

カーネル利用状況とメモリ使用量に着目した Linux Kernel Configの解析手法

栗石 卓耶^{1,a)} 松原 克弥^{1,b)}

概要: IoT やエッジコンピューティングなどが注目されるにともなう、組み込み機器が高機能化している。これらの組み込み機器において、ネットワークスタックなどの豊富なソフトウェア資産をもつ Linux の活用が広がっている。一方、Linux は、バージョンアップにともなう肥大化しており、そのメモリ使用量が組み込み機器への導入の障害となっている。Linux Kernel Config の仕組みは、メモリ使用量削減のために使用しない機能を無効化できるが、その設定項目が複雑なため、システムの目的ごとに必要十分な設定を見つけることが難しい。本研究では、Linux カーネルのソースコードからカーネル内の各関数と Linux Kernel Config 項目との対応関係を機械的に抽出し、アプリケーションの動作中にカーネル内関数の呼び出し状況とメモリ使用量を観測することで、アプリケーション実行に影響を与えず、かつ、メモリ使用量削減効果の高い Linux Kernel Config 項目を特定する。piCore ディストリビューション上で Web サーバが動作するシステムへ本手法を適用した結果、Linux Kernel Config の変更により約 6.5MB のメモリ使用量を削減できた。

A Method of Analyzing the Linux Kernel Configurations Focusing Kernel Usage Statics and Actual Memory Consumptions

SHIZUKUISHI TAKUYA^{1,a)} MATSUBARA KATSUYA^{1,b)}

Abstract: The features of embedded devices are becoming more functionalized and complicated, as IoT and edge computing attract attention recently. Linux is being adopted spreadly for these embedded devices since it has abundant software assets such as network stack. However, unfortunately, the memory amount required by Linux could be one of the most obstacles for embedded products because the kernel is bloating at every its version updated. We can disable unused kernel features by specifying in the Linux kernel configuration, but it must be hard to find an enough configuration setting because of lots of complicated configuration items. This research proposes a method to mechanically extract the correspondence between each Linux Kernel Config item and kernel internal functions and to profile the kernel function invocations and actual memory consumption related to each Config item while an application is running. As a result, profiled outputs can help for Linux kernel tinification with finding kernel config items which could be less used and consume large memory. In this paper, we also show an experimental result that indicates a kernel running in a Ras-Pi based embedded system could shrink 6.5MB after applying tinification with the proposed method.

1. はじめに

IoT 製品やエッジコンピューティングなどが注目されるにともなう、組み込み機器に求められる機能が複雑化し

ている。Linux は、ネットワークスタックなどの豊富なソフトウェア資産があり [1]、組み込み用途の OS としても注目されている [2]。

一方で、Linux Kernel は、従来の組み込み向け OS と比較して、メモリ使用量が大いという課題がある。さらに、Linux Kernel は、バージョンと共に様々な機能やプラットフォームのためのコードが追加されており、肥大化している (図 1 参照)。組み込み機器のアプリケーションが、Linux

¹ 公立はこだて未来大学
Future University Hakodate

a) b1015208@fun.ac.jp

b) matsu@fun.ac.jp

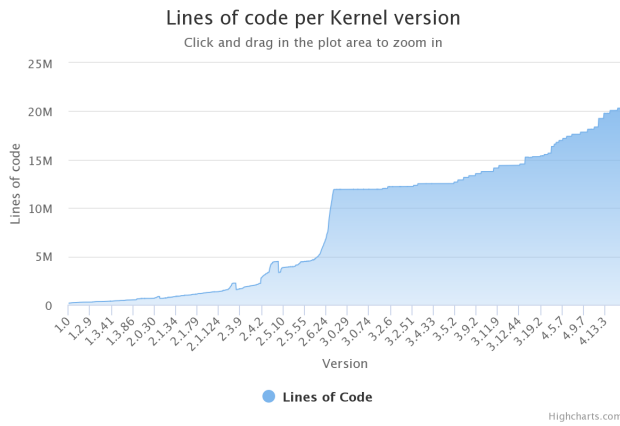


図 1 Linux Kernel のバージョンとソースコードの行数の関係
出典：The Linux Counter Project[3]

Kernel のもつ全ての機能を利用しない場合、利用されない機能のためのコードやデータが、メモリ使用量の負担となりうる。メモリなどの計算資源が限られる組み込み機器においては、利用されない機能を無効にし、メモリ使用量を削減することが重要である。

目的に応じて必要十分な最小限の機能を持った Linux Kernel を作成する作業を、Linux Kernel Tinification と呼ぶ。Linux Kernel には、Kernel Config と呼ばれる、コンパイル時に Kernel 機能を取捨選択する機構がある。この機能を用いて、アプリケーションの動作に必要な機能のみを有効にすることで、Linux Kernel のメモリ使用量を削減できる。しかし、Kernel Config の項目数は約 17,000 もあり^{*1}、また、それぞれに依存関係が存在する場合もある。そのため、アプリケーションの動作に必要な機能を手動で特定することは難しい。

本研究では、アプリケーション動作時の Kernel を動的に解析することで、アプリケーションが使用する Linux Kernel 内の関数 (Kernel 関数) を特定する。また、Linux Kernel のソースコードを解析することで、Kernel 関数と Config 項目の対応関係、Config 項目とメモリ使用量の対応関係を特定する。これらの解析により、アプリケーションの動作に必要なメモリ使用量を増大させている Config 項目の特定を支援し、効率の良い Linux Kernel Tinification を実現する。さらに、本稿では、超小型システムを目的とした Linux ディストリビューションのひとつである Tiny Core Linux の Raspberry Pi 向けパッケージ piCore を用いて、本手法の有効性を検証するための実験結果について示す。

2. Linux における Kernel Configuration システム

Kernel Config は、Linux Kernel のコンパイル時に、その機能を取捨選択するための機構である。Kernel のソー

^{*1} Linux Kernel v4.19 で計測

表 1 Config 項目の型と Linux Kernel v4.19 における項目数

型	設定できる値	項目数
bool	y(有効) か n(無効)	7783
tristate	y(有効), m(カーネルモジュールとして有効), n(無効) のいずれか	8368
string	文字列	86
hex	16 進数	104
int	整数	567

スコードに含まれている、図 2 に示す menuconfig などの設定ツールを用いることで、Config 項目ごとの設定を変更できる。図 2 の Config 画面の各行は、Config 項目または Config 項目の集まり (メニュー) であり、メニューを展開することで、更に詳細な Config を変更できる。

各 Config 項目は、Kernel ソースファイルの各所にある Kconfig という名前のファイルによって定義されている。図 3 の例では、「SCSI」が Config 項目の名前である。「SCSI device support」は、設定ツール内の項目に表示される文字列であり、prompt 属性と呼ばれる。Config 項目の中には prompt 属性が無く、menuconfig に表示されないものもある。そのような Config は、Kernel Config の状態をわかりやすくするためのダミーとして、Kernel ソースコード内部で使用される。また、各 Config 項目には型があり、自身の機能の状態に関する値をもつ。Kernel Config における型の種類と設定可能な値の一覧、および、Linux Kernel v4.19 における型別の Config 項目数を表 1 に示す。図 3 の例では、Config 項目の型が tristate である。したがって、SCSI は、「y(有効)」か「m(カーネルモジュールとして有効)」か「n(無効)」のいずれかの値に設定できる。また、Config 項目は、依存関係をもつことができる。図 3 の記述は、Config 項目が、BLOCK という他の Config 項目に依存していることを示している。そのため、BLOCK Config 項目の値が y か m のときにしか、SCSI Config 項目を y か m にできない。

設定ツールによって設定された Config 項目は、各 Config 項目名に接頭辞「CONFIG_」をつけたパラメータとして、Makefile やソースコード内で設定値を参照できる。図 4 の Makefile の例では、「SCSI」の値によって、scsi_mod.o ファイルを生成するかどうかを切り替えている。図 5 のソースコードでは、CPP ディレクティブを用いて、「SCSIPROC_FS」の値によって、ソースコードの一部の記述を切り替えている。Kernel Config は、Config 項目の値によって、生成される.o やソースコードを変化させることで、機能の取捨選択を実現している。

3. Kernel Config 調整支援による Tinification 手法

本研究では、Linux Kernel Tinification を支援するために、アプリケーション動作の解析を通して、Kernel Config

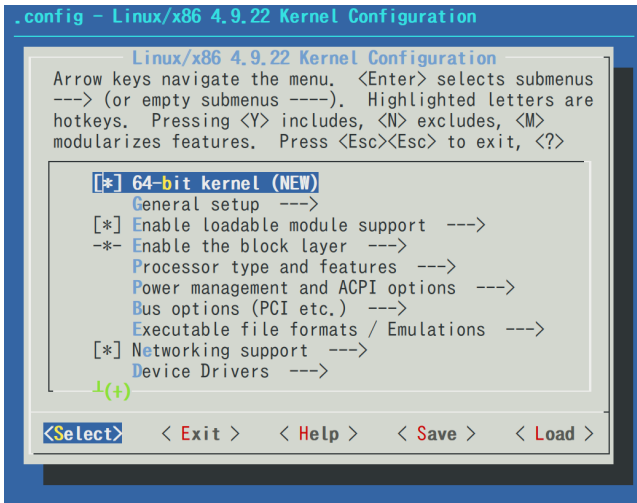


図 2 menuconfig の実行画面

```
config SCSI
    tristate "SCSI device support"
    depends on BLOCK
    ...
```

図 3 Kconfig ファイルによる Config 項目の定義例

```
obj-$(CONFIG_SCSI) += scsi_mod.o
...
```

図 4 Makefile による Kernel Config 項目の値の参照例

```
#ifdef CONFIG_SCSI_PROC_FS
static int
sg_idr_max_id(int id, void *p, void *data)
{
    int *k = data;
    ...
}
```

図 5 Linux Kernel のソースコードによる Kernel Config 項目の値の参照例

項目ごとの利用頻度と使用メモリ量を計測し、提示する手法を確立する。以降、本研究における解析手順について述べる。

3.1 解析対象とする Config 項目

本手法において解析対象とする Config 項目は、次のすべての条件を満たすものとする。

- (1) Config 項目の型が、bool か tristate である
bool や tristate 以外の型をもつ Config 項目は、対応する機能を無効化するための値が自明ではないため、解析の対象から除外する。
- (2) Tinification を実施する前の Kernel Config(以降、デフォルト Kernel Config) の値として、y か m が設定されている

Tinification 実施前から無効となっている Config 項目はアプリケーション動作に必要なではないことが自明であるため、解析の対象から除外する。

- (3) Config 項目が、prompt 属性を持っている

prompt 属性を持たない Config 項目は、設定により無効化できないため、解析の対象から除外する。

3.2 Config 項目ごとの利用頻度とメモリ使用量の解析手順

本提案手法では、以降に述べる 3 つの手順により、Config 項目ごとの利用頻度とメモリ使用量を計測する。はじめに、起動からアプリケーション動作中までの Kernel をプロファイリングして、アプリケーションが利用する Kernel コード片を特定する。次に、Linux Kernel のソースコードを静的解析して、Config 項目に対応する Kernel コード片を特定する。最後に、対象の Config 設定のみを変更した Kernel を動作させた環境で、各 Config 項目を無効にしたときに削減できるメモリ使用量を計測する。

3.2.1 手順 1: アプリケーションが利用する Kernel コード片の特定

Linux Kernel のトレース機能を用いて、OS 起動からアプリケーション動作中までの各 Kernel 関数の呼び出し回数を計測する。デフォルト Kernel Config で設定された Kernel でトレース機能を有効にし、アプリケーションのテストケースを実行する。アプリケーション動作に必要な Kernel 関数の呼び出しが十分に行われるまでトレースを続ける。

次に、Kernel 関数と Kernel コード片の関係を特定する。本研究における Kernel コード片は、単一のソースファイルや CPP(C プロプロセッサ) ブロックであると定義する。Kernel の実行ファイルである vmlinux や、カーネルモジュールを静的解析することで、トレースログにおける各 Kernel 関数のアドレスから、その関数が定義されているソースファイル名と行番号が得られる。一連の解析によって、アプリケーション実行にともなって、Kernel コード片ごとに何回実行されたかという値を計測できる。

3.2.2 手順 2: Config 項目に対応する Kernel コード片の特定

Config 項目に対応する Kernel コード片とは、各 Config 項目を無効にしたときに削減されるコード片であると定義する。Config 項目を無効にすることで、削減される Kernel コード片を調べ、Config 項目に対応する Kernel コード片を特定する。

解析対象の Config 項目が他の Config 項目と依存関係をもっている場合、その項目に依存している他の項目について、Kernel Config の状態に矛盾がなくなるまで、再帰的に無効にする処理を行う必要がある。例えば、図 6 の場合、CONFIG.A だけを無効にすることはできず、CONFIG.B も無効にしなければならない。図 7 の場合には、CON-

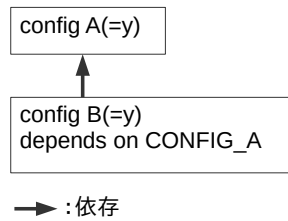


図 6 Config 項目の依存関係の例 1

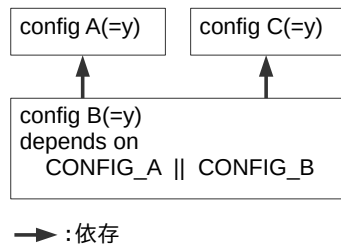


図 7 Config 項目の依存関係の例 2

FIG_A の無効化にともなって CONFIG_B を無効にする必要はない。CONFIG_A だけを無効にしても Kernel Config の状態に矛盾は生じず、また無効にする Config 項目は少ないほうが、Config 項目について、より正確な利用頻度とメモリ使用量を計測できるためである。また、図 6 の例において、CONFIG_A を無効にする過程で CONFIG_B を無効にした場合は、CONFIG_B に関しても、それに依存する項目について同じような点について留意しなければならない。

CPP ブロックと Config 項目の対応関係について特定するために、カーネルソース内の CPP ディレクティブを解析して、各 CPP ディレクティブが有効となる条件を特定する。まず、デフォルト Kernel Config で有効となる CPP ブロックの一覧を得る。次に、デフォルト Kernel Config から解析対象の Config 項目を無効化した設定においても有効となる CPP ブロックの一覧を得る。前者と後者の一覧を比較し、対象 Config 項目に対応する Kernel ソースコード内の CPP ブロックを特定する。

ソースファイルと Config 項目の対応関係の特定は、Kernel ソースコード内の Makefile に対して、CPP ブロックの解析と同様の手順を適用する。デフォルト Kernel Config でコンパイル対象となるソースファイル一覧と、解析対象の Config 項目を無効化したときにコンパイル対象となるソースファイル一覧を比較することで、対象 Config 項目に対応するソースファイルを特定する。

前述の解析により、各 Config 項目に対応する Kernel コード片を特定できる。

3.2.3 手順 3: 各 Config 項目に依存するメモリ使用量の特定

前述の手順 2 に沿って、デフォルト Kernel Config から

解析対象 Config 項目を無効化した Kernel を作成し、実際にその起動したシステム環境でアプリケーションのテストケースを実行した状態でのシステムのメモリ使用量を測定する。解析対象の Config 項目についてメモリ使用量を測定し、デフォルト Kernel Config の Kernel が動作するシステム環境におけるメモリ使用量と比較することで、Kernel Config ごとのメモリ使用量を計算する。

4. 実装

4.1 手順 1: アプリケーションが使用する Kernel のコード片の特定

Linux Kernel に実装されたトレース機能 ftrace の一部である kernel function profiler を用いて、アプリケーション動作中に利用される Kernel 関数を特定する。Linux Kernel に実装されている kernel function profiler の機能だけでは、本提案手法の解析に必要な情報が不足するため、Kernel 関数のシンボルアドレスやオフセット、実行ファイル名 (vmlinux や kernel module 名) を合わせて出力するように kernel function profiler を改造した。さらに、kernel function profiler によるプロファイリングを Kernel 起動直後から有効化するために、initramfs 内の init コマンドを編集し、init 処理開始の直前に本プロファイリング機能を有効にする処理を追加した。

kernel function profiler は、Kernel 内の関数呼び出しをメモリ空間内のアドレスとして記録する。記録されたアドレスは、addr2line ツールを用いて Kernel ソースコード内の行番号に変換することで、当該行のコードを含む関数が、利用された Kernel 関数として特定できる。

4.2 手順 2: Config 項目に対応する Kernel のコード片の特定

Kernel ソースコード内の CPP ブロックに関する情報を抽出するために、Julio Sincero らが実装したツール UNDERTAKER[4] を用いた。UNDERTAKER は、C 言語のソースコードを静的解析して、CPP ブロックの範囲と条件式を抽出する機能をもつ。UNDERTAKER を Kernel ソースコードすべてに対して実行し、すべての CPP ブロックの範囲と条件式を求めた。

また、UNDERTAKER から得られた CPP ブロックの条件式を評価するためのプログラムを実装した。このプログラムは、デフォルト Kernel Config と解析対象の Config 項目を無効にした Config について、CPP ブロックの条件式を評価し、除外される CPP ブロックを特定する。なお、本プログラムは静的解析にもとづいているため、動的に値が決まるような変数を含む条件式が評価できないという問題がある。本提案手法の実装では、Config 項目以外の条件が含まれる CPP ブロックは、常に有効となると解釈することで、誤検出を防ぐこととした。

Kernel のコンパイル対象となるソースファイルを特定するために、Christian Dietrich らが開発した GOLEM[5] を用いる。GOLEM は、Linux Kernel の Makefile を改造することで、実際に Kernel をコンパイルせずとも、コンパイル対象となるソースファイルを効率的に特定する機能をもつ。デフォルト Kernel Config と解析対象の Config 項目を無効にした Config を、Linux Kernel に適用し、その都度 GOLEM を実行することで、除外されるソースファイルを特定した。

4.3 手順 3: 各 Config 項目を無効にしたときに削減できるメモリ使用量の特定

3.2.3 節で述べた手順 3 では、解析対象の Config 項目の数だけ Kernel を作成して、それらを実際に起動して解析を行う必要がある。本実験では、Kernel を起動するプラットフォームを仮想化環境 (VM) を用いて多重化することで、解析時間の短縮を図った。また、解析作業を自動化するため、複数の実行環境の制御と解析の実行、結果の収集を自動的に行うシステムを実現した。図 8 は、実現した解析自動化システムの構成を示している。解析用のシェルスクリプトを用いて、設定された並列数だけ VM を生成し、それぞれの VM で Config 設定の異なる Kernel を起動する。また、解析結果を受け取るために、VM のシリアルデバイスを Kernel ごとのファイルにリダイレクトする仕組みを実装した。並列起動された VM を非同期にテストするために、テストサーバによる集中制御を実装した。テストサーバと各 VM は仮想ブリッジと Tap デバイスにより接続されており、HTTP を用いて相互に通信可能である。VM は、OS の起動が完了すると自動的にアプリケーションを起動し、テストサーバへテストを要求する。テストサーバは、テストの要求を受けて対象の VM をテストし、テスト完了後に終了命令を送信する。VM は終了命令を受けると、free コマンドの「used - buffer - cache」の値を解析結果としてシリアルデバイスへ出力し終了する。VM が終了すると、シェルスクリプトは新たな VM を起動し、未テストの解析対象 Kernel がなくなるまで繰り返す。

5. 評価

本手法の効果を評価するため、サンプルアプリケーションの Kernel Tinification を行った。Apache サーバーを用いて静的ページを配信する Web サーバーを piCore 上に構築し、Raspberry Pi で動作させた。

5.1 実験条件

Raspberry Pi 2 に piCore v9.0.3(Linux Kernel v4.9.22) を導入し、その上に Apache サーバーをインストールした。Apache サーバーのドキュメントルートには、49 バイトの index.html ファイルを配置した。また、サンプルアプリ

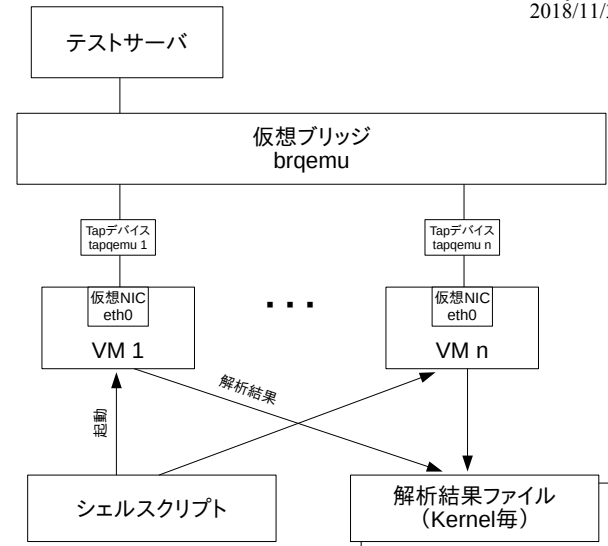


図 8 解析自動化システムの構成

表 2 ログの解析を行った VM 環境

OS	Ubuntu 18.04.1 LTS
CPU	Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz 32 コア vCPU
メモリ	96GB vRAM

ケーションのテストのための環境を構築した。piCore が起動すると、テストサーバから Raspberry Pi 上の Apache に対して、次の内容のリクエストを 10 回繰り返す。

- (1) index.html を GET する
- (2) サーバ上に存在しないファイル NotFound.html を GET する

piCore のデフォルト Kernel と本提案手法により Tinification を行った Kernel のそれぞれを用いてテストを行い、ログの収集、解析を行った。ログの解析プログラムを実行した VM 環境に関する情報を表 2 に示す。なお、手順 3 における VM を用いた解析は、ハイパーバイザとして QEMU を用い、Raspberry Pi の仮想環境を複数用意した。また、各解析における測定では、3 回の解析結果の平均を測定結果として採用した。Tinification を行った Kernel Config は、piCore のデフォルト Kernel Config に対して「呼び出し回数が 0 かつ、メモリ使用量が減る」と判断された Config 項目 (表 4 の A 群) をすべて無効化したものである。

さらに、解析結果の変動について評価するため、解析対象の Config 項目から無作為に選んだ 9 個の項目のメモリ使用量測定結果に対して評価を行った。それぞれの項目について 32 回評価を行い、解析結果の分布について調査した。

5.2 結果

本提案手法適用前後におけるメモリ使用量、Kernel イメージの容量、initramfs の容量の変化を表 6 に示す。default 行は、手順 1 のトレースによって得られた値である。本提案手法による Tinification 適用後の値は、本提案手法による解析の結果、必要ないと思われる Config 項目を無

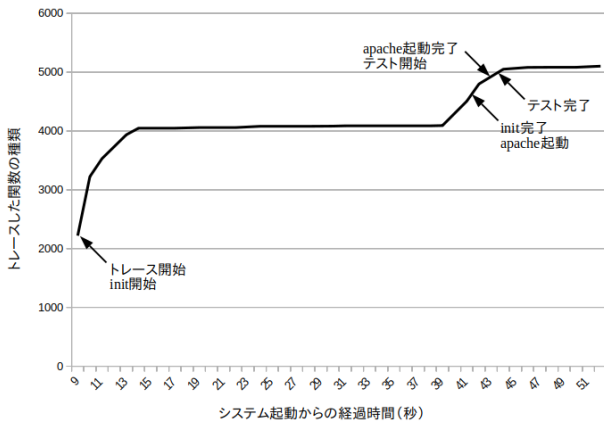


図 9 トレースした関数の種類と時間の関係

効にした Kernel で計測した値である．本提案手法をサンプルアプリケーションに適用することで，実際にメモリ使用量の削減が可能であることを確認した．また，vmlinux や initramfs についても，容量が削減されていることを確認した．

各 Config 項目の解析結果を，呼び出し回数とメモリ使用量で分類した結果を表 4 に示す．Config 項目を呼び出し回数によって「0 回，1～9 回，10 回以上」の 3 つに分類し，さらにメモリ使用量によって「減少した，増加した，起動に失敗した」の 3 つに分類した．

本実験で無効にした呼び出し回数が 0 の Config 項目のうち，削減されるメモリ使用量が大きいと判断された上位 10 項目を表 5 に示す．表 5 を見ると，NFS_FS(ネットワークファイルシステム) や BTRFS_FS(B-tree file system) などが必要のない Config 項目だとわかる．

図 9 は，本提案手法の手順 1 における，トレースした関数の種類と経過時間の関係を．横軸は時間であり，システム起動からの経過時間である．縦軸はトレースした Kernel 関数の種類である．各矢印は，システム内におけるコマンドの起動や完了の時刻を表している．図 9 のトレース終了間際に着目すると，トレースした関数の種類があまり変化していないとわかる．

図 10 は，本提案手法の手順 3 における測定したメモリ使用量の分布についてプロットしたグラフである．横軸は解析対象の Config 項目名である．縦軸は，解析対象の Config 項目を無効にした Kernel のメモリ使用量である．図中のひげは，解析結果の最大値と最小値を表し，箱の高さは四分位範囲を表し，箱の線は中央値を表す．

また，本提案手法の実施において，解析に要した時間を表 3 に示す．

5.3 考察

表 6 において，最も削減割合が高かったのは initramfs であった．これは，超小型システムを目的とした piCore

表 3 本手法の適用に要した時間

手順	要した時間
手順 1	1:26
手順 2	2:47:15
手順 3	9:59:35
合計	12:48:17

表 4 Config 項目の解析結果の分類と傾向

	呼び出し回数	メモリ使用量	項目数
A 群	0 回	減少した	307
B 群	0 回	増加した	277
C 群	0 回	起動に失敗した	575
D 群	1 回～9 回	減少した	11
E 群	1 回～9 回	増加した	2
F 群	1 回～9 回	起動に失敗した	13
G 群	10 回以上	減少した	33
H 群	10 回以上	増加した	20
I 群	10 回以上	起動に失敗した	26
合計			1264

表 5 呼び出し回数が 0 であり，メモリ使用量が大きいと判断された上位 10 件の Config 項目

Config 項目	メモリ使用量 (KB)
NFS_FS	1434.6
BTRFS_FS	1196.0
IEEE802154_CC2520	1166.6
NLS_CODEPAGE_737	1144.0
DM_CRYPT	1136.0
TASK_IO_ACCOUNTING	1126.6
NETFILTER_XT_TARGET_CONNMARK	1105.3
L2TP_ETH	1072.0
ECRYPT_FS	1054.6
HID_CYPRESS	1041.3

表 6 本手法適用前後におけるメモリ使用量，vmlinux の容量，initramfs の容量の変化

	メモリ使用量	vmlinux	initramfs
default	78.9MB	13.8MB	23.0MB
本手法による Tinification 適用後	72.3MB	12.1MB	16.7MB
削減量	6.5MB	1.7MB	6.2MB
削減割合	8.3 %	12.3 %	27.2 %

が，メモリ使用量の削減のために，Kernel の機能の多くを kernel module として有効にしているためであると考えられる．一方，メモリ使用量の削減割合は他 2 つと比較して低く，8.3 %にとどまった．その理由として，piCore ディストリビューションの軽量さが考えられる．piCore ディストリビューションは，メモリ使用量に対しても最小限となるように設計されており，デフォルト Kernel Config の時点で，既にある程度軽量であったと考えられる．デフォルトの状態で必要のない機能が少ない Linux Kernel に対して本手法を適用したため，削減量も比較的少なくなったと考えられる．

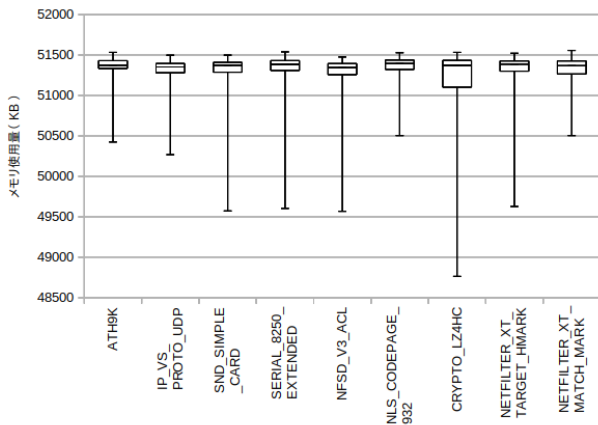


図 10 Config 項目別メモリ使用量解析結果の変動

各 Config 項目の解析結果の分類と傾向に関して、表 4 の結果では、299 個の Config 項目で、無効化することによりメモリ使用量が増加したという結果が出た。この理由として、メモリ使用量の解析の偏差の影響が考えられる。図 10 の結果から、どの Config 項目もメモリ使用量測定値の最大と最小の幅が大きく、たかだか 3 回の計測では、その誤差の影響を排除するのに不十分であったと考える。表 4 の 4 行目を見ると、呼び出し回数が 0 回にもかかわらず、QEMU での起動に失敗した項目が 575 個あるとわかる。これは、本手法のトレースが完全ではないためであると考えられる。例えば、本提案手法のトレースは、init コマンド直前から有効になっているため、それ以前の Kernel の挙動に関してはプロファイリングできない。トレースや解析の精度を上げることで、この問題に対処する必要がある。一方、表 4 の 5 行目以降を見ると、呼び出し回数が 0 でないにもかかわらず、QEMU での起動に成功した Config 項目が 66 個あるとわかる。これらは、動作しているがアプリケーションには必要のない Config 項目であると考えられる。例えば、初期化処理などのために呼び出されたが、以後使用されていない機能などである。このような Config 項目は、呼び出し回数が 0 でなくとも無効にできる可能性があり、本提案手法が、より幅の広い Kernel Tinification の実現に有効であることを示している。

表 5 に列挙されたメモリ使用量の多い Kernel Config 項目の名前からその機能を類推すると、本実験のサンプルアプリケーションでは利用していない機能に関する Config であることが類推できる。また、最も多いメモリ使用量をとまなう Config 項目として特定された NFS_FS は、池田 (2006) の研究においても、メモリ使用量の大きな Config 項目として紹介されている。したがって、本実験で無効にした Config 項目は、Kernel Tinification として妥当なものであったと考えられる。

本実験の手順 1 において、トレース時間が十分であったことを確認した。関数の種類が十分に収束してからトレース

スを終了しているの、トレースが十分に行われていたと考えられる。一方、システム起動から 40 秒付近で、関数の種類に急激な増加が起きている原因が不明であるという課題がある。当該箇所は init 処理の途中であり、今回の実験では、その中身まで調査することができなかった。必要となる Kernel の機能を余分に増加させている可能性もあるため、今後の詳しい調査が必要である。

QEMU を用いたメモリ使用量計測の偏差に関して、図 10 の結果では、それぞれの四分位範囲は、おおよそ 200KB ~ 400KB に収まっているが、最大値と最小値の範囲は 2MB を超えているものもある。本実験によって削減されたメモリ使用量が 6.5MB であることを考慮すると、この偏差は受容できないものであると考えられる。QEMU を用いたメモリ使用量計測の偏差を少なくするために、計測回数を増やす方法が考えられる。しかし、この方法はログの解析時間とトレードオフの関係にあるため、最適な計測回数を設定することが重要である。また、本実験では検証できなかったが、Raspberry Pi といった実際のプラットフォームにおいても、メモリ使用量の偏差が大きい可能性がある。そのため、実際のプラットフォームにおいても追加の実験や調査が必要である。

表 3 に示すとおり、本提案手法の解析作業に要した時間について、計 13 時間弱を要した。この解析時間が長いかわりかは、Tinification 実行者のおかれる環境によって変化するため、断定することはできない。しかし、本手法のログの解析は自動化されているため、Tinification 実行者への負担は少ないと考えられる。この解析時間を短縮するために、解析時間の最も長い手順 3 の改善が考えられる。手順 3 では、大量の Kernel をコンパイルする必要があるため、解析に時間がかかっていると考えられる。表 4 のから、今回の実験では 1264 回の Kernel のコンパイルが必要であったとわかる。1 回のコンパイルに要する時間は、Kernel Config の内容やキャッシュの状態によって異なるが、今回の実験環境では、おおよそ 20 秒程度であった。これを 1264 回行くと仮定すると、コンパイルに要する時間が約 7 時間であるという計算ができ、手順 3 にかかる時間の大部分を締めていることが推測できる。よって、Kernel のコンパイルの高速化の手法について、検討していくことが重要である。

6. 関連研究

徐 振宇ら (2016) の研究では、ユーザーランドのすべてのアプリケーションを静的解析し、アプリケーションが使用しないシステムコールを特定することで、その実装を Kernel から排除する手法を提案した。IoT システムを対象とした Kernel Tinification を行う点で本研究の目的と類似している。しかし、本研究では、メモリ使用量を測定するために動的解析を採用した。徐 振宇らの提案する静的解

析手法では、すべてのアプリケーションから1度も呼び出される可能性のないシステムコールのみを削減するのに対し、本研究が提案する動的解析手法では、呼び出し回数を実メモリ使用量への効果を考慮しつつ無効化する Config 項目を選択することができる。特に、実メモリ使用量の削減効果の高い Config 項目や利用頻度を特定できることは、手動によるコード改変をとまなう Tinification にも有用であると考えられる。

Josh Triplett ら (2014) は、Kernel シンボルの解析やシステムコールの削減、Link-Time Optimization を用いて、Kernel イメージ (vmlinux) の軽量化を行った [6]。Kernel ソースコードの改変を必要とするシステムコール削減には、実装に対する深い理解と複雑な作業をとまなう。一方、本研究の提案手法では、Config 項目ごとの呼び出し回数とメモリ使用量にのみ着目して、Config 設定を無効化することにより Tinification を実現しており、Kernel 実装に対する知識を必要としない。

Kernel Config と Kernel イメージサイズやメモリ使用量の関係を調査した研究として、池田 (2006) の研究がある。この研究では、各 Config 項目のみを有効にした Kernel をコンパイルし、実際に起動して、Kernel イメージサイズや Kernel のメモリ使用量の関係を測定した [7]。Config 項目ごとのメモリ使用量に着目している点で、本研究と類似している。しかし、メモリ使用量の測定を OS 起動直後に行っており、アプリケーション動作にとまなうメモリ使用量の影響を考慮していない点で、本研究と異なる。また、各 Config 項目の呼び出し回数を考慮していないため、アプリケーションの動作に影響を与えず無効化できる Config 項目の特定が難しい。

Anil Kurmus ら (2013) の研究では、アプリケーションを動作させた Linux Kernel を動的解析することで、余分な機能と適切な Kernel Config を特定し、Kernel イメージサイズの削減を行った [8]。また、この研究では、本研究の提案手法の手順2の実装で用いた UNDERTAKER と GOLEM の2つのツールを用いた解析を行っている。Kernel 関数を対象に動的解析している点や、Kernel のコード片を解析しているという点、UNDERTAKER や GOLEM を用いている点で、本研究の手法と関連性が深い。しかし、無効にする Config 項目の決定方法が本研究とは異なる。Anil Kurmus らの手法では、Config 項目の呼び出し回数によって無効にする Config 項目を決定しているが、本研究では Config 項目ごと呼び出し回数とメモリ使用量の両方を考慮することによって、無効化する Config 項目を選択できる。例えば、「ある Config 項目の呼び出し回数は1だが、メモリ使用量が大きいため、Kernel ソースコード内の呼び出し箇所の改変を含めた Tinification を検討」するような場合にも本提案手法による解析結果を用いることができる。

7. まとめ

本研究では、アプリケーション動作時の Kernel を動的に解析することで、動作に不要な機能を特定し、効率の良い Kernel Tinification を支援する手法について提案した。OS 起動からアプリケーション動作中に利用される Kernel コード片を動的解析により特定し、静的解析により特定した Kernel コード片と Config 項目の対応用いて、各 Config 項目の利用頻度を計算する。さらに、各 Config 項目を無効化したときのメモリ使用量の変化を計測することで、Config 項目が与えるメモリ使用量への影響も示すことで、優先して無効化すべき Config 項目の判断材料とする。本提案手法の実現では、改良した Kernel トレース機能に加えて、関連研究で実装された UNDERTAKER と GOLEM の2つのツールを用い、さらに、解析作業の自動化のためのシステムを構築した。また、本稿では、軽量 Linux ディストリビューション piCore を用いた組み込みシステムを構築して、本提案手法の有効性を検証する実験について述べた。実験結果では、システム全体のメモリ使用量に対して、約 8.3 % の削減効果があることを確認した。

今後の課題として、解析時間を短縮するための最適化手法の考案がある。本稿で示した実験では、解析作業に約 13 時間の時間を必要とした。解析手順の見直しや、解析プログラムの最適化によって実行時間の短縮を図る方法を検討すべきである。2つ目の課題は、Kernel Tinification の評価項目の追加である。組み込み用途の OS に求められる特性は、メモリ使用量の少なさだけではない。起動時間の短さや、消費電力の少なさなど、他の特性にも着目して Kernel Tinification を行う手法を確立したい。

参考文献

- [1] 佐藤 喬, 多田好克: 用途に応じて UNIX システムを削減するための支援機構の設計, 技術報告 101(2006-EMB-002), 電気通信大学大学院情報システム学研究所, 電気通信大学大学院情報システム学研究所 (2006).
- [2] 中島達夫: 組込みシステム用標準 OS としての Linux, 情報処理, Vol. 43, No. 2, pp. 148-153 (2002).
- [3] Lhner, C.: Funny Statistics for the Linux Kernel - Lines of Code, Bad words, Good words, The Linux Counter Project (online), available from (<https://www.linuxcounter.net/statistics/kernel>) (accessed 2018-10-26).
- [4] Sincero, J., Tartler, R., Lohmann, D. and Schröder-Preikschat, W.: Efficient Extraction and Analysis of Preprocessor-based Variability, *SIGPLAN Not.*, Vol. 46, No. 2, pp. 33-42 (online), DOI: 10.1145/1942788.1868300 (2010).
- [5] Dietrich, C., Tartler, R., Schröder-Preikschat, W. and Lohmann, D.: A Robust Approach for Variability Extraction from the Linux Build System, *Proceedings of the 16th International Software Product Line Conference - Volume 1, SPLC '12*, New York, NY, USA, ACM, pp. 21-30 (online), DOI: 10.1145/2362536.2362544 (2012).

- [6] Triplett, J.: Linux Tinification, Linux Plumbers Conference 2014 (online), available from <https://events.static.linuxfound.org/sites/events/files/slides/tiny.pdf> (accessed 2018-10-27).
- [7] 池田宗広: Linux カーネルサイズ・使用メモリ検証 / 改善 P J , NEC OSS 推進センター (オンライン), 入手先 https://elinux.org/images/c/c2/Size_tool_esec_2006-06-29_jp.pdf (参照 2018-10-27)
- [8] Kurmus, A., Tartler, R., Dorneanu, D., Heinloth, B., Rothberg, V., Ruprecht, A., Schröder-Preikschat, W., Lohmann, D. and Kapitza, R.: Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring., *NDSS* (2013).