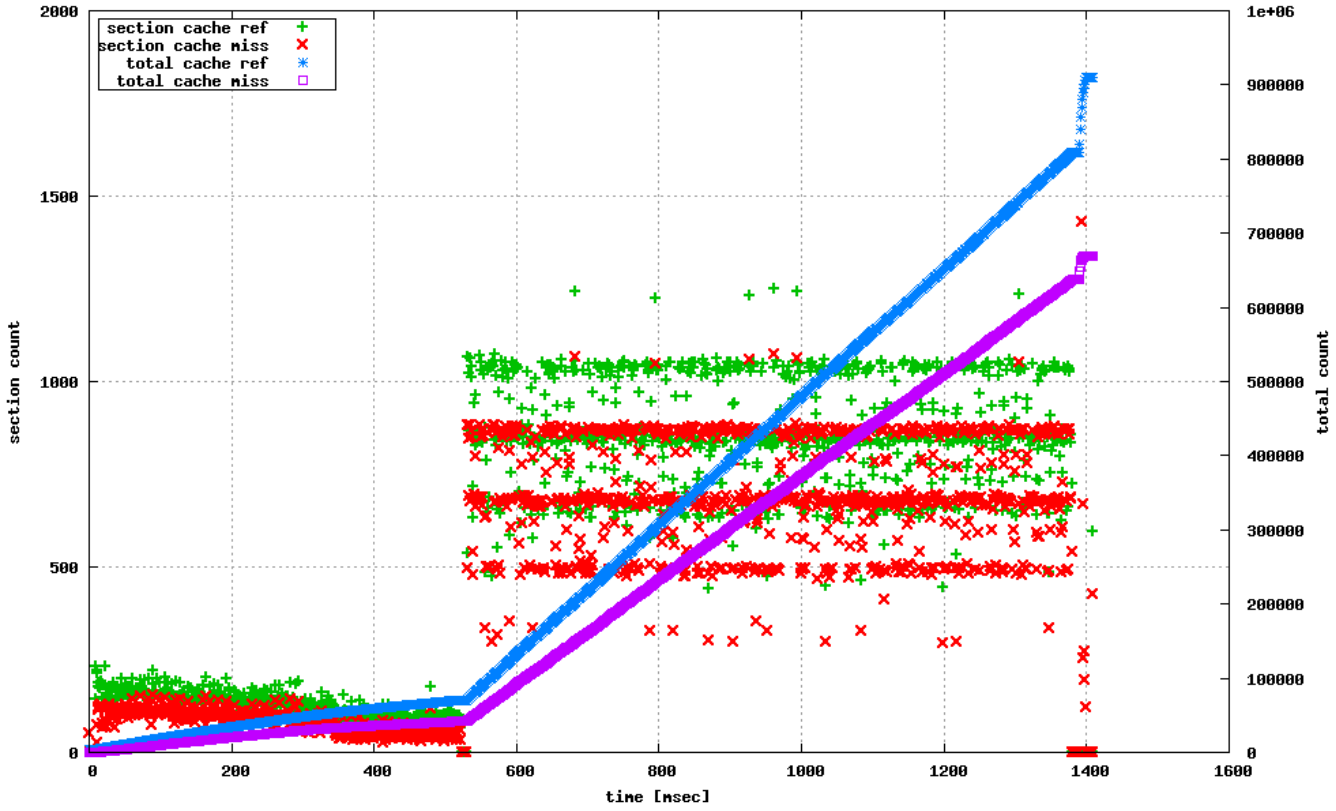

図 4.2　巨大なワーキングセットのプログラムでのキャッシュミスの推移のグラフ

第 5 章

関連研究

本章では，関連研究について述べ，本研究との比較を行う．

5.1 各スレッドの実行時の情報の取得

各スレッドが持つキャッシュメモリの使用状況やスレッド間でのデータの共有状況といった実行時の情報を取得する関連研究について述べる．ただし，これらの研究はスレッドのワーキングセットの情報の収集方法についてのみを対象としているため，本研究で取り扱っているスケジューリングに関する部分は対象としていない点が共通した差異である．

- Qin らの研究 [8] は Memory Shadowing を用いた動的解析によってスレッド間の実行依存関係，データの共有関係，キャッシュメモリの使用量等の情報を取得する手法を提案している．この動的解析は，プログラムの実行時間が通常の実行と比べておよそ 5 倍程度悪化するとされており，実行時間の悪化が顕著であることからリアルタイムにスケジューラに反映する本研究では解析手法としての使用は難しい．
- キャッシュを含むメモリ全般の解析ツールとしてオープンソースのプロダクトとして Valgrind[9] が存在する．
- Valgrind からキャッシュメモリの状態のシミュレーションに特化した Cachegrind[10] が存在する．

5.2 キャッシュヒット率を考慮したスレッドのスケジューリング

キャッシュヒット率を考慮したスケジューリングに関する研究は既に複数存在している．それぞれについて本研究との違いを述べる．これらの研究はすべて提案しているスケジューラがカーネルモード内で実装されているが，本研究ではそれをユーザモードで実装している点が共通する差異となる．

- 内倉らの研究 [4] は，SMT(Simultaneous MultiThreading) プロセッサ上のスレッドを対象としている．SMT プロセッサは原則として CPU 内の状態を保存する部分以外のすべてを共有しているため，マルチコアプロセッサと比べてさらにリソースの競合によるパフォーマンス低下が発生しやすい．パフォーマンスカウンタからキャッシュラインの追い出し回数を監視し，この追い出し回数が少なくなるスレッドの組み合わせを相性が良いと定義し，この組み合わせで実行されるスケジューリング方式を提案している．本研究では SMT プロセッサではなくマルチコアプロセッサを対象としている点で差異がある．
- 本橋らの研究 [5] は, マルチコアプロセッサ上で実行される VMM(Virtual Machine Monitor) の物理 CPU とゲストの仮想 CPU のスケジューリングを対象としている．仮想 CPU で実行されているプログラムのワークロードを判別して Memory-bound か，仮想 CPU が物理 CPU コアをまたがっていないかを監視し，Memory-bound なゲストを持つ仮想 CPU を同じ物理 CPU コアにまとめることでキャッシュミスの発生を抑制する．本研究は VMM ではなくネイティブで実行される Linux を対象としている点，実行されているプログラムのワークロードについては関知していない点，パフォーマンスカウンタによって実際のキャッシュの状態を監視している点が異なる．
- 佐藤らの研究 [6] は，一般的なマルチコアプロセッサ上で実行されるプログラムを対象とし，プログラムごとのワーキングセットの空間的局所性を監視し，このサイズに応じたキャッシュパーティショニングを行うことでキャッシュ競合を抑制，さらに余剰となったキャッシュメモリの電源供給を遮断することで消費電力を削減する手法を提案している．本研究はワーキングセットのサイズなどの情報は使用せず，リアルタイムなキャッシュミス値の監視によってスレッド間の相性を判定している点，消費電力の削減は目的としていない点で異なる．
- Zhuravlev らの研究 [12] はメモリコントローラやメモリバス，ハードウェアプリフェッチ等で発生するリソース競合を削減するためのスケジューリング手法を提案している．本研究ではメモリコントローラやメモリバスといったハードウェアに起因するリソース競合については関知していない点で異なる．

第 6 章

今後の課題

本章では本研究の今後の課題や発展について検討する.

6.1 スケジューラの実装と評価

本論文では，スケジューラについては提案にとどめ，実装および評価は行わなかった. 提案した手法についても，

- キャッシュ競合によるキャッシュミスが大量発生しているが変化が小さいため検出できない
- キャッシュ競合によるキャッシュミスの増加率は大きいが無視できる程度に低い
- あるスレッドがキャッシュ競合しているとしても，どのスレッドと相性が悪いのか検証できない

といった場合に，非効率なスケジューリング判断をしてしまう可能性がある.

さらに，先行する研究において様々なスケジューリング手法が提案されている [1] ため，本システムにも適用可能か実験する必要がある.

本研究のシステムは，スケジューラをユーザモードのプログラムとすることで実装・拡張の容易さを実現しているためこれらの課題の調査を行うことが可能であると考える.

6.2 スレッド間の相性判定手法の追加

キャッシュメモリの状態を崩さないスレッド間の相性判定の手法としてメモリアクセスの絶対量が多いスレッドと少ないスレッドを同時に実行させることなどが考えられる. また現在の実装では，ホストプログラムは各スレッドごとのラストレベルキャッシュのロードミス値しか取得していない.

perf ではこれ以外にもラストレベルキャッシュへの書き込み回数や L1 キャッシュも同様にミス・アクセス回数を取得できるようになっているためこれらの情報の活用も考えられる.

6.3 ホストプログラムの負荷低減

ホストプログラムがスレッドを停止するかの判断を行う際，現在の実装では各スレッドごとにキャッシュミス値を保存したリングバッファから移動平均値を計算している．これによってホストプログラムが比較的広いメモリ空間を操作することになり，ホストプログラムと対象プログラムの間でキャッシュ競合によるキャッシュミスが発生してしまう可能性がある．各リングバッファは 64bit(unsigned long int) × 100 要素の配列によって構成され，1 キャッシュラインのサイズを超えることから 1 つのリングバッファの計算で複数のキャッシュラインでキャッシュライン入れ替えが発生しキャッシュ競合につながる可能性がある．

データ構造の改善や 6.1 節で述べたスケジューリング手法の評価と関連してスレッド一時停止判定アルゴリズムの調整によって改善する可能性がある．

6.4 pthread 以外のスレッドシステムへの対応

現在の実装では，対象プログラムには pthread を使用することを前提としている．Linux の場合，clone システムコールを直接呼び出してスレッドを作成する事が可能であるため，この方法でカーネルスレッドを作成されるとスレッドの生成を認識できない問題がある．

現在の実装で使用している LD_PRELOAD で clone システムコールの C ラッパー関数をフックすることで実現可能と考える．

6.5 マルチプロセスへの対応

マルチプロセスではプロセスごとにメモリ空間が独立しているため，メモリ空間を共有しているマルチスレッドと比べるとキャッシュ競合が発生する頻度は低くなると考えられる．本研究では pthread によるマルチスレッドにのみ対応しているが，マルチプロセスプログラムにも本手法が拡張可能と考える．

第 7 章

まとめ

7.1 まとめ

一般的な OS で使用されている既存のプロセス・スレッドスケジューラはスケジューリングの際に使用している情報が少ないことから，実行効率の悪いスケジューリングが行われてしまうことがあった．そこで，本研究ではユーザモードのプログラムで必要な情報を収集し，それをもとに独自のスケジューリングを行い，その結果をカーネル内の既存のスケジューラに対して反映する仕組みを構築した．

実際の本システムを実装し，スケジューリングの代わりにリアルタイムでキャッシュメモリへのアクセス状況を表示するプログラムを追加し，プロセスがワーキングセットのサイズによってキャッシュメモリへのアクセス状況が大きく変化することを示し，キャッシュメモリの状態がパフォーマンスに大きく影響することを確認した．これによりスケジューリングを除いた本システムが適切に動作することを示した．

参考文献

[1] Zhuravlev, Sergey, Saez, Juan Carlos, Blagodurov, Sergey, Fedorova, Alexandra, and Prieto, Manue: “Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors.” ACM Comput. Surv. Vol.45, No.1, pp.4:1–4:28

[2] Nestbit, K. J., Aggarwal, N., Laudon, J., and Smith, J. E.: Fair queuing memory systems. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39). pp.208–222.

[3] 小川 周吾, 平木 敬,: “プロセスの実行時情報を用いたスケジューラによる高速化手法”, 情報処理学会論文誌. コンピューティングシステム (SIG_12(ACS_11)), Vol.46, No.12, pp.161–169, 2005.

[4] 内倉 要, 他,: “SMT プロセッサにおけるスレッドスケジューラの開発”, 情報処理学会論文誌. コンピューティングシステム (SIG_12(ACS_11)), Vol.46, No.12, pp.150–160, 2005.

[5] 本橋 剛, 山田 浩史, 吉田 哲也, 河野 健二,: “マルチコア CPU 環境における L2 キャッシュの影響を考慮した VM スケジューラ”, 情報処理学会研究報告. システムソフトウェアとオペレーティング・システム 2009-OS-112(1), pp.1–9

[6] 佐藤 雅之, 小寺 功, 江川 隆輔, 滝沢 寛之, 小林 広明,: “ワーキングセット評価に基づくスレッドスケジューリング”, 情報処理学会研究報告. 計算機アーキテクチャ研究会報告 2009-ARC-184(11), pp.1–10

[7] Chen, Shimin, Gibbons, Phillip B., Kozuch, Michael, Liaskovitis, Vasileios, Ailamaki, Anastassia, Blelloch, Guy E, Falsafi, Babak , Fix, Limor, Hardavellas, Nikos, Mowry, Todd C., and Wilkerson, Chris,: “Scheduling threads for constructive cache sharing on CMPs”, Proceeding SPAA ’07 Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures, pp.105–115

[8] Zhao, Qin, Koh, David, Raza, Syed, Bruening, Derek, Wong, Weng-Fai, and Amarasinghe, Saman,: “Dynamic Cache Contention Detection in Multi-threaded Applications”, Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, Vol.46, No.7, pp.27–38 (Mar. 2011).

[9] Valgrind http://valgrind.org/

[10] Cachegrind http://valgrind.org/docs/manual/cg-manual.html

[11] Sujay Parekh, Susan Eggers, Henry Levy, and Jack Lo,: “Thread-Sensitive Scheduling for SMT Processors”, Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’ 10). pp.129–142.

[12] Zhuravlev, Sergey, Blagodurov, Sergey, and Fedorova, Alexandra,: “Addressing Shared Resource Contention in Multicore Processors via Scheduling”, Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’ 10). pp.129–142.