システムソフトウェアと
オペレーティング・システム 87－1
（２００１．６．２８）

特定用途のための汎用OSのサイズ縮小に関する考察

森若和雄 * 中西恒夫 † 福田晃 ‡
* 九州大学 システム情報科学府 † 奈良先端科学技術大学院大学 情報科学研究科
‡ 九州大学 システム情報科学研究院
*moriwaka@f.csce.kyushu-u.ac.jp †tun@is.aist-nara.ac.jp
‡fukuda@f.csce.kyushu-u.ac.jp

あらまし

組み込みシステムに汎用 OS を利用する場合、そのサイズが問題となる。本研究では、特定のユーザーソフトウェアだけを実行するようなシステムに汎用 OS を適用する場合に、汎用 OS 中の不要なコードを自動的に削除することでサイズを縮小することを考察した。削除の単位をプログラム言語の文とする場合、関数とする場合のそれぞれについて考察した。

実際の OS を利用してどの程度サイズ縮小ができるのか試行した。MINIX を対象として、init を動作させるために必要な最低限の関数を残す場合、モジュールごとに 10%〜30%程度のコード削減を確認した。

Size Reduction of Multi-purpose Operating Systems for Application-specific

Kazuo Moriwaka * Tsuneo Nakanishi † Akira Fukuda *
*Graduate School of Information Science and Electrical Engineering, Kyushu University
†Graduate School of Information Science, Nara Institute Science and Technology

Abstract

When employing a multi-purpose OS(Operating System) for an embedded system, its size reduction is one of inevitable problems This paper describes the size reduction of a multi-purpose OS for application-specific system by deleting the unused code of the OS for the application programs. We study two cases where unit of deletion is statement of program language, or procedure.

We try how much size reduction can be performed by using an existing OS. The results show 10% 〜 30% of it is reduced under the condition where only the 'init' program is activated.

1 はじめに

組み込みシステムの機能が複雑化し、製品サイクルの短期化によって組み込みシステム開発におけるソフトウェア開発工数の短縮が重要な課題となってきている。さらに家電や FA(Factory automation) 機器の世界でも ネットワークや GUI(Graphical User Interface) への対応が求められるようになり、OS(Operating System) を利用して開発時間の短縮を図るケースも増えてきた。

システムの複雑化に伴って、メモリ空間を分割しての保護の必要性が増え、PC や WS での開発環境が利用できる利点もあって、汎用 OS をプラットホームとして開発される組み込みシステムが増えている。

ところで組み込みシステムでは、一旦完成してしまうと、その後ソフトウェアを入れ替えたり、ハードウェアの構成を変更することはほとんどない。そのような利用形態では、利用する OS の機能は限定できる [1]。組み込みシステム向けに作られている OS は、アプリケーションに合わせて細かなカスタマイズを行うことで、必要最低限の資源で OS が動作するように作られている。

汎用 OS では、サイズが大きいだけでなく、カスタマイズの粒度が粗い場合が多い。ほとんどの汎用 OS において、API(Application Program Interface) セットは固定的でカスタマイズができない。また、上位互換性のために、通常は利用されない API が実装されていることもある。

通常カーネルとユーザプログラムは別のメモリ空間で動作しているので、OS のサービスを呼び出す方法はソフトウェア割り込みやメッセージパッシングを利用したシステムコールの形式になっている。そのため関数呼び出しで OS のサービスを提供している組み込みシステム向け OS で利用されている、ライブラリリンクを利用して必要最低限の関数群で OS を構成するという手法は使えない。

本研究では、このような問題に対応するために、あるユーザプログラムに特化したコンパクトな OS を自動的に作る手法を考える。

2 汎用 OS の組み込みシステムへの適用

「組み込みシステム」を、「内部に組み込まれたマイクロプロセッサとソフトウェアの組み合わせで機器の制御を行い、それをコンピュータと認識されないようなシステム」の総称とする [2]。具体的には、テレビ、ビデオ、エアコン、炊飯器、自動車のような家庭用のものから、工業用ロボットまで多種多様な分野のものがある。

汎用 OS を組み込みシステムに採用する要因としては、以下のものがあげられる [3, 4]。

- 組み込みシステム向け OS に比べ、安価である。特にオープンソースのものは製品に使用した場合に製品一つごとのライセンス料がかからない。
- 開発・テスト環境が充実している。
- PC の普及により、PC 上での開発に慣れた技術者が増えた。
- 標準的なハードウェア部品を使用することでコストを安く抑えるために、PC を利用することが増えた。
- 汎用 OS には相互運用について実績のあるネットワークスタックなどがある。
- サードパーティ性のソフトウェア資産が多い。
- ユーザプログラムはカーネルと異なるメモリ空間で実行されているので、ユーザプログラムの不具合がシステム全体に影響することは少ない。

汎用 OS を組み込みシステムに適用する場合、以下のような欠点があげられる。

- 使用する資源 (メモリなど) の量が大きい。

- リアルタイム性、割り込み応答性をあまり考慮していない。
- 組み込み向け OS に比べてアプリケーションにあわせた細かなカスタマイズができない。多くの場合、カーネルの API セットを替えることは難しい。
- OS を経由せずにユーザレベルで直接ハードウェアにアクセスする手段を提供していないことが多い。

3 研究の目的

汎用 OS を組み込みシステムに利用するにあたっての欠点のうちいくつかは、OS の構成をアプリケーションに特化したものに細かくカスタマイズできれば解決すると考えられる。本研究ではこれらの欠点のうち OS の消費する資源の量が大きい点に対応し、OS の半自動的なカスタマイズによって OS のサイズ縮小を狙ったシステムについて考察する。

このシステムのユーザには組み込みシステムの開発者を想定している。目標としては、以下のものがあげられる。

1. コードの部分的な削除による ROM 使用量の削減
   OS のコードのほとんどは、アプリケーションからのシステムコール呼び出しによって実行される。アプリケーションが利用するシステムコールと引数を解析し、実際には利用されない部分を発見して削除することで、OS の無駄なコードを記憶するために使用される ROM の量を削減する。

2. ローカル変数の削除によるスタック使用量の削減
   ローカル変数の削除により、OS が使用するスタックの総量が減る可能性がある。スタック使用量の最大値を減らすことができれば、スタックに割り当てる領域を小さくして RAM の使用量を削減できる。

3. グローバル変数の削除による RAM 使用量の削減
   グローバル変数の削除により、OS が使用する RAM の量が減らせる。

これらをふまえて、OS の半自動的なカスタマイズを行うことを考える。

4 サイズ縮小手法

ここでは、不要なコードを抽出するための手法について、コード削除の単位を基準にして考える。

4.1 文を単位としたサイズ縮小

まず、文を単位としたサイズ縮小について考える。プログラムスライシングは、プログラム内の命令の間の依存関係を明らかにする技術である。1982 年からソフトウェア工学の分野で研究されており、テスト、デバッグ、保守などの広範囲に応用されている [5]。

プログラムスライシングでは、データ依存関係と、制御依存関係を利用して、プログラム依存関係グラフを構築している [6]。

静的にプログラムを解析することにより、プログラム内のある命令の実行に影響を与える可能性のあるすべての命令を抽出したり、ある命令を変更した場合に影響を受ける可能性がある命令の集合を求めることができる。

また、プログラムを動的に解析して、プログラム内のある命令の実行に影響を与えた可能性のあるすべての命令を抽出することができる。

プログラムスライシングの特徴として、スライシングによって求められたスライスは、ある基準 (特定の変数や入出力) について、元のプログラムと等価であり、実行可能である点があげられる [7]。

Thomas Reps と Todd Turnidge はプログラムスライシングの応用のひとつとしてソフトウェアの専用化を提案している [8]。その中で例として UNIX の Word Counter を模した行数、文字

数をカウントするプログラム wc から、行数のみを出力するプログラムを生成している。これは行数を表示する命令について、後方スライスをとることで、行数を表示するのに必要な部分のみを切り出している。

プログラムスライシング技術の応用である、プログラムの専用化を、OS に適用することを考えると、以下のような問題がある。

- C 言語とアセンブリ言語の混在
  OS はハードウェアを直接制御するので、C 言語だけでは書けず、アセンブリ言語が必要になる。現在あるプログラムスライシングを行うソフトウェアでアセンブリ言語に対応しているものはみつからなかった。

- 意味のある busy loop など、タイミングの問題
  OS 内では、busy loop でタイミングをとることもある。しかしプログラムスライシングでは、出力に影響しない busy loop は不要な部分とみなされる。

- 実行されないはずだが必要な処理
  一部のエラー処理ルーチンなど、到達しないはずの文でも、残しておきたいことがあるが、ソフトウェアスライシングでは対応しない。

- 複数の実行開始位置
  OS 中の実行開始位置としては、起動時、API の入り口、割り込みハンドラ、メッセージを受ける関数があり、ソフトウェアスライシングを単純には適用できない。

- 変数操作に見えるメモリマップド I/O
  メモリマップド I/O は、C 言語レベルで見ると変数の操作に見えるが実際は入出力である。これを変数の操作として扱うと、依存関係が正しく解析できない。

- 関数ポインタ配列による分岐など、うまく適用できない場合もある。
  関数ポインタ配列による分岐を考えると、動的スライシングによって配列中のどの関数ポインタが利用されたかは判るが、利用されない部分をダミーのポインタに置き替えるなどの処理をしないと、呼ばれていない関数についても、リンク時にシンボル参照があるのでリンクしようとしてしまう。

これらの問題の多くに関しては、人手により補助的な情報を与えることが必要になる。たとえば、ある関数の内部ではコードの変更を行わない、といった指定を行う。

4.2 関数を単位としたサイズ縮小

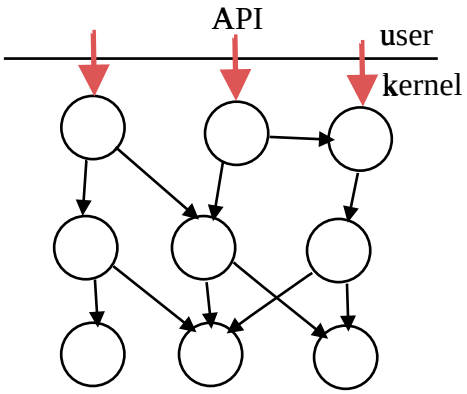

図 1: 関数の呼び出し関係グラフ

関数を単位としたサイズ縮小を考える。

カーネル内でグローバルなスコープを持つ変数がある場合など、呼び出されることのない関数に対してデータ依存が存在していても、実際には実行されないので、関数の呼び出し関係だけを追跡すればよい。

関数を単位としたサイズ縮小では、以下の操作を行う。

1. 図 1 のように関数の依存関係の有向グラフを作る。

2. 各 API に対応する OS 内の関数を調べて対応表を用意する。

3. ユーザアプリケーションが利用する API を列挙する。

4. 上の API に対応する関数を対応表を引いて見つける。
5. 上の関数一覧に、割り込み処理、起動時など、動作するために必須の関数を加える。
6. グラフを操作し、上の一覧から到達不可能な関数を不要な関数として検出する。
7. 不要な関数を削除する。

関数の呼び出し関係グラフを利用した場合の問題点としては、以下のものがある。

- 関数の依存関係グラフを作るのが難しい場合がある (関数ポインタ経由の呼び出しなど)。第 5 章で述べる試行では、内部の解析が必要になる部分を避けて、API と対応する関数の表を利用した。
- 文を単位としたものより粒度が大きく、変数の値も考えていないので削ることのできる部分が少ない。
- 関数のみ削除しても呼び出し部分が残っていればリンクエラーになる。

関数ポインタ経由の呼び出しに関しては、関数ポインタについてデータ依存を追跡して取りうる値を列挙するなどの方法が考えられる。実行できることを保証しつつどこまで不要なものを削れるかは今後の課題である。

リンクエラーの問題については、リンカの名前解決について、ダミーのアドレスを割り当てることで対応する。このこともあわせて、関数を単位とした削除については、リンカステージでの最適化として実装するのが適当だと考えている。

5 MINIX を使った試行

5.1 サイズ縮小に際して

関数ポインタを利用した関数呼び出しがある場合、適切な関数の呼び出し関係のグラフを自動的に作ることは難しい。

MINIX の場合、デバイスドライバを呼び出す時と、システムコールのメッセージを受けた直後の分岐で利用されているが、これを正しく解釈するには、関数ポインタについてのデータ依存を考えて、とりうる値を列挙するしかない。関数ポインタのとりうる値が全て関数のシンボルで表現されていればよいが、アドレスを即値で指定されている場合には、実行時のアドレス情報が必要になる。

今回の試行ではデバイスドライバ部分に関しては評価せず、システムコールについても、対応する関数の拾い出しは手動で行った。

5.2 試行

本手法が有効であることを確認するために、実際の OS のソースコードを利用して、実際にどの程度のコードが利用されるのかを調査した。

MINIX は、PC 上で動作するマイクロカーネル構成でデザインされた、UNIX clone である [9]。本評価には最新版である 2.0.2 のソースコードを利用した。

MINIX 実装上の特徴としては、以下のものがあげられる。

- 規模が小さい。
- アセンブラによる記述が非常に少ない。
- マイクロカーネル構成になっており、中心的なモジュールは、ハードウェアの操作、スケジューリング、メッセージパッシングなどを行う kernel、シグナルやプロセス生成、メモリ割り当ての中心となるメモリマネージャ (MM)、ファイルシステム、端末ドライバなどを提供するファイルシステム (FS) の 3 つからなる。
- MM と FS には、それぞれメッセージを受ける個所が一つあり、メッセージに対応した関数を、関数ポインタの配列を引いて呼びだす。

表 1: init が必要とするシステムコール

_exit	errno	lseek	reboot	time
alarm	execve	memset	setsid	wait
close	fcntl	open	sigaction	waitpid
dup	fork	pipe	sigemptyset	write
dup2	fstat	read	sleep	

表 2: 各モジュール別の計測結果 (単位は行数)

モジュール	全体	最小限	init
カーネル (デバイスドライバ以外)	3150	1031	2228
(デバイスドライバ)	5425	—	—
メモリマネージャ (MM)	1788	296	1611
ファイルサーバ (FS)	4437	1389	3536

- 各システムコールに対応する関数は、do_* のような命名をされている。

今回は規模の小さいことと、アセンブラによる記述の少ないことから、MINIX を対象とした。実際のプログラムを使った例題として init を使い、init を MINIX 上で実行する時に呼ばれうる関数を全て実装した場合、どの程度のコード量になるかを計測した。

init から呼ばれるシステムコールは表 1 の通りである。API と関数の対応付けは手動で行い、関数単位の依存関係グラフの作成、追跡は専用のプログラムを作った。

その結果は表 2 の通りである。カーネルのデバイスドライバについては考慮していない。カーネルの 70.7%、メモリマネージャの 90.1% ファイルサーバの 79.7% が必要とされた。

表 2 に、各モジュールのコード全体の行数と、そのうち最小限必要な行数を示す。

6 関連研究

C 言語で記述されたソフトウェアの解析と専用化については、Lars Ole Andersen によって、部分評価を用いてソフトウェアの専用化を行う CMIX というツールが開発されている [10]。

組み込みシステム用に作られた OS のほとんどは、アプリケーションに特化した機能のみを提供するようにカスタマイズが可能である。カスタマイズの手法としては、以下のようなものがある。

- モジュールに分割して実装し、構成時または実行時にモジュール毎の追加・削除ができるようにする。
  QNX*、OS-9† など

- OS のサービスをライブラリの形で提供し、ユーザのプログラムをリンクする段階で、必要なものだけがリンクされることで、自動的にカスタマイズする。

*QNX Software Systems Ltd. http://www.qnx.com/
†Microware Systems Corporation Homepage http://www.microware.com/

VxWorks[‡]、ITRON 準拠 OS の一部など

- `#define` 文などをソースに埋め込み、プリプロセッサを利用してソースの切り貼りを行う。
  eCos [§]、Linux など

- ソースコードを提供してユーザが書き替えられる。
  eCos など

7 おわりに

本研究では提供するサービスのカスタマイズが細かく行えない汎用 OS に対して、特定用途に特化させるカスタマイズを可能にすることでサイズ縮小をはかるシステムについて考察した。

具体的にはアプリケーションから利用される API を調べ、OS の内実行されない部分を抽出し、削除すること OS 中の不要な命令の一部を削除することができる。

文を単位とする手法と、関数を単位とする手法について考察した。文を単位とする手法の方が詳細な結果を望めるが、単純にソフトウェアスライシングを適用することは難しく、かなりの手間がかかる。関数を単位とする手法では、API と OS 内の関数との対応表を用意することで、その他の処理はほとんど自動化できる。関数ポインタ経由の呼び出しについては実行時の変数の値を考慮した処理が必要となる。

MINIX と、その上で動作する init を例題として、サイズがどの程度縮小できるかについて、試行を行った。結果として、10%〜30%程度の縮小ができそうだと認められた。

これからの課題としては、以下のものがある。

- 関数単位の OS サイズ縮小システムの実装
- 文単位のサイズ縮小と関数単位のサイズ縮小の併用手法についての考察
- 複数言語 (C と asm など) の混在への対応
- ソフトウェアスライシング以外の手法

[‡]Wind River 社 `http://www.wrs.com/`

[§]`http://www.redhat.com/embedded/technologies/ecos/`

謝辞

本研究の一部は科学技術振興事業団戦略的基礎研究推進事業 (CREST) の支援のもとに行われたものである。

参考文献

[1] 森若和雄, 久住憲嗣, 中西恒夫, 片山徹郎, 最所圭三, 福田晃: 電車模型制御用ソフトウェアシステムの設計, 情報処理学会研究報告, 2000-OS-84, pp.55–61 (2000).

[2] 坂村健 監修: μITRON3.0 標準ハンドブック, パーソナルメディア (1997).

[3] 中本幸一, 高田広章, 田丸喜一郎: 組込みシステム開発の現状と動向, 情報処理, Vol. 38, No. 10, pp.871–878 (1997).

[4] 堀内岳人, 加納護: 組み込み OS としての BSD ,BSD マガジン, Vol.6, pp.17–23 (2000).

[5] M. Weiser: Program slicing, Proc. of the Fifth Int'l Conf. on Software Engineering, pp. 439–449 (1981).

[6] J. Bergeretti and B. Carr'e: Information-flow and data-flow analysis of while-programs, ACM Trans. on Programming Languages and Systems,Vol. 7, No. 1, pp. 37–61 (1985).

[7] 下村隆夫: プログラムスライシング技術と応用, 共立出版株式会社 (1995).

[8] T. Reps and T. Turnidge: Program specialization via program slicing, Proc. of the Dagstuhl Seminar on Partial Evaluation, pp.409–429 (1996).

[9] A. Tanenbaum: The MINIX Home Page (1996), `http://www.cs.vu.nl/~ast/minix.html`.

[10] L. O. Andersen: Program analysis and specialization for the C programming language, PhD thesis, DIKU,University of Copenhagen (1994).