Vol.2012-OS-123 No.10
Vol.2012-EMB-27 No.10
2012/12/5

アプリケーション - カーネル間共有メモリを用いたユーザ関数カーネル内実行機構の実現

安井 裕亮[1]　齋藤 彰一[1]　津邑 公暁[1]　毛利 公一[2]　松尾 啓志[1]

概要：システムコールは，CPU の割り込みを用いて実装されてきた．しかし，システムコールによる割り込みがアプリケーションやカーネルの実行を妨げることが指摘されている．この問題に対して，システムコール発行時に割り込みを必要としない手法として FlexSC が提案されている．しかしこの手法には共有メモリへのアクセスコストの問題や，同じデータにアクセスするシステムコールの扱いに関する問題がある．そこで本研究では，FlexSC の持つ問題点を解決する手法として，ユーザ関数の非同期カーネル内実行機構である Sakura Call を提案する．評価においてこの Sakura Call が FlexSC よりも大きな実行時間の削減を達成していることを示した．

A Mechanism Enabling In-Kernel Execution of a User Function with a Shared Memory between Application and Kernel

Yusuke Yasui[1]　Shoichi Saito[1]　Tomoaki Tsumura[1]　Koichi Mouri[2]　Hiroshi Matsuo[1]

Abstract: System calls have been implemented with an interruption mechanism provided by a CPU. However, it is claimed that an interruption cased by a system call impacts the performance of applications and a kernel. FlexSC which requires no interruptions at invoking a system call is proposed to solve the problem. Nevertheless there are some problems such as cost of accessing to a shared memory and a method of handling system calls which share the same data with each other. To fill up deficiencies of FlexSC, we propose a new mechanism, called Sakura Call, which enables asynchronous in-kernel execution of user functions. We show that Sakura Call achieves more reduction of an execution time than FlexSC by an evaluation.

1. はじめに

システムコールは，CPU の割り込みを用いて実装されてきた．しかし，割り込みによるシステムコールはアプリケーションやカーネルの実行を妨げることが指摘されている．割り込みがもたらすコストには直接的なものと間接的なものがある．直接的なコストは割り込みに要する時間である．割り込みには CPU の動作モード切り替えやパイプラインフラッシュ，実行領域の遷移などの時間を要する処理が伴う．間接的なコストは，実行領域の遷移によるメモリキャッシュや TLB の汚染や，パイプラインフラッシュによる実行速度の低下である．

これらの問題を解決する手法として，システムコール発行時に割り込みを必要としないシステムコール発行方式である FlexSC[1], [2] が提案されている．FlexSC は共有メモリとカーネルスレッドを用いることで，割り込みを用いないシステムコール発行を実現している．しかしこの手法には共有メモリへのアクセスコストの問題と，同じデータにアクセスするシステムコールの扱いに関する問題がある．

そこで本稿では，これらの問題点を解決した Sakura Call を提案する．Sakura Call は FlexSC を拡張し，システムコールを含むユーザ関数のカーネル内非同期実行を実現したユーザ関数発行型手法である．Sakura Call ではシステムコール単位ではなくユーザ関数単位でのカーネルへの処理要求が可能である．このため，FlexSC が持つ共有メモリアクセスコストの問題と，同じデータにアクセスするシ

[1] 名古屋工業大学
Nagoya Institute of Technology
[2] 立命館大学
Ritsumeikan University

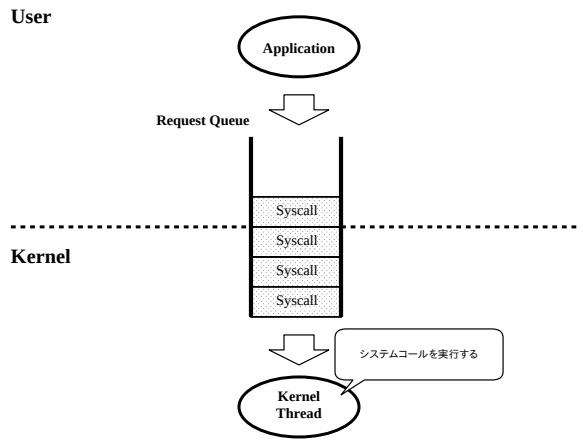

図 1 FlexSC の概念図

ステムコールの扱いに関する問題を解決できる．

以降，2 章においてシステムコールに関する既存手法について述べ，3 章において提案手法について述べる．4 章において提案手法の実装について述べ，5 章において評価について述べる．最後に 6 章においてまとめる．

2. 既存手法

現在までに提案されている，システムコールに伴う割り込みを削減するための手法について述べる．また，提案手法の基盤となった研究である FlexSC について述べる．

2.1 システムコール一括実行手法

システムコールによる割り込みを削減するためのアプローチとして，複数のシステムコールを 1 つにまとめることに着目した手法が提案されている．Cassyopia[3] はコード移動やループ展開，インライン展開といったコンパイラ最適化の手法を用いて複数のシステムコールを一箇所にまとめ，それら一連のシステムコールを一つのマルチコールと呼ばれる大きなシステムコールにまとめる手法である．一方で，Cosy[4] や Zadok らの手法 [5] では，Cosy-GCC と呼ばれる独自のコンパイラによって，指定したコードブロックをカーネル内実行可能な形に変換することで複数のシステムコールを 1 つにまとめている．

これらの手法によってシステムコールをまとめることで，割り込みの回数を削減することができる．しかしこれらの手法では割り込みを完全に排除することはできない．

2.2 FlexSC

FlexSC は割り込みを必要としないシステムコール発行方式である．概念図を**図 1** に示す．図 1 に示すように，FlexSC はシステムコール実行専用のカーネルスレッドとシステムコール要求発行用のリクエストキューから構成される．リクエストキューはアプリケーションとカーネルスレッドの間で共有される共有メモリである．FlexSC を用いた場合，アプリケーションはシステムコールを発行する際に，割り込みを発生させる代わりにリクエストキューに対してシステムコール実行要求を発行する．発行された要求はカーネルスレッドによって取得され，システムコールが呼び出される．

FlexSC ではシステムコールによる割り込みを排除できる点が従来のシステムコールや割り込みを必要とする他の手法と異なる．また，カーネルスレッドをアプリケーション側のスレッドが動作するコアとは別のコア上で動作させることができるため，メモリキャッシュや TLB を効率良く使用することができる．

割り込みを用いないシステムコール発行方式は Rhoden らの手法 [6] でも採用されている．また，SplitX[7] では割り込みを用いずにハイパーコールを発行する方式として，FlexSC と同様な手法を VM に適用している．さらに MegaPipe[8] では，ネットワーク I/O のスループットを向上させるために，FlexSC が用いられている．

2.3 FlexSC の問題点

FlexSC には主に二つの問題点がある．1 つ目の問題点は，発行するシステムコールの数に比例した回数のリクエストキューへのアクセスが要求されることである．FlexSC では 1 つのシステムコールを処理するために，アプリケーション側のスレッドにおいてシステムコール実行要求の発行と結果の取得のために 2 回，カーネルスレッドにおいて要求の取得と結果の書き込みのために同様に 2 回のリクエストキューへのアクセスが必要となる．つまり，FlexSC では割り込みのコストの代わりにリクエストキューへのアクセスコストが発生する．

2 つ目の問題点は，同じデータにアクセスするシステムコールが同じコア上で実行される保障がないことである．同じデータにアクセスするシステムコールとは，例えば，read システムコールで読み出したデータを write システムコールで書き込む処理における read システムコールと write システムコールである．これら 2 つのシステムコールは，メモリキャッシュを考慮すると同じコア上で実行されることが望ましい．同じデータにアクセスするシステムコールが同じコア上で実行されることの重要性は Pesterev らの論文 [9] でも述べられている．しかし，FlexSC ではこのような関連性のあるシステムコールのコアへの割り当てについては考慮されていない．

3. 提案手法

本研究の提案手法である Sakura Call について述べる．

3.1 Sakura Call

Sakura Call は，FlexSC を拡張して，任意のユーザ関数の実行をカーネルスレッドに要求できるようにした手法で

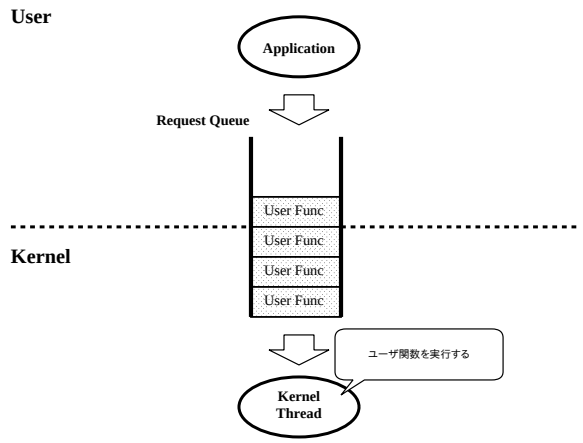

図 2 Sakura Call の概念図

```
/* HTTPリクエストハンドラ */
request_handler(client_fd)
{
  /* クライアントからリクエストを受信 */
  recv(client_fd, req);
  /* リクエストからファイルパスを取得 */
  filepath = parse_request(req);
  /* ファイル読み出し */
  fd = open(filepath);
  read(fd, buf);
  close(fd);
  /* ファイルの中身を送信 */
  send(client_fd, buf);

  close(client_fd);
}

main()
{
  server_fd = socket();
  listen(server_fd);

  while (1) {
    client_fd = accept(server_fd);
    /* リクエストキューに要求を発行する */
    sakura_call(request_handler, client_fd);
  }

  close(server_fd);
}
```

図 3 Sakura Call を用いた Web サーバの例

ある．概念図を**図 2** に示す．Sakura Call を用いたアプリケーションは，リクエストキューにユーザ関数実行要求を発行する．発行された要求はカーネルスレッドによって取得され，ユーザ関数が呼び出される．

Sakura Call を用いて実装した Web サーバの例を**図 3** に示す．`main` 関数の `while` ループ内で呼び出している `sakura_call` 関数はリクエストキューにユーザ関数実行要求を発行する関数である．この Web サーバでは，この `sakura_call` 関数によって，HTTP リクエストを処理する関数である `request_handler` 関数の実行要求を発行している．すなわち，`request_handler` 関数はカーネルスレッドによって実行される．この Web サーバの例からわかる通り，Sakura Call はスレッドライブラリの代わりとして利用することもできる．

3.2 Sakura Call の利点

Sakura Call ではシステムコール単位ではなくユーザ関数単位で処理を要求するため，FlexSC の 1 つ目の問題点のようにシステムコール毎にリクエストキューへのアクセスは必要ない．このため，Sakura Call では FlexSC に比べてリクエストキューへのアクセス回数を削減できる．例えば図 3 で示した Web サーバを FlexSC を用いて実現した場合，`request_handler` 関数には 6 つのシステムコールが含まれるため，HTTP リクエスト数 ×6 回のリクエストへのアクセスが必要となる．対して Sakura Call では，必要なリクエストキューへのアクセス回数は HTTP リクエスト数分だけである．

もう 1 つの利点は同じデータにアクセスするシステムコールが同じコア上で実行されることが保障されることである．Sakura Call では，カーネルスレッドはシステムコール単位ではなく，ユーザ関数単位で処理を行う．このため，同じユーザ関数内に含まれるシステムコールは，同じコア上で実行されることが保障される．したがって，同じデータにアクセスするシステムコールを同じユーザ関数内に配置すれば，これらを確実に同じコア上で実行することができる． 2.3 節でも述べた通り，同じデータにアクセスするシステムコールが同じコア上で実行されることは，メモリキャッシュを効率的に利用する上で重要なことである．例えば図 3 で示した Web サーバにおいて，`request_handler` 関数中のファイルを読み出す read システムコールと，そこで読み出したデータをクライアントに送信する send システムコールは，同じコア上で実行されればキャッシュを利用して高速にデータをやりとりすることができるが，異なるコア上で実行されれば，大きなキャッシュミスペナルティを負うことになる．

4. 実装

Sakura Call の実装と，比較実験を行うために独自実装した FlexSC の実装について述べる．なお，本章と次の 5 章では，FlexSC という言葉を独自実装した FlexSC を指す言葉として用いることとする．実装は，Linux カーネル 2.6.38.4 上に行った．本章では，カーネルスレッドとリクエストキューの生成方法と，カーネルスレッドの実装方法と，カーネルスレッドのスケジューリング方式について述

べる．

4.1 カーネルスレッドとリクエストキューの生成

カーネルスレッドとリクエストキューの生成は，Sakura Call 及び FlexSC を利用するアプリケーションが最初に呼び出す初期化関数の中で生成される．この初期化関数は Sakura Call 及び FlexSC が提供するライブラリに含まれるものである．

初期化関数の中では，まずカーネルスレッドを生成するためのシステムコールが呼び出される．このシステムコール内でのカーネルスレッドの生成は kernel_thread 関数によって行われる．kernel_thread 関数は do_fork 関数のラッパ関数であり，do_fork 関数に与えるフラグを指定することができる．Sakura Call のカーネルスレッドの生成時には kernel_thread 関数に対して，メモリ空間を共有するための CLONE_VM，ファイルシステムに関わる情報を共有するための CLONE_FS と CLONE_FILES，同じスレッドグループに所属させるための CLONE_THREAD と CLONE_SIGHAND をフラグとして与えている．これらのフラグを与えることで，システムコールを呼び出したスレッドと生成されたカーネルスレッドはメモリ空間やファイルシステムに関わる情報を共有することができる．これにより，リクエストキューの実現やカーネルスレッドによるファイル操作の代行が可能となる．

リクエストキューの生成は mmap システムコールによって行われる．mmap システムコールによって確保されたメモリ領域はカーネルスレッドからも特別な操作なしにアクセスできる．これはアプリケーションとカーネルスレッドはメモリ空間を共有しているためである．

4.2 カーネルスレッドの実装

カーネルスレッドの疑似コードを**図 4** に示す．図 4 に示した通り，カーネルスレッドは 3 つの処理フェーズを繰り返している．以降では，この 3 つのフェーズうちの 1 つである要求実行フェーズの実装と，要求の取得から実行までの処理コストを削減するために行った最適化について述べる．なお，要求実行フェーズは FlexSC と Sakura Call で動作が異なるので，それぞれ個別に説明する．

4.2.1 FlexSC における要求実行

FlexSC のカーネルスレッドは要求実行フェーズでシステムコールサービスルーチンを呼び出す．システムコールサービスルーチンの呼び出しは，sys_call_table 配列に格納されているシステムコールサービスルーチンの関数ポインタを利用する．sys_call_table 配列はすべてのシステムコールサービスルーチンの関数ポインタを格納している配列であり，システムコール番号をインデックスとして参照することで，システムコール番号に対応するシステムコールサービスルーチンの関数ポインタを得ることができる．

```
kthread_main()
{
  while (1) {
    /* リクエストキューからの要求のフェッチ */
    request = fetch_request(queue);
    /* 要求の実行 */
    ret = execute(request);
    /* リクエストキューへの結果の書き戻し */
    return_value(queue, ret)
  }
}
```

図 4 カーネルスレッドの疑似コード

```
#define sakura_open(a0,a1,a2)   \
((int(*)(char*,int,int))0xc10b6688)(a0,a1,a2)
#define sakura_close(a0)    \
((int(*)(int))0xc10b651e)(a0)
#define sakura_read(a0,a1,a2)   \
((int(*)(int,void*,int))0xc10b84d2)(a0,a1,a2)
#define sakura_write(fd, buf, count)  \
((int(*)(int,void*,int))0xc10b832b)(a0,a1,a2)
```

図 5 システムコールサービスルーチンを直接呼び出すためのマクロ

システムコールサービスルーチンを呼び出す方法として，switch 文による分岐を利用する方法も考えられるが，この方法ではシステムコール番号の大きさに比例して値の一致比較回数が増加するので，sys_call_table 配列を用いた方法を採用した．

4.2.2 Sakura Call における要求実行

Sakura Call のカーネルスレッドは要求実行フェーズで，アプリケーションから指定された関数を呼び出す．CLONE_VM の指定により，カーネルスレッドとアプリケーションはメモリ空間を共有しているため，カーネルスレッドはアプリケーションから与えられたユーザ関数の関数ポインタを通常の関数呼び出し手順によって呼び出すことができる．このため，ユーザ関数の呼び出しに対しては特別な機構を設ける必要はない．しかし，ユーザ関数を意図した通りに実行するためには，いくつか考慮すべき点がある．これはユーザ関数内にシステムコール呼び出しが含まれる場合である．

x86 系の CPU では，システムコールの呼び出しは通常 sysenter 命令によって行われる．しかし，カーネルスレッドのコンテキストでは sysenter 命令によってシステムコールを呼び出すことはできない．カーネルスレッドからシステムコールを呼び出すには，システムコールサービスルーチンを直接呼び出す必要がある．このため，ユーザ関数をカーネルスレッドが実行可能な形式に変換するには，システムコールを呼び出すコードをシステムコールサービ

```
sysno = queue->head.sysno;
memcpy(args,queue->head.args,sizeof(int) * 2);
queue->ptr++;
ret = (sys_call_table[sysno])(args[0],args[1]);
```

図 6 リクエストキューからの要求の取得と実行（最適化前）

スルーチンを直接呼び出すコードに変換する必要がある．この変換を容易に実現するために，Sakura Call では**図 5** に示すようなマクロを提供している．ユーザ関数中の read や write といったシステムコールラッパー関数を Sakura Call が提供するこれらのマクロに置き換えることによって，ユーザ関数をカーネルスレッドが実行可能な形式に変換することができる．

ユーザ関数中のシステムコール呼び出しに関して，もう一つ考慮すべきことがある．read や write のように，ポインタを受け取るシステムコールは，システムコールサービスルーチン中で受け取ったポインタがユーザ空間に属するものか否かのチェックを行っている．このため，ユーザ関数の局所変数として定義された領域のポインタをシステムコールに渡している場合，システムコールの実行が正しく行われない．この問題を回避するため，システムコールの引数として与えられたポインタのチェックを行う `access_ok` マクロに，Sakura Call のカーネルスレッドのコンテキストではチェックをスキップする処理を加えた．Sakura Call のカーネルスレッドのコンテキストであるか否かは，`task_struct` 構造体に追加したメンバ変数をフラグとして判断する．

4.2.3 要求の取得から実行までの最適化

リクエストキューから要求を取得し，実行するまで処理の最適化について述べる．FlexSC のカーネルスレッドが 2 引数のシステムコールの要求を取得し，実行する場合を例に説明する．該当個所の最適化前の疑似コードを**図 6** に，最適化後のコードを**図 7** に示す．図 6 に示した実装は一般的な実装であるが，この実装ではシステムコールに与える引数をリクエストキューから一時バッファへとコピーし，その後一時バッファからシステムコールサービスルーチンの引数領域へとコピーする必要がある．すなわち合計 2 回のメモリコピーが必要となる．一方で図 7 に示した実装では，インラインアセンブラを用いて，システムコールに与える引数をリクエストキューからのシステムコールサービスルーチンの引数領域へと直接コピーする動作を実現している．したがって必要なメモリコピー回数は 1 回となる．

この最適化は，特に FlexSC で引数の数が多いシステムコールを処理する場合に有効である．

4.3 カーネルスレッドのスケジューリング方式

カーネルスレッドのスケジューリング方式として，起動数固定方式と起動数変動方式の 2 つの方式を実装した．

```
sysno = queue->head.sysno;
asm ("subl␣$3,%%esp\n\t"
     "movl␣0x8(%0),%%eax\n\t"
     "movl␣%%eax,0x8(%%esp)\n\t"
     "movl␣0x4(%0),%%eax\n\t"
     "movl␣%%eax,0x4(%%esp)\n\t"
     "movl␣(%0),%%eax\n\t"
     "movl␣%%eax,(%%esp)\n\t"
     :
     : "r" (queue->head.args)
     : "%eax");
queue->ptr++;
asm ("call␣*%1\n\t"
     "addl␣$3,%%esp"
     : "=a" (ret)
     : "r" (sys_call_table[sysno])
     : "%edx","%ecx");
```

図 7 リクエストキューからの要求の取得と実行（最適化後）

起動数固定方式は各コア毎に事前に決めた数のカーネルスレッドを予め生成して起動する方式である．各コアのカーネルスレッドのコンテキストスイッチのタイミングはカーネルのスケジューラによって決定される．この方式では各カーネルスレッド対して 1 つずつリクエストキューが割り当てられる．

起動数変動方式は各コア毎にリクエストキューのエントリ数分のカーネルスレッドを生成し，そのうち 1 つのみを最初に起動しておく方式である．各カーネルスレッドは I/O 待ちが発生した際に，待機しているカーネルスレッドの中から自身の代わりとなるものを 1 つ起動する．この方式では各コア毎に 1 つずつリクエストキューが割り当てられる．起動数変動方式は既存手法の FlexSC が採用しているスケジューリング方式である．

I/O 待ちが多いユーザ関数を実行する場合は，起動数固定方式で 2 つ以上のカーネルスレッドを生成するか，起動数変動方式を用いることが有効であると考えられる．

5. 評価

従来の割り込みによるシステムコール及び FlexSC に対する Sakura Call の有効性を示すための評価を行った．評価では，割り込みの削減による実行時間削減を検証するための基本的な実験を行った．なお，評価に用いた FlexSC は 4 章で示した独自実装による FlexSC であり， 4 章で述べた通り，本章でも FlexSC という言葉を独自実装した FlexSC を指す言葉として用いる．

従来のシステムコールと他の 2 手法との比較実験を行うために，Sakura Call のカーネルスレッドをユーザスレッドに置き換えた手法を作成した．以降，この手法を従来手法と呼ぶ．さらに，以降の説明の中で用いる言葉を以下の

表 1 実験環境

CPU	Intel(R) Core(TM) i5 CPU 760 @ 2.80GHz
Memory	8GB
Kernel	Linux i686 2.6.38.4
Storage	SSD SATA2 3.0Gb/s 64GB

ように定義する．

ワーカースレッド 従来手法におけるユーザ関数実行用のユーザスレッド及び，FlexSC，Sakura Call におけるカーネルスレッド

マスタースレッド ワーカースレッドに対して要求を発行するユーザスレッド

5.1 実験環境

実験環境を**表 1** に示す．実験は 4 コア環境で行い，いずれの実験においても，3 コアにワーカースレッドを，残りの 1 コアにマスタースレッドを配置した．ワーカースレッドのスケジューリング方式としては，起動数固定方式を採用し，ワーカースレッドの生成数は 1 とした．これは今回の実験では I/O 待ちが発生しないため，1 つ以上のワーカースレッドの起動が意味を為さないためである．

5.2 公平性を検証するための評価

評価を行う前に，カーネルスレッドがユーザスレッドよりも高い優先度で実行されていないことを検証するための実験を行った．この検証は，後の評価における高速化の効果が，スレッドの実行優先度の差に因らないこと保障するために必要である．実験では，ループを実行するだけのユーザ関数を，従来手法と Sakura Call で実行することで行い，発行したすべての要求が完了するまでの時間を計測した．なお，リクエストキューに投入する要求の総数は 65536 とした．この実験において実行されるユーザ関数はシステムコールを含まないため，従来手法と Sakura Call で実行時間に差が生じないはずである．

実験結果をに**図 8** 示す．縦軸は実行時間をミリ秒で，横軸はユーザ関数内のループ回数を示している．図 8 からわかる通り，従来手法と Sakura Call の両者で実行時間に差がないことが確認できる．この結果から，カーネルスレッドはユーザスレッドよりも高い優先度で実行されていないことが示されている．

5.3 割り込みの削減による実行時間削減の評価

本評価では，システムコールを複数回呼び出すタスクを並列処理するプログラムを従来手法，FlexSC，Sakura Call の 3 つの手法を用いて作成し，実行時間の比較を行った．呼び出すシステムコールとしては，内部でループを実行するだけのシステムコールを使用した．従来手法及び Sakura Call では，タスクに対応するユーザ関数をプログラム内に定義し，マスタースレッドがそのユーザ関数の実行要求をリクエストキューに発行する．一方，FlexSC を用いたプログラムでは，マスタースレッドが各タスクから 1 つずつシステムコールを抽出してリクエストキューに発行する．なお，処理するタスク数は 65536 とした．

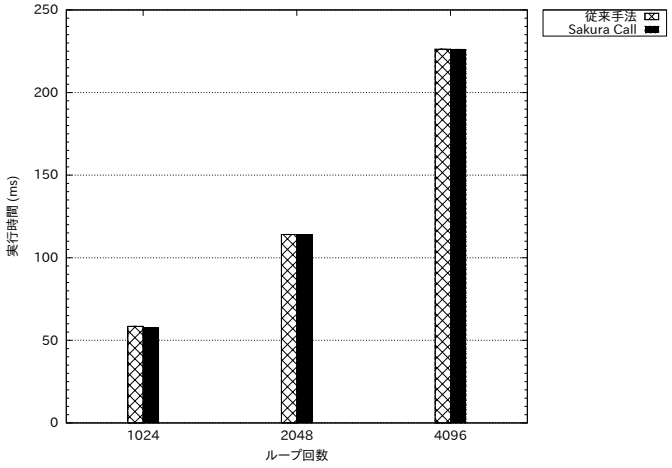

図 8 ループを実行するだけのユーザ関数による評価

まず，システムコール内のループ回数を 128 回に固定して，1 タスク内のシステムコール呼び出し回数を変化させた場合の結果を**図 9** に示す．縦軸が実行時間，横軸が 1 タスク内のシステムコール呼び出し回数である．システムコール呼び出し回数が増加するにしたがって，従来手法と他の 2 手法の差が大きくなっていることがわかる．これは，システムコール呼び出し回数が増加するにつれて，割り込みの削減により減少する時間も増加するためであると考えられる．Sakura Call が FlexSC に比べて大きな実行時間の削減を達成している理由は，リクエストキューへのアクセス回数が削減できているためであると考えられる．システムコール呼び出し回数が 1 のときに FlexSC での実行時間が Sakura Call での実行時間を下回っている理由は，Sakura Call では関数呼び出しとシステムコール呼び出しが必要となるが，FlexSC ではシステムコール呼び出しのみが必要となるためであると考えられる．

次に，1 タスク内のシステムコール呼び出し回数を 16 回に固定して，システムコール内のループ回数を変化させた場合の結果を**図 10** に示す．この場合は，従来手法と他の 2 手法の差は回数に関わらず一定であることがわかる．これは発行されたシステムコールの回数がすべての場合において同じであるためである．システムコール内のループ回数が 64 の場合に FlexSC の実行時間が大幅に増加している理由は，システムコールの処理時間が短かすぎて，マスタースレッドが要求を発行する速度に対して，システムコールの処理が完了する速度が大きくなったため，結果返却用のキューが飽和したことが原因であると考えられる．

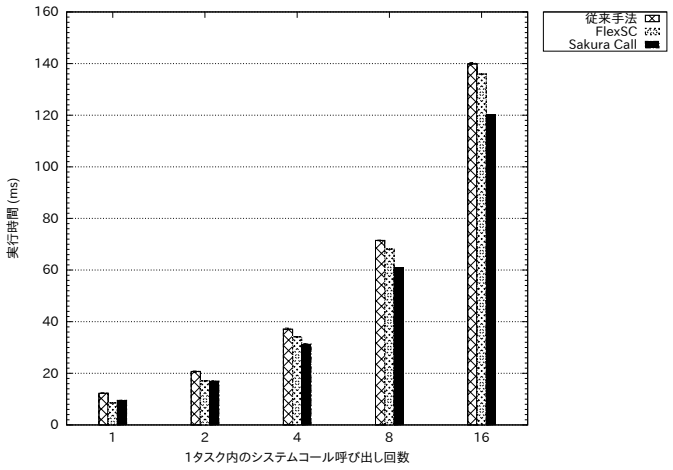

図 9 システムコール内のループ回数を 128 回に固定した場合

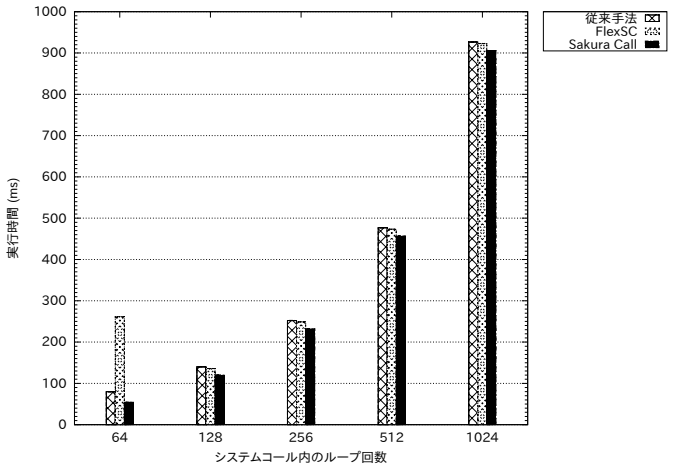

図 10 1 タスク内のシステムコール呼び出し回数を 16 回に固定した場合

5.4 今後の課題

Sakura Call の課題について述べる．1 つ目はアプリケーションの移植の困難性の問題である．FlexSC は提供するライブラリによって比較的容易にアプリケーションの移植が実現できる．しかし Sakura Call ではプログラマが大幅に書き換える必要がある．

二つ目はセキュリティの問題である．Sakura Call ではカーネルスレッドが任意のユーザ関数を実行するため，ユーザ関数を安全に実行できる枠組みが必要となる．この問題に関しては，現在未実装であるが，Cosy などの手法でも用いられているような x86 のセグメント機構による対策が可能であると考えている．

6. まとめ

ユーザ関数のカーネル内実行機構である Sakura Call を提案した．Sakura Call は割り込みを必要としないシステムコール発行方式である FlexSC を拡張し，アプリケーションがカーネルスレッドに任意のユーザ関数の実行要求を非同期に発行することを可能とした手法である．評価において Sakura Call が FlexSC に比べて大きな実行時間の削減を得られることを示した．

今後の課題としては，アプリケーション移植の困難性の問題の解決や，セキュリティの問題の解決などが挙げられる．また，キャッシュへの影響を検証するための評価や，I/O が発生するような実際のアプリケーションを想定した評価を行う必要がある．

参考文献

[1] Soares, L. and Stumm, M.: FlexSC: Flexible system call scheduling with exception-less system calls, *In Proc. of the 9th USENIX conference on Operating systems design and implementation* (2010).
[2] Soares, L. and Stumm, M.: Exception-less system calls for event-driven servers, *In Proceedings of the 2011 USENIX conference on USENIX annual technical conference* (2011).
[3] Rajagopalan, M., Debray, S., Hiltunen, M. and Schlichting, R.: Cassyopia: compiler assisted system optimization, *In Proceedings of the 9th conference on Hot Topics in Operating Systems* (2003).
[4] Purohit, A., Wright, C., Spadavecchia, J., Zadok, E. et al.: Cosy: Develop in user-land, run in kernel-mode, *In Proceedings of the 9th conference on Hot Topics in Operating Systems* (2003).
[5] Zadok, E., Callanan, S., Rai, A., Sivathanu, G. and Traeger, A.: Efficient and safe execution of user-level code in the kernel, *In Proceedings of the 19th IEEE International conference on Parallel and Distributed Processing Symposium* (2005).
[6] Rhoden, B., Klues, K., Zhu, D. and Brewer, E.: Improving per-node efficiency in the datacenter with new OS abstractions, *In Proceedings of the 2nd ACM Symposium on Cloud Computing* (2011).
[7] Landau, A., Ben-Yehuda, M. and Gordon, A.: SplitX: Split guest/hypervisor execution on multi-core, *In Proceedings of the 3rd conference on I/O virtualization* (2011).
[8] Han, S., Marshall, S., Chun, B. and Ratnasamy, S.: MegaPipe: A New Programming Interface for Scalable Network I/O, *In Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation* (2012).
[9] Pesterev, A., Strauss, J., Zeldovich, N. and Morris, R.: Improving network connection locality on multicore systems, *In Proceedings of the 7th ACM european conference on Computer Systems* (2012).