Vol.2021-OS-151 No.2
2021/3/1

タスク同期方式 FTC のための コールグラフ活用による非決定性タスクスイッチの抽出方法

河合英宏[1]　小川雅昭[1]

概要：FTC（Fault Tolerant Computer）方式の一つであるタスク同期方式は，タスクスイッチを行うプログラム上の箇所と順序をノード間で合意形成することで，各ノード上で動作するアプリケーションの出力について決定性保証することを一つの特徴とする．これを Linux 上で実現するには，システムコールの処理から非決定性のタスクスイッチを抽出し，決定性の振る舞いをするよう同期処理を追加する必要がある．以前はこの抽出作業を人手で実施したが，今回はコールグラフを活用した非決定性タスクスイッチの網羅的抽出手法を考案し，効率化を図った．これを実際に FTC 用 Linux カーネルのバージョンアップの際に適用し，抽出作業に要する期間を従来比 7 割減とすることができた．

キーワード：FTC，コールグラフ, Linux

Call Graph-based Method of Extracting Nondeterminism for Task Sync-based Fault Tolerant Computer

HIDEHIRO KAWAI[†1]　MASAAKI OGAWA[†1]

Keywords: FTC, callgraph, Linux

1. はじめに

FTC（Fault Tolerant Computer）の実現方式の一つにタスク同期方式[1][2]がある．タスク同期方式は SMR（State Machine Replication）の一種であり，ノード間の同期通信によりタスクスイッチのシリアライズと，システムコールの結果一致化等を行うことでこれを実現する．各ノードに複製されたアプリケーションは, それぞれ同じ入力を受信し，決定的にそれを処理し，同じ状態に遷移しつつ，同じ出力を同じタイミングで行う．これにより，一部のノードで障害が発生しても，残りの正常ノードでシームレスに処理を継続できる．

タスク同期方式の FTC を汎用 OS である Linux 上で実現するにあたり，様々な非決定性の要素を同期対象のタスクから排除する必要がある．その代表的な要素は，同期対象外のタスクと共有するリソースに起因する非決定性のタスクスイッチである．例えば，その共有リソースにアクセスするにあたり mutex ロックをとる場合，ロックを獲得できれば処理を続行し，できなければタスクスイッチしてロックが解放されるまで待つ. つまりタスクスイッチの有無は，非同期タスクが当該ロックを所持しているか否かによって変化する．このような処理があると，タスクスイッチの有無がノード間で割れ，同期状態を維持できなくなる．

当初，こうした非決定性の要素の抽出は，膨大なカーネルソースコードを人手により精査して実施した．しかし，カーネルのバージョンアップの都度，再精査を実施したのでは多大な工数を要するだけでなく，見落としによる品質低下も懸念される．

そこで本研究では，コールグラフを活用した網羅的ソースコード精査方法を考案し，これにより同期失敗の原因となる非決定性タスクスイッチを効率的に抽出する．

実際にカーネルバージョンアップの際に適用した結果，従来と比べ，精査に要する工数を約 7 割削減することができた．また，テスト段階で発見した“見落とし”は 1 件のみであり, 品質保証の観点においても有用であると示せた．

2. タスク同期方式 FTC

本章では本研究が前提とするタスク同期方式 FTC について，関連する内容を簡単に説明しておく．

2.1 システムアーキテクチャ

図 1 に本研究が対象とする FTC システムのアーキテクチャを示す．各ノード上では同一の Linux とサーバアプリケーションが動作し，四重系を構成する．サーバアプリケーションは複数のタスク（プロセス）から構成さる．FTC 用に改造を加えた Linux カーネルは，これらのタスクについて決定性の振る舞いをするよう適宜，同期ネットワークを通してノード間で同期をとりながら実行する．

クライアントから見ると，図 1 の破線で囲った部分全体が一つのサーバであるかのように見える．つまりクライアントは多数決装置に対してリクエストメッセージを送り，多数決装置はこれを複製して 4 台のノードに転送する．各ノードのサーバアプリケーションはこれを決定的に処理し，同一のレスポンスメッセージを同一タイミングで多数決装

1 (株)日立製作所
Hitachi Ltd.

置に返す．多数決装置は各ノードから受信したレスポンスメッセージに対して多数決を行い，確からしいメッセージをクライアントに転送する．

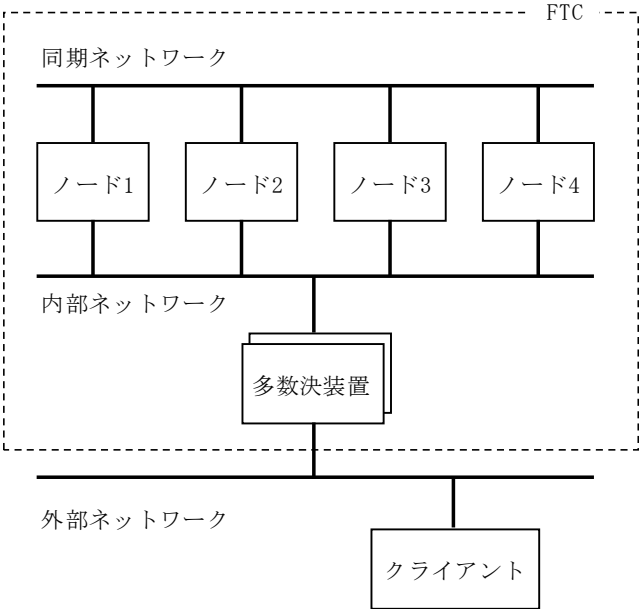

図 1 FTC アーキテクチャ

各ノード上で動作するタスクは，次の二種類に分類される．

(1) **同期タスク**

タスクスイッチの有無やシステムコールの結果について，決定的な動作の保証対象とするタスク．サーバアプリケーションの出力内容に影響を与えるタスク群を同期タスクにする．特定の 1 CPU コア上でのみ動作し，シングルスレッドで動作する．システムコール中は適宜，ノード間で同期をとりながら動作する．

(2) **非同期タスク**

決定的な動作の保証対象としないタスク．OS のハウスキーピング処理や，ロギング，障害監視などを行うタスクなどが該当する. 同期タスクとは別の CPU コア上で動作する．ノード間の同期は行わない．

2.2 動作原理

タスク同期方式ではアプリケーションの振る舞いとそのタイミングをノード間で一致化させるにあたり，タスクスイッチのシリアライズと，システムコールの結果一致化等を行う．他にも様々な対策を行っているが，詳細は先行研究[1][2]を参照されたい．

(1) **タスクスイッチのシリアライズ**

タスクスイッチのシリアライズは，タスクスイッチを行うカーネルコード上の箇所と実行順序を，同期通信によりノード間で一致化する．これにより同期タスクの振る舞いの他，出力のタイミングも揃える．

タスクスイッチ箇所を一致化するにあたり，各同期タスクは FIFO スケジューリングポリシーにて実行される．つまり，タイムスライス切れにより任意のタイミングでプリエンプションされることはない．また，非同期タスクの振る舞いの影響により，非決定性のタスクスイッチが起こる箇所については次のように対処する．ノード間で同期通信を行い，(a) 必ずタスクスイッチして同じ状態になるのを待ち合わせる，あるいは(b) 同じ状態になるのを待ち合わせて，必ずタスクスイッチしないようにする．

(2) **システムコールの結果一致化**

アプリケーションが，同じ入力に対し同じ状態遷移と出力を行うには，システムコールの入力と出力が同じであれば良い．入力については，同期タスクが保持する「状態」や，同期対象のシステムコールから得た値を使っていれば必然的に一致する．出力については，非決定性の要素により結果が変化するケースについてノード間で同期通信を行い，同じ結果を返すよう合意形成する．例えば poll システムコールにて受信メッセージの有無を確認するにあたり，全ノードが受信済みの場合のみ「有り」を返し，そうでない場合は「無し」を返すようにする．

3. コールグラフを活用した非決定性タスクスイッチの抽出手法

タスク同期方式 FTC 用の Linux カーネルをバージョンアップするにあたり，新たな非決定性タスクスイッチが存在しないか, 改めて膨大なソースコードを見直す必要がある．これを効率的に行うため，本研究ではコールグラフを活用した網羅的精査手法を考案，実施した．本章ではその手法について説明する．

3.1 基本アイデアとアプローチ

Linux カーネルにおいてタスクスイッチを行うのは schedule 関数である．したがって，非決定性のタスクスイッチを網羅的に抽出するには，(1) 最終的に schedule 関数を呼び出し得るコールパスを全て抽出し，(2) そのパス上に非決定性の条件分岐が無いか，すなわち非同期タスクの振る舞いによって当該パスに至る場合とそうでない場合に分岐しないか確認すれば良い．

これを実現するため，簡単な解析機能付きのコールグラフ生成ツールを作成した．その解析機能の一つ，マーキング機能は, 特定の関数とその全ての呼び出し元関数に対し，指定のマークを付与してコールグラフを出力する. つまり，schedule 関数に対して SCHED マークを付与すれば，前述の(1) を実現することができる. (2) の自動化は困難なため, 依然として人手による確認が必要だが，それでも全体としては大幅な工数短縮になる．

前述したように，同期処理により非決定性タスクスイッチを排除する必要があるのは，アプリケーションタスクが利用するシステムコールの処理のみである．よって，これ

らのシステムコールについてコールグラフを生成し，ソースコードの精査を実施すれば良い．実際にはサポート対象のシステムコールを 128 種に絞っており（これは従来から実施），精査の範囲を縮小している．

以上の方法で非決定性タスクスイッチを効率的に網羅的に抽出できるが，実際には規模の問題，false-negative の問題，false-positive の問題を解決する必要がある．以降の節にてこれらの説明を行う．

3.2 コールグラフ規模の縮小

Linux カーネルは比較的規模の大きいソフトウェアであり，システムコール一つとっても数万行規模のコールグラフが生成される.そこで,効率的な精査を可能とするため，幾つかの工夫を行った．

(1) 実行バイナリに基づくコールグラフ生成

Linux カーネルは様々な用途に応じるため，多数のコンフィグオプションを用意している．このオプションによりコンパイルされない関数や関数呼び出しも多数あるが，これらは精査対象とはならないため，生成するコールグラフに含めるべきではない．そこでコールグラフの生成にあたっては，Linux カーネルのソースコードではなく実行バイナリを静的解析して生成するアプローチを採用した．これにより，明らかに不要な部位のコールグラフ出力を回避する．

(2) コールグラフ部位の重複排除

Linux カーネルのような大規模ソフトウェアにおいては，同じ関数の呼び出しがコールグラフ上，何度も現れる．これを逐一コールグラフとして出力すると，非常に巨大なグラフとなり，注目すべき関数呼び出しが見えにくくなる．そこで，既出の関数呼び出しについては，以降の子関数呼び出しをコールグラフから省略する．

なお，インライン関数については，この省略の対象外とする．理由は，同じインライン関数であっても，コンパイラによる最適化によって，ある関数呼び出しを含むものと含まないものに分かれる可能性があるためである．図 2 は該当ケースの一例である．もしインライン関数 func1 を func1(arg, true)のように呼び出せば，実行バイナリ上，func2 関数への call 命令が出力される．一方，func1(arg, false)のように呼び出した場合, func2 への call 命令は出力されない．

```
static inline int func1(void *arg, bool flag)
{
        if (flag == ture)
                return func2(arg);

        return 0;
}
```

図 2 状況により子関数の呼び出し有無が変化するケース

(3) コールグラフの折り畳み/展開

数万行規模のコールグラフとなると，例えば func_a の二つの子関数，func_a1 と func_a2 の間が 1 万行以上離れていることもある．そのままでは，精査者は 1 万行も離れたコールグラフを往復しながら確認作業を行うことになるため，非効率である．そこで，コールグラフの折り畳み/展開をサポートし，親関数から子関数へ（抽象から具体へ）向かって概観を把握しながら追っていけるようにした．

(4) コントロールフロー解析

精査の過程では schedule 関数呼び出しに至るコールパス上に，非決定性の条件分岐が無いかソースコードを確認する，と述べた．逆に言えば，そのコールパス上において，条件に依らず必ず一度呼ばれる関数についてはソースコードの確認は不要である．そこで，関数単位でコントロールフロー解析を行い，条件付きで呼ばれる関数と無条件で呼ばれる関数を区別できるようにした．

図 3 はコントロールフロー解析を行った結果を示すイメージ図である．番号付きの四角はそれぞれ BB（Basic Block）を意味しており，BB #1 が関数の入り口，BB #6 が関数の出口（return）である．例えば BB #2 は条件分岐により BB #4 にジャンプする場合と，分岐せずにそのまま BB #3 に入ることがある．BB #3 は無条件分岐により BB #6 にジャンプする．ここで，BB #1 と#6 はどのような経路を通ろうとも必ず一度だけ実行される．このような BB から呼ばれる関数は無条件で呼ばれる関数として，そうでない関数は条件付きで呼ばれる関数として，コールグラフ上に属性を付加する．なお BB #2 については，条件によって実行回数が変化するため「条件付き」に分類している．もし BB #2 の延長にタスクスイッチがあり，かつ非決定性の要素によりループ回数が変化する場合，同期失敗となる可能性があるため，ソースコードレベルの確認が必要である．

以上のようにして無条件関数呼び出しかどうか分かるようにしておけば，当該関数呼び出しを行う箇所のソースコードの確認を省略できる．

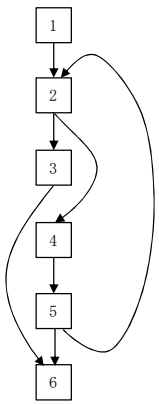

図 3 コントロールフロー解析の一例

3.3 false-negative の排除：間接コールの解決

コールグラフを生成する上で特に問題となるのは間接コール（C 言語で言うところの関数ポインタ呼び出し）の取り扱いである．実行バイナリに基づくコールグラフ生成では，例えば call 命令のオペランドとして静的に記述されているブランチ先アドレスにて呼び出し先の関数名を得ることができる．しかし，間接コールについてはブランチ先が動的に決まるため，どの関数が呼ばれ得るか機械的に調べるのは困難である．

上記問題に対し，今回は人力で間接コールを定義（caller-callee 定義）することで対処した．これは図 4 のように，間接コールの呼び出し元関数（インデント無し）と，呼び出し得る関数群（インデント有り）のペアを列挙する形で定義する．また，未解決間接コールについて<UNRESOLVED>マークを自動付与することにより，どのコールパス上に未解決の間接コールが残っているか容易に把握できるようにした．これにより，関数呼び出しの“漏れ”を防ぐ．

```
$PROTO_OPS_RECVMSG
        netlink_recvmsg
        unix_stream_recvmsg
        unix_dgram_recvmsg
        unix_seqpacket_recvmsg

__sock_recvmsg_nosec
        $PROTO_OPS_RECVMSG
```

図 4 間接コールの定義例：マクロ展開を用いることで，繰り返し登場する間接コールの定義を簡素化できる

3.4 false-positive の排除

実際に schedule 関数に SCHED マークを指定してコールグラフを生成すると，大半のコールパスに SCHED マークが付与されてしまう．これはメモリ確保や排他制御など，プリミティブな関数の延長に schedule 関数呼び出しが存在するためである．

このまま SCHED マークの付いたコールパス上の非決定性条件分岐を探す作業を行うと，結局，多くのソースコードを確認することとなり，省力化できない．しかし実際には，これらの SCHED マークは多くの false-positive を含む．これらを除外することで真に着目すべきコールパスを絞り込み，省力化を実現する．

(1) ホワイトリスト

事実上，非決定性のタスクスイッチが起こらないことが明らかな関数はホワイトリストに登録し，親関数へのマークの伝播を抑止する．

例えばメモリ上の空きページを確保する alloc_pages 関数は，空きページが枯渇している場合，ディスク I/O の完了待ちを伴うページ回収を行うことがある．つまり，alloc_pages は非決定性のタスクスイッチを含む．しかし，このようなメモリ枯渇が起きぬ適切にシステムを設計すれば，alloc_pages は事実上，非決定性のタスクスイッチを含まないといえる．このように実際の利用上，安全に呼び出せる関数をホワイトリストに入れることで false-positive な SCHED マークを削除し，精査範囲を縮小する．

(2) コールバック割り当てによる精度向上

図 5 はコールバックを用いた C 言語のプログラム例である．func3 はコールバック用の関数ポインタを引数 cb にて受け取り，それを呼び出す（間接コール）．そして func1 から func3 が呼ばれた場合は cb1 が，func2 から呼ばれた場合は cb2 が，間接コールにて呼ばれる．

ここで，通常の間接コールと同様に「func3 からは cb1 と cb2 が呼ばれ得る」と caller-callee 定義をすると，不正確なコールグラフが生成されてしまう．すなわち，func1 の延長でも cb2 が呼ばれ，func2 の延長でも cb1 が呼ばれるコールグラフとなるが，実際にはそのようなことは起きない．

そこで，コールパス上のどの関数を経由したかによって間接コール先を変更できる機能を caller-callee 定義に追加し，より正確なコールグラフを生成できるようにした，

```
int func1() {
  return func3(cb1);
}

int func2() {
  return func3(cb2);
}

int func3(int (*cb)(void)) {
  return cb();
}
```

図 5 コールバックを用いたプログラム例

3.5 非決定性タスクスイッチの抽出手順と具体例

非決定性タスクスイッチを抽出するための精査手順を以下にまとめる．

(1) schedule 関数を SCHED でマーキングしたコールグラフを生成
(2) SCHED マーク無し，かつ<UNRESOLVED>有りのコールパスがある場合，その原因となっている間接コールの解決（caller-callee 定義）を行う
(3) SCHED マーク付きのコールパスについて，ソースコードレベルの確認を行う
  (a) false-positive な SCHED マークだと判明した場合，当該関数をホワイトリストに入れて SCHED マークを除去
  (b) 非決定性の条件分岐により当該コールパスを通るか否かが分かれる場合，非決定性のタスクスイッチ有りとして抽出

①	②	③	④
<UNRESOLVED>	SCHED		
<UNRESOLVED>	SCHED		- sys_clone
<UNRESOLVED>	SCHED	U	- do_fork
<UNRESOLVED>	SCHED	CL	- copy_process
	SCHED	C	- copy_creds
		C	+ prepare_creds
		C	+ create_user_ns
	SCHED	C	+ install_thread_keyring_to_cred
		C	+ delayacct_tsk_init
		C	+ init_sigpending
		C	. task_io_accounting_init
		C	+ posix_cpu_timers_init
	SCHED	C	+ threadgroup_change_begin
		C	+ cgroup_fork
		C	+ sched_fork
<UNRESOLVED>	SCHED	C	- perf_event_init_task
<UNRESOLVED>	SCHED	U	+ perf_event_init_context
<UNRESOLVED>	SCHED	C	+ perf_event_free_task

図 6 clone システムコールのコールグラフ（抜粋）と精査の様子

図 6 は子プロセスやスレッドを生成する clone システムコールについて，精査の様子を Excel 形式にて示したものである（重要でない関数呼び出しは非表示にしている）．コールグラフは④のようにインデントによりコール階層を表現する．do_fork は copy_process を呼び出し，copy_process は copy_creds, delayacct_tsk_init, init_sigpending,…等を呼び出す，といった具合である．同じ関数から呼ばれる子関数はソースコード上の登場順にソートして出力されるので，例えばロックを獲得した後に呼ばれる関数かどうかも容易に把握できる．また，「-/+」記号をダブルクリックするとグラフの折り畳み/展開ができるようになっている．

図 6①はその関数の延長に未解決の間接コールを含むことを意味するマーク，②は schedule 関数呼び出しを含むことを意味するマークの付与状況を示している．例えば perf_event_init_task 関数の延長には未解決の間接コールと schedule 呼び出しが存在し，それにより付与された <UNRESOLVED> と SCHED マークは親関数である copy_process, do_fork, sys_clone にも波及している．ここで，perf_event_～等の関数はパフォーマンスカウンタ機能を利用する際に必要な初期化処理を行うものだが，当該 FTC では同期タスクにおけるパフォーマンスカウンタの利用を許可していない．よって，これらの関数は実質何もせず，schedule は呼ばれないし間接コールも起こらない．つまりこの<UNRESOLVED>と SCHED マークは false-positive である．一般に間接コールの解決には手間を要するが，このようなケースでは未解決のままで問題ない．また，install_thread_keyring_to_cred と threadgroup_change_begin にも SCHED マークが付与されているが，ソースコードを確認したところ，これらの関数はスレッドを生成するときにしか呼ばれないことが判明した．前述した通り，同期タスクにはマルチスレッドの使用を認めていないため，これらの関数は呼ばれず, SCHED マークは false-positive といえる．後はこれらの関数をホワイトリストに入れ，マーク付与の対象外とすることで，真に確認すべきコールパスを絞り込むことができる．

図 6③はコントロールフロー解析の結果を示している．U（Unconditional）は無条件に呼ばれる関数, C（Conditional）は条件付きで呼ばれる関数を意味する．SCHED マークが付与されている子関数が全て U 属性であれば，その親関数のソースコードを確認せずとも，そこに非決定性の要素はないと判断できる．なお，L（Loop）はループ内から呼ばれる関数，という意味である．

以上のようにして非決定性のタスクスイッチを抽出し，そこに同期処理を組み込むことで，同期タスクの振る舞いを決定性のものにすることができる．実際にどのような同期処理を組み込むかは先行研究[2]を参照されたい．

4. システムコールの結果一致化調査

システムコールの結果一致化のための調査はコールグラフを活用せず，別の方法で実施している．本報告の主題から外れるため詳細は割愛するが，簡単に調査方法を説明しておく．

システムコールの結果に非決定性の要素が含まれるか否かは，基本的にその外部仕様にて判断できる．例えば EAGIN を返すシステムコールは, 受信メッセージの有無などの非同期的イベントによって結果が左右される．こうしたシステムコールについては従来版と同様，結果一致化のための対策を行うが，非決定性の結果を返す新たなオプション機能がないか確認する必要がある．これについては，システムコールのマニュアルレベルで差分を抽出し，新たな機能仕様に非決定性の結果を返すものがないか確認した．

結果として,新たに問題となるケースは見つからなかった.

5. 提案手法の適用結果

表 1 にコールグラフ縮小施策の効果を示す．まず，コールグラフの重複排除を有効にすると，コールグラフの行数は約 99%削減された．300 万行のコールグラフを確認するのは現実的ではないので，必須の機能と言える．また，ホワイトリストの定義により 17%ほど行数を削減した．さらに schedule 関数に SCHED マークを付与し，精査対象の関数を絞り込むことで,その関数呼び出しの数は 5186 にまで削減できた．なお，全ての false-positive な SCHED マークをホワイトリスト化しているわけではない[a]ので，実際にソースコードレベルの確認を行った関数はさらに少なくなる．

コントロールフロー解析については，U 属性（無条件で呼ばれる関数）は全体の 17%であった．ソースコードレベルの確認を省略できる条件は，SCHED マークが付与された全ての子関数が U 属性であること，である．省略可となるケースは厳密に勘定できていないが，該当ケースは少なく，余り効果は感じられなかった．

表 2 は非決定性タスクスイッチの抽出において，従来手法と提案手法を比較したものである．カーネルコード規模はバージョン 2.6.12 から 2.6.32 になるにあたり 2 倍ほどになっているが，精査の対象としたシステムコール数はほぼ同数であり，精査対象のコード規模もそこまで大きくなってはいない（概算で 1.5 倍程度）.

精査の結果，新たに 2 箇所の非決定性タスクスイッチが見つかり，同期処理を追加する対策を行った．また，2.6.12 にて同期処理を追加した箇所の近傍に別の非決定性タスクスイッチが見つかり，同期処理を行う位置を変更したケースが 2 件あった．従来手法を 2.6.32 カーネルに適用した結果は存在しないので網羅性の比較は単純にはできないが，少なくともこうした変化に気付くことはできた．

精査に要した工数については，期間の粒度でしか情報が残っていないため，作業期間×のべ人数の相対比（小数点第二位以下は切り捨て)にて表現している．またこの作業期間には，純粋に非決定性タスクスイッチの抽出に要した時間を比較するため，コールグラフ生成ツールの開発に要した時間や,間接コールの解決に要した時間を含んでいない．結果として，提案手法により工数を 7 割ほど削減することができ，その有用性を示せた．

品質については，精査後に見つかった不具合（精査時の見落とし）は 1 件のみであった．この 1 件は解析担当者が手順に不慣れであったため，未解決間接コールが残っていたことに起因する．適切に手順通りに実施していれば起きなかった不具合であり,品質維持にも有効であると言える．なお，テストにより当該不具合が見つかった後，提案手法により原因と思しき個所を絞り込むことで，30 分程で原因解明に至っている．そうした後付けの確認にも役立った．

以上により，提案手法は工数削減と品質維持の双方において有効であるといえる．

表 1 コールグラフ縮小効果

コールグラフ縮小施策	コールグラフ規模（行）
重複排除無し	3,280,490
重複排除有り	48,758
重複排除＋ホワイトリスト	40,477
上記＋マーキング	5,186

表 2 従来手法との比較

項目	従来	今回
カーネル	2.6.12	2.6.32
同期対象システムコール数	全 286 中 127	全 312 中 128
非決定性タスクスイッチの対策箇所	64 箇所	新規 2 箇所, 対策箇所変更 2 件
作業期間（相対比）	1.0	0.3
精査後不具合	不明	1 件

6. 関連研究

タスク同期方式以外の FTC 実現方式としては仮想マシン技術を用いた方法[3][4]もある．これらに対し，タスク同期方式は同期負荷が小さく，かつ安定していることから，リアルタイムシステムにも適している．

実行バイナリからコールグラフを生成するツールとしては bscg[5]があるが, bscg は有向グラフによりコールグラフを図示するため，大規模ソフトウェアには向かない．これに対し，自作ツールではインデントにてコール階層を表現し，省スペース化している．

コールグラフを生成するにあたり，網羅性を保証するには間接コールも適切に取り扱う必要がある．間接コールもサポートするコールグラフ生成ツールとしては egypt[6]がある．egypt は関数ポインタの代入箇所を抽出し，そこからポインタの先の関数へのグラフを伸ばす．つまり，間接コールしている場所から当該関数へのグラフが伸びるわけではないので正確性に欠ける．また，関数ポインタテーブルを用いている場合など，多段でポインタ参照している場合は間接コールを抽出できず，網羅性に欠けている．SVF[7]は，LLVM IR を解析し，間接コールも含めてコールグラフを生成する．しかし，実際に Linux カーネルに適用したところ，false-positive, false-negative が非常に多く，実用には

a) 例えば図 6 にある SCHED マークはいずれも false-positive だが，ホワイトリスト化してコールグラフを再生成せずとも，これらが false-positive だとメモ書きしておけば精査する上では十分である．

向かなかった．恐らくカーネルソースコードにインラインアセンブラが含まれることが原因の一つと思われる．以上のように，間接コールについては十分な精度で追跡できるツールが無かったため，今回は人力で caller-callee の関係を定義している．

7. おわりに

タスク同期方式 FTC を Linux 上で実現するには，システムコールの延長で発生する非決定性のタスクスイッチを全て抽出し，適切な同期処理を追加する必要がある．この課題に対し，本研究ではコールグラフを用いた効率的な抽出手法を考案した．実際に FTC 用 Linux カーネルのバージョンアップに適用したところ，従来比で 7 割ほど作業工数を削減でき，その有効性を示せた．また，関連不具合も抽出手順ミスに起因する 1 件のみであり，品質維持にも貢献した．

適切なコールグラフを生成するには間接コールの解決が必須であるが，これを現実的な時間と精度で自動化することができなかったため，今回は人手により約 1 ヶ月かけて実施している．これを高精度で自動化することは今後の課題である．

提案手法の応用としては，例えばサポート対象外のシステムコールを，同期タスクが誤って呼ぶことがないか確認する際にも利用できる．具体的には，サポート対象外のシステムコールにマークを付与し，ライブラリも含めたコールグラフを生成する．そして当該マークが付与されたコールパスについて，実際の利用上，通り得るパスかどうかソースコードレベルで検証すれば良い．

参考文献

[1] 大島訓，他：ソフトウェアによる制御システム向け Fault Tolerant Computer の提案, 第 9 回情報科学技術フォーラム（FIT 2010），第 1 分冊，p.423-426（2010）
[2] 関山友輝，他：ソフトウェアによる制御 FTC の実現に向けたオペレーティングシステム同期機能の開発，第 9 回情報科学技術フォーラム（FIT2010），第 1 分冊，p.427-432（2010）
[3] YaoZu Dong, et al: COLO: COarse-grained LOck-stepping Virtual Machines for Non-stop Service, *ACM Symposium on Cloud Computing* (*SoCC'13*), No. 3, p.1-16 (2013)
[4] VMware: VMware vSphere 6 Fault Tolerance Architecture and Performance, https://www.vmware.com/files/pdf/techpaper/VMware-vSphere6-FT-arch-perf.pdf, (参照 2021-02-01)
[5] 権藤勝彦，鈴木朝也，川島勇人：C 言語用 CASE ツールへの DWARF2 デバッグ情報の応用，コンピュータソフトウェア，Vol.23，No. 2，p.175-198（2006）
[6] Andreas Gustafsson: egypt, https://www.gson.org/egypt/, (参照 2021-02-01)
[7] Yulei Sui and Jingling Xue: SVF: Interprocedual Static Falue-Flow Analysis in LLVM , Proc. *International Conference on Compiler Construction* (*CC '16*), p.265-266 (2016)