Vol.2012-OS-120 No.1
2012/2/28

OS カーネルにおける SIMD ユニットの活用

斎藤　奨悟[†1]　　追川　修一[†2]

近年，アプリケーションの動作を高速化するため，プロセッサが SIMD (Single Instruction Multiple Data) ユニットを搭載することが一般的となっている．SIMD ユニットは，動画像など，大きなデータ処理の高速化に用いられるが，システムソフトウェアで利用されることは想定されていない．しかし，OS (Operating System) カーネルは多くのデータを扱う特徴を持つため，SIMD ユニットを活用することで高速化を図る事が出来る箇所があると考えられる．本研究では、システムソフトウェアにおける SIMD ユニットを活用するための考察および，検証を行った．また，実際の適用例として，OS カーネルにおけるデータコピーにまつわる処理の高速化を確認した．

Utilization of a SIMD unit in the OS Kernel

Shogo Saito[†1] and Shuichi Oikawa[†2]

Nowadays, it is very common that a processor includes a SIMD (Single Instruction Multiple Data) unit in order to accelerate application processing. While a SIMD unit is a part of a processor, it evolves more rapidly than the integer unit of the processor. Since the use of an FPU (Floating Point Unit) and a SIMD unit is basically abandoned from the kernel, there can be places inside the kernel where a SMID unit works effectively to deal with a large amount of data processing. This paper describes our preliminary work to explore the possibility to utilize a SIMD unit in the kernel. We performed preliminary experiments by using UML (User Mode Linux) and show that data copying can be improved.

†1 筑波大学 システム情報工学研究科
University of Tsukuba
†2 筑波大学 システム情報系情報工学域
University of Tsukuba

1. はじめに

近年，アプリケーションの処理速度向上を目的とし，多くのプロセッサが SIMD (Single Instruction Multiple Data) ユニットを搭載している．SIMD ユニットを利用することで，多くのデータを一度に演算することができる．この機能により，SIMD ユニットによりマルチメディア用途やデータ圧縮，暗号化などで高い性能向上が得られることが知られている．また，大量のデータ処理を行う必要があるデータマイニングなどの分野でも，SIMD ユニットは高い性能を発揮する．近年のプロセッサに搭載される SIMD ユニットは，レジスタの数の増加やビット幅の増加，命令の追加など，より性能向上が進んでいる．OS カーネルは，入力と出力，双方について大量のデータを扱う．このため，OS カーネルでも大量のデータ処理をどのように行うかが重要である．現在の OS カーネルでは FPU (Floating Point Unit) や SIMD ユニットは通常使われていないが，これらのユニットが OS カーネルの性能を向上させる余地があると考えられる．そこで本稿では，SIMD ユニットを OS カーネルで利用するにあたり，どれほどの性能向上が得られるかを検討した結果について述べる．通常の OS カーネルは，SIMD ユニットを使う事は想定していない．さらに，プロセッサも，SIMD ユニットがシステムプログラムから利用される事を想定していない．このため，SIMD ユニットを OS カーネルで利用するには，OS カーネルに大規模に手を加える必要があるが，SIMD ユニットの有用性を調べる用途としては非効率的である．そこで，本研究では UML (User Mode Linux) を利用し，OS カーネルにおける SIMD ユニットの有用性について調査を行った．

2. SIMD ユニット

SIMD ユニットは，一般に汎用プロセッサの一部として組み込まれている演算器であり，主にマルチメディア用途で大量のデータを演算する際に使われる．SIMD ユニットのサポートする命令セットのうち，広く普及している物としては，Intel の x86 アーキテクチャに搭載されている SSE, SSE2, SSE3, AVX 命令セットや，ARM アーキテクチャに搭載されている NEON などが挙げられる．SIMD ユニットは，複数データを一つの命令で演算することができる点が最大の特徴である．SIMD ユニットは複数のレジスタを持つ．これらに処理したいデータを格納した後に，データ処理用の命令を呼び出すことで，複数のレジスタそれぞれに対し演算が行われ，結果が格納される．これによりデータ処理に必要な演算命令の呼び出し回数を減らすことができ，処理時間の短縮ができる．SIMD ユニットを用いた演算

Vol.2012-OS-120 No.1
2012/2/28

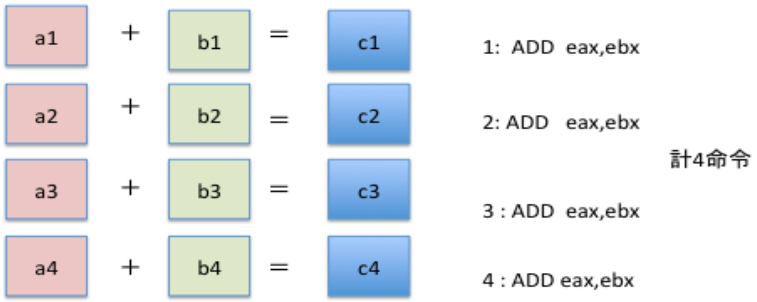

図 1 通常の汎用演算命令による加算

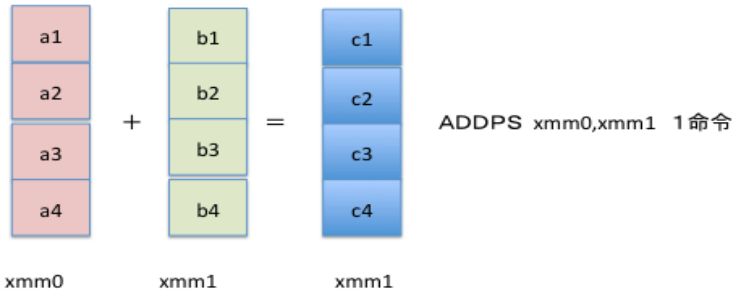

図 2 SIMD ユニットを利用した加算

が高速に実行できる例として，4 組の数の和を求める計算を考える．図 1 に汎用演算命令による加算を，図 2 に SIMD 命令を用いた加算の様子を示す．汎用演算命令による加算では，演算に必要な命令数は 4 個となる．一方で，SIMD ユニットを用いた場合の演算に必要な命令は一つである．さらに，メモリからレジスタへのロードおよびストアの回数も削減することが出来る．このような演算を複数回連続して実行する場合，非常に効率よく計算を行うことが出来るため，この差はより大きくなっていく．このように，SIMD ユニットを用いた高速化は，複数のデータに同一の処理を行うような場合に特に有効である．この SIMD ユニットの特徴は，マルチメディア用途などの，大きなデータ処理に適している．この特徴により，SIMD ユニットを利用する事で，通常の整数演算では長く計算時間のかかる処理を，短時間で処理することが可能である．

SIMD ユニットは特定のマルチメディア用途のアプリケーションでは盛んに利用されているが，ほとんどのシステムプログラムでは利用されていない．これは，プロセッサの設計が，SIMD ユニットが OS などのシステムプログラムが動作する特権モード上でを利用されることを想定していないためである．例として，以下のような問題が挙げられる．SIMD ユニットは整数演算ユニットとは並行に動作するため，ユーザプログラムが SIMD ユニットを使っている最中に OS に制御が移り，OS が SIMD ユニットを利用しようとした場合に問題が発生する．このとき，ユーザプログラムが SIMD ユニットによる演算を完了するまで，OS は処理を一時的に停止してしまうためである．また，OS カーネル側も SIMD ユニットを利用する事を想定していない．また，近年，汎用プロセッサに搭載されている SIMD ユニットの性能向上が進んでいる．SIMD ユニットの利用する専用のレジスタの数の増加や，ビット幅の増加，また AES 暗号化向けの特殊命令が追加されるなど，機能が拡張されつつある．これにともない，SIMD ユニットの用途は従来のマルチメディア用途以外にも拡張され，さまざまな用途において利用する事が可能となりつつある．これらの背景から，SIMD ユニットの適用範囲を動画像処理以外にも拡張し，カーネルに適用することは，今後のアーキテクチャの進歩を踏まえても有望であると考えられる．

3. UML (User Mode Linux)

前項で述べたとおり，SIMD ユニットは現在の一般的な OS カーネル内から利用した場合，OS カーネルが予期しない動作をする可能性がある．このため，通常のカーネル内において SIMD ユニットを利用するためには，カーネルに変更を加える必要がある．しかし，この変更はカーネルに大きなオーバーヘッドを生じさせると予想される．これは，本研究の目的である OS カーネル内部における SIMD ユニットの有用性を検討する上で障害となる可能性がある．そこで，本研究では UML (User Mode Linux) を用いて OS カーネルにおける SIMD ユニットの有用性について検証を行った．UML は，Linux 上で，単一プロセスとして独立して動作する Linux カーネルである．UML はユーザモードで動作するため，SIMD ユニットのシステムレベルでの利用に関する制限を受けない．UML の動作する状況と，SIMD へのアクセスを表す図 3 に示す． ユーザレベルで動作する UML からは SIMD に自由にアクセスする事が出来るが，特権モードで動作するホストの Linux カーネルから

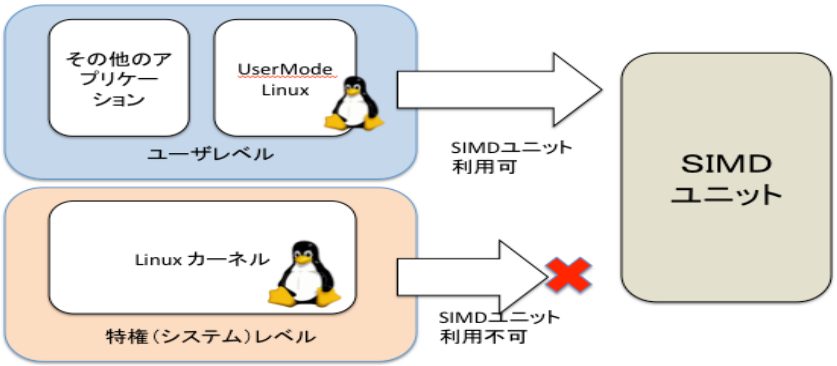

図 3 最適化されるコードの例

```
int a[256],b[256],c[256]
foo () {
    int i;

    for( i = 0 ; i < 256 ; i++)
    {
        a[i] = b[i] + c[i];
    }
}
```

図 4 最適化されるコードの例

は SIMD ユニットは自由に利用することができない．UML は単一バイナリとして提供され，動作中にプロセッサの特権モードを利用する事はない．UML は Linux のカーネルにおいてアーキテクチャの一部として定義されており，カーネルビルド時にビルドオプションのアーキテクチャに指定することでビルドすることができる．なお，ユーザモードで動作するソフトウェアは，SIMD ユニットを制限無く利用する事が出来る．これにより，システムへの影響を考慮することなく，OS カーネルにおける SIMD ユニットの有用性を調査することが出来る．

4. 自動ベクタ化

自動ベクタ化は，GNU Compiler Collection (GCC) により，最適化オプションとして提供されている．この機能は，GCC が C 言語のコードを静的に解析し，SIMD を利用するコードを出力する．コンパイラによる自動ベクタ化機能を利用せずに SIMD ユニットを利用する場合，プログラマが SIMD ユニットを活用するコードを，アセンブリで記述する必要がある．自動ベクタ化は C 言語のコードを静的に解析し，コンパイラにより自動的にコードのベクタ化を行い，SIMD を使ったコードを出力する．これにより，プログラマはベクタ化について深く意識せずにコードを記述することが出来る．このように， 自動ベクタ化はプログラマへの負荷を軽減し，SIMD ユニットの活用によりアプリケーションの性能を向上させる．自動ベクタ化の最適化オプションは，GCC のコンパイル時に ftree-vectorize オプションを追加することで有効となる．GCC はこのオプションを指定された場合，ソースコードを解析し，SIMD ユニットを活用するように最適化およびコンパイルを行う．自動ベクタ化の対象は，主にコードの中のループ処理である．これらのうち，ループごとの処理が，前のループの処理に依存していないものは並行に演算が可能といえる．GCC による自動ベクタ化では，これらを静的に解析することで，SIMD ユニットを使った処理に置き換える．図 4 に最適化されるコードの例を示す．この例では，int 型の変数 a[256], b[256] , c[256] を相互に加算している．たとえば，a[5] = b[5] + c[5] と，a[6] = b[6] + c[6] では，演算内容に依存関係は無い．このため，これらの計算を SIMD ユニットを使い，同時に計算を行うよう置き換える事が出来る．しかしながら，自動ベクタ化にはいくつかの制限が存在する．例として，一般に自動ベクタ化が行われるループは，静的解析によりループ回数が確定できる物に限られる．ループ回数が実効時に動的に変化するループは，SIMD 命令に置き換える事が難しく，自動ベクタ化が行われない．

5. 予 備 実 験

SIMD ユニットの OS カーネルにおける有用性を調査するため，いくつかの項目について計測を行った．カーネルのコードは大規模であるため，コンパイラによる自動ベクタ化が有効な箇所が存在すると予想し，自動ベクタ化を適用する実験を行った．そこで，まず，自動ベクタ化を Linux カーネルに適用し，どの箇所について自動ベクタ化が行われるかを調査した．次に，自動ベクタ化を適用したカーネル上でベンチマークを実行し，動作のプロファイリングを行った．これにより，自動ベクタ化により，OS カーネルが高速化されているか

について検証を行った．また，ソフトウェアの高速化では，一般に，実行時間が集中するホットスポットを高速化することが効率的である．このホットスポットを探すため，gprofを用いて動作のプロファイリングを行い，時間を多く消費している関数を調査した．

5.1 評価環境

SIMD ユニットの有用性を調べる実験のため，UML を利用した．UML の Linux カーネルのバージョンは 2.6.39.3 を利用し，コンパイラには GCC 4.6.1 を利用した．自動ベクタ化は，コンパイラのバージョンにより挙動が異なるため，ベクタ化の性能が高いと考えられる比較的新しいバージョンのコンパイラを利用した．本稿の実験は，全て IBM ThinkPad X1 を用いて行った．CPU には Intel Core-i5 2520M 2.50GHz，RAM は 4GB を搭載している．Intel Core-i5 2520M は，Intel の SIMD 命令セットである SSE3 および，AVX をサポートしている．

5.2 自動ベクタ化の UML への適用

UML を GCC の ftree-vectorize オプションおよび，ftree-vectorize-verbose オプションを追加し，コンパイルを行った．また，バイナリサイズの最適化オプションは削除した．これは，ベクタ化とバイナリサイズの最適化は両立できないためである．ftree-vectorize オプションは GCC に自動ベクタ化を行うように通知するオプションである．ftree-vectorize-verbose オプションは，コンパイラによるソースコードの静的解析および，自動ベクタ化の結果をログとして出力する機能である．これにより，どの箇所がベクタ化されたかを得ることができる．この結果により，カーネルのソースコードのうち，23 箇所のループが GCC により自動ベクタ化されたことがわかった．表 1 に，自動ベクタ化された箇所を示す．これらのループについて調査したところ，いくつかの共通点が見つかった．もっとも多いパターンは，ループ内で配列を 0 でクリアしている箇所である．これらは固定の領域について，ループ内で変数に 0 を格納するので，静的解析が働きやすかったものと考えられる．

5.3 考　　察

今回の実験により得られた自動ベクタ化箇所は，変数のゼロクリアの処理時間など，全体の実行時間に占める割合が小ささく，処理時間への影響が小さなものばかりであった．これらの結果から，OS カーネルへの単純な自動ベクタ化の適用では，出力されるコードには大きな変化は見られないことが判明した．自動ベクタ化が効率的に行われない原因としては，OS カーネル内のループ処理は入力されるデータにより，処理内容が変化する事が多い．このため，ソースコードの静的解析時にループ回数が判明せず，自動ベクタ化を行うことが出来なかったものと考えられる．また，OS カーネル内部のデータ構造には単純な配列ではなく，柔軟性の高いリンクを用いた構造が多いことも原因として考えられる．

表 1 Linux カーネル内の，自動ベクタ化されたループ一覧

source file	vectorized loop num
/arch/um/drivers/drivers/slip_user.c	1
/mm/vmstat.c	1
/fs/ext2/inode.c	2
/fs/ext3/inode.c	2
/fs/ext3/hash.c	2
/fs/isofs/util.c	1
/fs/reiserfs/fix_node.c	1
/drivers/base/map.c	1
/net/ipv4/inet_hashtables.c	1
/net/ipv4/tcp_input.c	1
/net/ipv4/devinet.c	1
/lib/sort.c	1
/lib/bitmap.c	5
/lib/cmdline.c	1
/net/core/dev.c	1
/crypto/algapi.c	1

5.4 UML の動的プロファイリング

SIMD ユニットの OS カーネルにおける有用性は，自動ベクタ化のみでは十分に計測することはできなかった．そこで，UML に対し GNU Profiler (gprof) および Unix Bench[3] を用いることで，効率的に最適化を行うことが出来る箇所について調査を行った．これらの結果をもとに，ベクタ化が性能向上に大きく影響する箇所を抽出する事を目的としている．これにより得られた箇所に対し，手動でベクタ化を行うことで，SIMD ユニットの有用性を調べることが出来る．本調査では，UNIX Bench を動作させている UML について，gprof によりプロファイリングを行った．これにより，UML 内のどのような関数で多くの処理時間が使われているかについて調査を行った．

この結果，プロセッサ時間を多く消費している UML 内の関数および，それぞれの消費時間のリストとして表 2 を得た．UNIX Bench のベンチマーク結果のスコアは，ベクタ化を行ったカーネルと行わないカーネルで，ほぼ同じ結果となった．

この結果から，最適化がもっとも効果を上げると考えられるのは，memcpy 関数と考えられる．memcpy 関数は頻繁に利用される関数であり，この関数の性能向上が，システム全体の性能向上に与える影響は大きい．

表 2 Linux カーネルのプロファイル結果

function	consumption time (sec)	share (%)
memcpy	12.96	8.72
os_arch_prctl	9.26	6.23
hard_handler	9.2	6.19
userspace	5.31	3.57
strncpy	3.86	2.97

6. 手動による SIMD を利用した最適化

先述の結果より，memcpy 関数の高速化はパフォーマンスへ大きく影響を与えることがわかる．memcpy は，ある領域上のデータを他の領域にコピーする関数である．この機能は SIMD ユニットの利用に適している．なぜなら，memcpy 内で行われる複数回のメモリコピーは，相互に依存していないためである．このため，SIMD ユニットの持つビット幅の広いデータ転送命令を使うことで，速度を向上することが出来ると考えられる．そこで，SIMD を利用する memcpy を実装し，通常の memcpy との比較を行った．SIMD を利用する memcpy は intel の SSE を利用する．SSE とは，intel の x86 アーキテクチャにおける SIMD 命令セットである．SIMD を利用する memcpy では，movdqu 命令を利用する．SIMD ユニットは独自のレジスタを持っており，専用の命令によって，これらに値をロード/ストアすることが出来る．値の転送元，転送先には，命令によって制限はあるものの，メモリおよびレジスタを指定することができる．movdqu 命令は一命令で 128 バイトのデータをロード/ストアする事ができる．これを利用する事で，通常の汎用レジスタを用いたデータコピーと比較し，少ない命令数でデータをコピーすることが可能となる．しかし，movdqu 命令は 128 バイト単位でのロード/ストアしか行うことが出来ないため，128 バイト未満のサイズのデータをコピーする事は出来ない．また，128 バイト単位のコピーごとに，残りのコピーサイズを調べるための条件分岐を行うと，条件分岐命令の数が増えてしまうため，効率が良くない．このため，SIMD を利用した memcpy では，512 バイト単位でのコピーのループを繰り返し，残りのコピーすべきサイズが 512 バイトよりも小さくなった場合は，汎用のストリング命令を用いたコピーを行う実装とした．この処理のフローチャートを図 5 に示す．

なお，標準ライブラリの memcpy では，x86 アーキテクチャのストリング命令を用いて全てのコピーを行っている．この SIMD を利用した memcpy と，カーネルの memcpy との比較を行った．それぞれの関数について，5,000,000 回の呼び出しについて計測を行った．

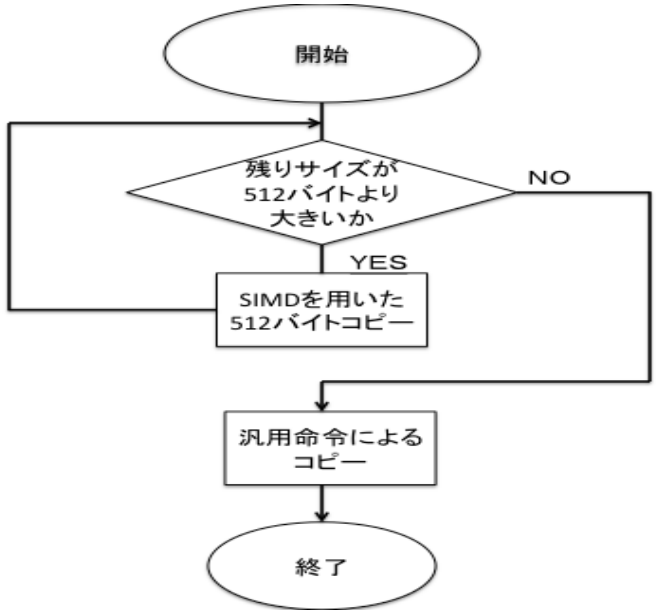

図 5 SIMD を用いた memcpy のフローチャート

コピーするデータサイズは，128 バイトから 2048 バイトまでを対象とした．結果を図 6 に示す．この結果より，データサイズが 512 バイト未満の場合にはカーネルの memcpy の方が SIMD を利用した memcpy よりも高速に動作している．これは，２つの memcpy が 512 バイト未満のサイズのメモリをコピーするルーチンは同じストリング命令を用いたものになるが，SIMD を用いた memcpy では残りのコピーサイズが 512 バイト以上であるか，を判定するためのルーチンが含まれるため，この処理の分，動作が遅くなっているものと考えられる．512 バイト以上のデータサイズでは，SIMD を用いた memcpy が高速に動作していることがわかる．また，グラフよりわかる特徴として，512 の倍数のサイズのデータをコピーする際，前後のサイズに比べ高速に動作している点が挙げられる．これは，512 バイトの倍数では SIMD を用いたコピーのみで処理が完了し，x86 のストリング命令によるコピーが行われないためだと考えられる．

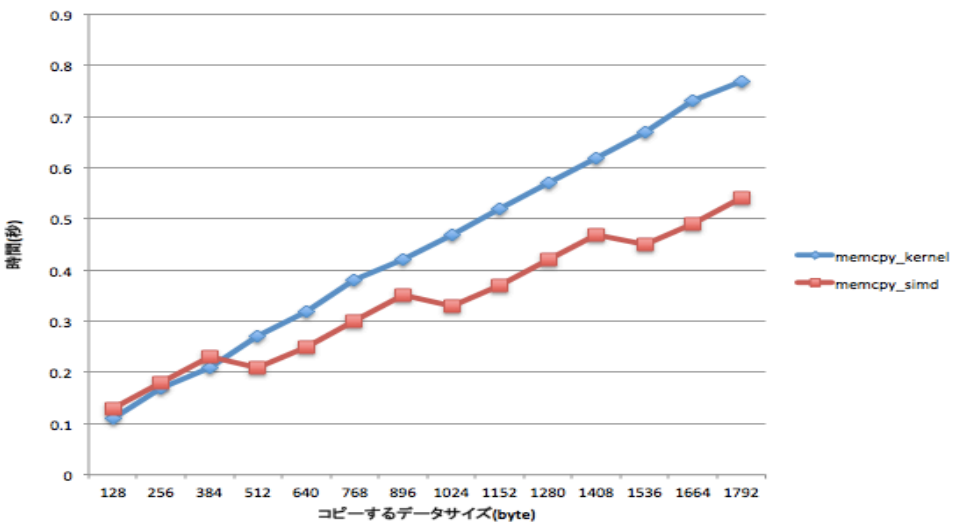

図 6 SIMD を用いた memcpy と，カーネルの memcpy の比較

表 3 SIMD を用いた memcpy を適用した UML

function	consumption time (sec)	rate (%)
os_arch_prctl	8.82	5.68
memcpy	8.75	5.64
hard_handler	8.72	5.62
userspace	5.27	3.40
strncpy	5.16	3.33

7. SIMD ユニットを利用する memcpy の，カーネルへの適用

先の SIMD を用いた memcpy 関数をカーネルに適用する事で，どれほどの高速化が出来るかについて実験を行った．UML の memcpy 関数を，さきほど述べた SIMD ユニットにより高速化を行った memcpy 関数と差し替え，それぞれの UNIX BENCH の結果について比較を行った．表 3 に，SIMD を用いた memcpy を適用した UML の実行時間のデータを示す．これは，先述の UML のプロファイリングの例と同様，UNIXBench を実行し，gprof を用いて収集した関数ごとの実行時間のデータを表す．

SIMD の適用により，memcpy 関数の実行時間が 12.96sec から 8.75sec へと削減され，約 33%高速化されている．また，実行時間全体に占める割合も，8.72%から 5.68%へ減少していることが確認できた．さらに，結果として UNIXBench に占める OS カーネルの実行時間は 155.11sec から 148.55sec へと減少し，5%の速度向上を得ることが出来た．表 4 に，通常の UML と，SIMD を利用した高速化を行った UML のそれぞれの実行結果を示す．

表 4 SIMD_memcpy によるカーネルの性能向上

	total time	memcpy time	memcpy rate(%)
normal UML	155.11s	12.96s	8.72%
SIMD_memcpy UML	148.55s	8.75s	5.68%

8. 考　　察

8.1 OS カーネルの特性に合わせた最適化

システムソフトウェアの特徴として，サイズが固定なデータを扱うことが多い点が挙げられる．これらを考慮した最適化を行うことで，SIMD 命令による高速化を効率よく行うことが出来ると考えられる．例として，固定サイズのコピーが挙げられる．先の項で比較を行った SIMD 命令を用いた memcpy は，全てのサイズのデータコピーに対応するため，一定サイズのコピーを複数回行う構造となっている．また，128 バイト未満のサイズのデータコピーに対応するため，各ループごとに残りサイズをチェックする処理が含まれている．これらの，ループに伴う条件分岐命令および残りのコピーサイズのチェックは，処理速度を低下させる．このように，全てのデータサイズに対応するための memcpy は，条件分岐命令を多く実行する必要があり，速度の低下に繋がっている．一方で，固定サイズのデータコピーについては，コピーするサイズごとに専用の関数を用意することで，条件分岐命令のないデータコピーを行うことができる．OS カーネル内には，memcpy に対し，固定サイズのコピーを指定するコードが多く含まれている．これらはコンパイル時にコピーするサイズが確定する．これらの固定サイズのコピー処理は，コンパイル時にすることが望ましい．例として，memcpy に 512 バイトのコピー要求を行うコードは，512 バイトのコピーを行う専用の関数 memcpy_512 を呼び出すように変更する．memcpy_512 の内部では，条件分岐をすることなく，4 回の SIMD 命令によるデータコピーを行う．これにより，高速にデータコピーを行うことが可能となる．OS カーネルにおけるデータ構造の特徴として，固定サイズのデータを多く扱う点が挙げられる．具体例として，ページテーブルのデータ構造は数多くの同サイズのデータを扱う．このような固定サイズのデータの処理は，専用のデータ処理命令を用意し，コンパイル前に静的に置換を行うことで高速化を行うことが出来ると考え

られる．

8.2 データサイズの局所性を利用した最適化

SIMD を用いた memcpy の特徴として，与えられるデータサイズおよび memcpy の実装により，SIMD 命令によりコピーするデータサイズと，ストリング命令によりコピーするサイズの比率が変化する点が挙げられる．SIMD を用いたコピーのほうが一度にコピーするサイズが大きいため，こちらのサイズを最大化することが望ましい．SIMD を用いた memcpy では，memcpy 関数内で一回のループで何バイトのデータを SIMD によりコピーするかで，高速にコピーすることのできるデータサイズが変化する．例として，一回のループでコピーするサイズを小さくした場合，小さなデータのコピーは高速となるが，大きなデータのコピーは効率が低下する．これは，小さなデータサイズのコピーに対しても SIMD 命令を用いることにより，高速化を図る事が出来るためである．一方，大きなサイズのデータをコピーする際は，ループの実行回数が増え，条件分岐命令を多く実行することとなり，効率が低下する．また，これとは逆に，一回のループでのコピーするサイズを大きくすることで，大きなサイズのデータを効率よくコピーすることが出来るようになる．このように，SIMD を用いた memcpy の特徴として，ループごとのメモリ転送量により，データサイズと実行速度間の特性が変化する点が挙げられる．一方で，一般的なシステムソフトウェアでは，それぞれの memcpy 関数のコピーするデータサイズは局所性を持つと考えられる．システムプログラムが memcpy を呼び出す際は，ある程度サイズの定まったメモリを操作する事が多い．このため，小さなサイズのデータをコピーする事が多い memcpy には，小さなサイズのデータコピーが高速な memcpy を呼び出すよう変更することで，データのコピーをより高速化することが出来ると考えられる．また，データ構造を 128 バイト単位のデータにより構築することで，SIMD によるメモリコピーを最大限活用するといった最適化手法も考えられる．このように，SIMD を用いた memcpy では一般の memcpy とは異なる特徴として，データサイズと速度の関係が挙げられる．これらを活用し，SIMD のデータコピーの性能を最大限発揮することで，より高いデータ転送性能を得ることが出来ると考えられる．

8.3 SIMD ユニットをシステムソフトウェアにおいて利用する際の制約

本研究では UML を用いて OS カーネルにおける SIMD ユニットの有用性について検討を行った．本章では，実際のシステムにおいて，カーネルが SIMD を利用できる場合の検討について述べる．ユーザレベルのプログラムが SIMD ユニットを利用しない場合，OS 等のシステムソフトウェアは SIMD ユニットを制限を受けること無く扱うことができる．ユーザレベルのソフトウェアが SIMD ユニットを利用しない状況として，システムのブート時の処理が挙げられる．ブート時は他のユーザレベルのソフトウェアが動作していないため，また，ブート時には OS 内部において，データ構造の初期化が行われる．これに伴い，データのコピーなどの処理が多く行われる．この処理に対し，考慮することなく SIMD 命令を適用する事が可能だと考えられる．また，ブート時以外にも，ユーザレベルのソフトウェアで SIMD 命令が利用されない環境も考えられる．また，近年，高性能化の進む組み込みアーキテクチャにおける利用も考えられる．例として，ARM アーキテクチャにおける SIMD ユニットがある．また，ARM アーキテクチャのサポートする SIMD ユニットである NEON は，現在チップごとに搭載がまちまちであり，一般的なアプリケーションは NEON を利用しない実装となっている．このため，このような環境においては，SIMD ユニットを OS が自由に活用することができる．また，ユーザレベルにおいて SIMD ユニットが利用されている場合，OS カーネルはコンテキストスイッチ時に SIMD の扱うレジスタを待避させることで，ユーザレベルのプログラムと OS カーネルの SIMD ユニットの共用が可能となる．

9. おわりに

本稿では，SIMD ユニットを OS カーネルで活用する有用性について述べた．SIMD ユニットがプロセッサの一部として組み込まれ，多くのアプリケーションの性能向上を実現しているにもかかわらず，OS カーネル内では有効活用されていない．データマイニングなどの分野と同様に，OS カーネルも多量のデータ処理を行う．このため，OS カーネル内部でも SIMD ユニットを活用したデータ処理が有用であると考えられる．本稿では，計測のための環境として UML(User Mode Linux) を利用し，GCC による自動ベクタ化を OS カーネルに対した．これにより，OS カーネル内のうち，自動ベクタ化できる箇所について調査を行った．さらに，OS カーネルの動作のプロファイリングを行い，実際に一部の処理について SIMD を用いて最適化を行うことで，処理を高速化できる可能性があることを示した．今後は，データのコピー以外に，OS カーネル内で SIMD ユニットを活用できる箇所を探し，最適化を行う．例として，ファイルシステムでのハッシュ値の計算や，暗号化などが挙げられる．また，近年の intel プロセッサに搭載されている最新の SIMD ユニットがサポートする AVX 命令セットは，256 バイトのデータを 1 命令で処理することが出来る．また，近年の SIMD はデータのビット幅が増えただけでなく，暗号化の支援など，特殊な命令が加わっている．これらの調査を行い，OS カーネル内部で利用できるものについて調査を行う．

Vol.2012-OS-120 No.1
2012/2/28

参 考 文 献

1) Takashi Nakamura, Satoshi Miki, Shuichi Oikawa, "Automatic Vectorization by Runtime Binary Translation," In *Proceedings of 2011 Second International Conference on Networking and Computing*,pp.87-94, 2011
2) The User-mode Linux Kernel Home Page http://user-mode-linux.sourceforge.net/
3) byte-unixbench Unix benchmark Suite http://code.google.com/p/byte-unixbench/
4) Intel 64 and IA-32 Architectures Software Developer's Manuals http://www.intel.com/products/processor/manuals/
5) Intel Applications Tuning for Streaming SIMD Extensions http://download.intel.com/technology/itj/Q21999/ PDF/apps_simd.pdf