アプリケーションに適応したRTOSの細粒度コンフィギュレーション手法

宮内 哲夫[1,a)]　田中 清史[1,b)]

概要：効率の良い組込みアプリケーション開発のためには，リアルタイムオペレーティングシステム（RTOS）の利用が有効である．RTOS は様々なアプリケーションに対応するための豊富なサービス群を含むが，アプリケーションが限定された場合は過剰な機能がフットプリントと実行効率に悪影響を及ぼすことになる．本稿では，アプリケーションのソースコードを解析することにより，使用されるサービスコールとその用途を特定し，コード断片レベルで最適化を行う手法を提案する．評価において，提案手法を適用することにより，サービスコールのバイナリサイズが平均 14.8%，実行命令数が平均 13.3 %削減された．

キーワード：RTOS，コンフィギュレーション，ITRON

Fine-Grained Configuration of RTOS Adapted to Applications

Tetsuo Miyauchi[1,a)]　Kiyofumi Tanaka[1,b)]

Abstract: Use of real-time operating systems (RTOS) is effective in efficiently developing embedded applications. RTOSs consist of rich services to support various embedded applications. However, excessive functions influence the footprint and execution efficiency when the application is fixed. In this paper, we propose a method of optimizing service calls at a code-fragment level by analyzing application source codes and identifying used service calls and their usage. In the evaluation, size of service calls and the number of instructions executed are decreased by 14.8% and 13.3%, respectively.

Keywords: RTOS, configuration, ITRON

1. はじめに

従来の組込みシステム開発の特徴としてリソース制約が厳しいことが挙げられる．モノのインターネット（Internet of Things, IoT）時代の到来とともに，リソース制約の厳しいデバイス（モノ）はより増加することが予想される．一方，組込みシステム開発において，開発効率の観点からリアルタイムオペレーティングシステム（RTOS）の利用が一般的になりつつある．RTOS を利用することで，ハードウェアの抽象化，カーネルオブジェクトを介する同期・通信機能，リアルタイムタスクスケジューリングなどが容易に実現される．

システムのビルドにより，使用される RTOS のサービス（サービスコール）は実行バイナリに含まれることになるが，そのバイナリサイズ削減と実行効率向上のために，各サービスコールは単機能であることが原則である．しかし，様々なアプリケーションに柔軟に対応するために，ルーチン内である程度の選択／分岐が含まれるサービスも存在する．

例えば，μITRON4.0 仕様 [1] に含まれるサービスコールの一つに snd_mbx がある．このサービスコールはメールボックスへメッセージを送信する機能を提供する．メールボックスの生成時に優先度付きメッセージを意味する属性 `TA_MPRI` が指定された場合は，送信時にメッセージに付加された優先度順にメッセージキューに入れられる．属性 `TA_MFIFO` が指定された場合は，送信順にキューに入れられる．商用の RTOS である VxWorks[2] においても，同様にメッセージ通信のためのメッセージキューに対して 2 つのオプション（`MSG_Q_FIFO`，`MSG_PRIORITY`）を指定可能となっている．

ここで，アプリケーションが限定された場合，選択肢が

[1] 北陸先端科学技術大学院大学
Asahidai 1-1, Nomi, Ishikawa, 923–1292, Japan
[a)] t-miyauc@jaist.ac.jp
[b)] kiyofumi@jaist.ac.jp

固定され，本来そのアプリケーションにとって実行されることのない不要なコード断片が存在することになる．例えばアプリケーションが TA_MFIFO のみを使用している場合は，snd_mbx に含まれる優先度順のメッセージ管理コードは不要である．実行オーバヘッドとフットプリントの減少のためには，このような不要コードは削除されることが望まれる．

不要コード断片の削除のために，コンパイル時に RTOS 部分に対して部分評価（partial evaluation）[3] による最適化を適用することも考えられるが，厳格に定義された（well-defined な）RTOS 仕様に基づくコードでは，より簡単にマクロ記述で不要コード削除に対応できることが期待できる．

我々は μITRON4.0 仕様のスタンダードプロファイルに準拠する 70 個のサービスコールから構成される RTOS を開発してきた [4]．本稿では，μITRON4.0 仕様の RTOS に対して不要コード断片を削除する手法を提案し，バイナリサイズと実行命令数に関して評価を行う．

2. 不要コード断片削除手法

本節において，アプリケーションソースコードを解析し，アプリケーションに適応して RTOS をコード断片レベル（細粒度）で最適化する手法を提案する．RTOS の各サービスコールのソースコードはアプリケーションの使用に応じてコード断片の選択を行うためにマクロ記述（“#ifdef”）を含む．このようなソースコードは “#ifdef hell”[5] と呼ばれる可読性の低さが問題点となることがあるが，RTOS のサービスコールはアプリケーションプログラムとは異なり，開発者にとっては定義された機能と API のみが重要であり，そのソースコードに対してアプリケーションプログラム程高い可読性が要求されることはない．したがってコード選択を容易に実現するためにマクロ記述を利用する．

以下に本研究において，アプリケーションを解析し，使用する機能のみを抽出して RTOS をコンフィギュレーションする手法について述べる．

本環境における RTOS のコンフィギュレーションのフローは図 1 のようになる．

システムコンフィギュレーションファイル解析では，システムコンフィギュレーションファイルに記述された静的 API を解析する．各サービスコールで使用されている ID 番号のリストの作成，データキュー，イベントフラグ，メールボックス，セマフォ，固定長メモリプールのタスクの待ちが FIFO 順か優先度順かのオプションのチェック，メールボックスの待ちが FIFO 順か優先度順かのオプションのチェック，イベントフラグのクリア条件の有無のチェックなどを行う．以下にその処理について述べる．μITRON4.0 ではシステムコンフィギュレーションファイルのビルド手順，静的 API の記述形式が定義されているため，定義された手順，記述形式に従った解析を行う．まず，システムコンフィギュレーションファイル内の INCLUDE 文で記述されたヘッダファイル中に記載されたマクロ定義を抽出するため，INCLUDE 文を#include に変換して C 言語プリプロセッサをかけ，マクロ定義を展開する．次に，μITRON4.0 の仕様に従い各静的 API の記述の抽出を行い，各 API 毎に定義されたサービスコールの属性を抽出する．サービスコールにより，複数の属性が指定できるものがあるため複数の属性についてはそれぞれ別々に抽出する．例えば，イベントフラグにおいては，TA_TFIFO または TA_TPRI のいづれか一方, TA_WSGL または TA_WMUL のいづれか一方または TA_CLR を指定することができる．これらの属性が指定されているかを別々に抽出し，その指定に応じて後述するマクロ定義を出力する．また，次のステップのアプリケーションプログラム解析において，指定された ID のチェック等のため静的 API で生成されたそれぞれの資源の数を生成資源数情報として次のステップに渡す．本処理の流れは図 2 のようになる．

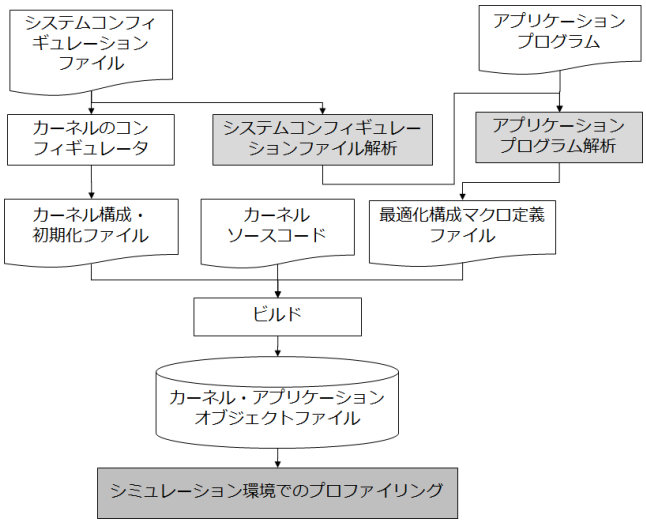

図 1　処理フロー

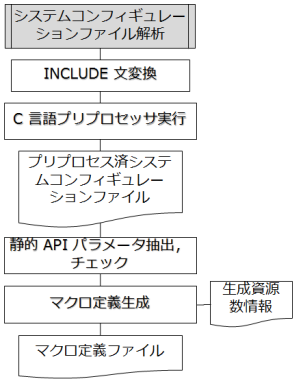

図 2　システムコンフィギュレーションファイル解析

アプリケーションプログラム解析では，C 言語で記述されたアプリケーションプログラムを解析して，μITRON4.0

のサービスコールを呼び出している箇所を抽出し，サービスコールのパラメータを解析し，システムコンフィギュレーションファイル解析の結果と合わせて，最適化構成マクロ定義ファイルを生成する．以下にその処理について述べる．まず，アプリケーションプログラムに記述されたヘッダファイルの展開，マクロ定義の展開を行うため，アプリケーションプログラムのソースファイルに対し C 言語のプリプロセッサを適用する．次に，各タスクに対して使用されているすべてのサービスコールを検索し，そこで利用されているパラメータを抽出する．抽出された各パラメータに対してエラー要因のチェックを行い，チェック結果に応じてマクロ定義を出力する．エラー要因のチェックにあたっては，ヘッダファイルに記述されたマクロ定義された定数においては，C 言語のプリプロセッサの適用により数値に展開されているため，その数値によりパラメータのチェックを行う．パラメータが変数で渡されており数値での判定ができない場合にはそのパラメータはカーネルでのチェックが必要と判断しパラメータのチェックを行うためのマクロ定義を行う．また、システムコンフィギュレーションファイル解析から渡された生成資源数情報と整合するかをチェックする．これらの処理の詳細については，呼び出し方法に起因する不要コードの削除として後節で述べる．本処理の流れは図 3 のようになる．

上記の一連の処理を Python で記述したプログラムで実行している．

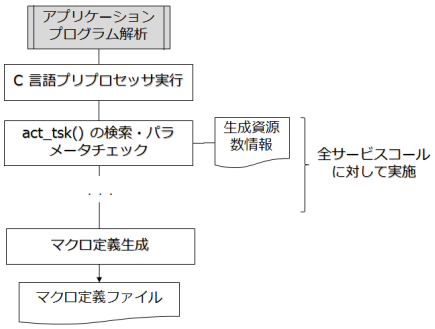

図 3 アプリケーションプログラム解析

アプリケーションプログラムのオブジェクトの構成にあたっては，生成された最適化構成マクロ定義ファイルと，μITRON4.0 仕様で規定されたカーネルのコンフィギュレータにより出力されたカーネル構成・初期化ファイル，およびカーネルのソースコードをコンパイル環境を用いてビルドする．現在の環境では ARM v7-A[6] を対象とした gcc を用いてオブジェクトコードを生成している．

生成されたオブジェクトコードの評価のため，ARM v7-A[6] 命令セットからなるオブジェクトを実行するシミュレータを作成した．このシミュレータでは，アプリケーション全体の実行に加え，μITRON4.0 のサービスコールの呼び出しのプロファイリングを行うことができ，各サービスコールの呼び出し回数，サービスコール毎の実行命令数を表示することができる．

2.1 属性固定に起因する不要コードの削除

μITRON4.0 のサービスコールには，カーネルオブジェクトの生成時に指定される属性によってカーネルのコード内で使用される機能が限定され，不要コード断片を削減できる場合がある．以下に wai_sem サービスコールを例にして説明する．

wai_sem サービスコールの属性は，システムコンフィギュレーションファイル内に記述された静的 API CRE_SEM 内のパラメータで指定される．セマフォの待ち順には，FIFO 順の場合と優先度順の場合があり，静的 API の解析によりセマフォの属性に，セマフォの待ち順として FIFO 順 (TA_TFIFO) が指定されていることがわかった場合，次のようなマクロ定義を出力する．

マクロ定義

```
#define CHK_SEM_FIFO
```

カーネルのソースコード内にはあらかじめ，以下のコード抜粋のように，待ちキューへの挿入箇所に対して，優先度順と FIFO 順の両方が使用される場合，優先度順のみが使用される場合，FIFO 順のみが使用される場合それぞれに応じて，マクロに対するディレクティブを記述しておく．

wai_sem のコード抜粋

```
#if defined(CHK_SEM_PRI) &&
                        defined(CHK_SEM_FIFO)
    if((semcb -> sematr & TA_TPRI) != FALSE){
        /* based on task priority */
        _kernel_queue_insert_tpri ( ... );
    } else {
        /* based on FIFO */
        ...
    }
#else /* CHK_SEM_PRI_FIFO */
#ifdef CHK_SEM_PRI
    _kernel_queue_insert_tpri ( ... );
    ...
#endif /* CHK_SEM_PRI */
#ifdef CHK_SEM_FIFO
    _kernel_queue_insert_prev ( ... );
    ...
#endif /* CHK_SEM_FIFO */
#endif
```

優先度順と FIFO 順の両方が使用される場合には，実行時に属性をチェックして一方を選択するようにしなけれ

ばならないが，どちらか片方のみの場合には，マクロ定義された方の処理のみを行えばよい．例えば，前述のマクロ定義のように CHK_SEM_FIFO のみが指定された場合には，#ifdef CHK_SEM_FIFO 以下のコード片のみがコンパイル時に有効になり，優先度順の処理コードを判断するコード部分は出力されない．これにより，コードサイズ，処理速度の削減を図ることができる．

2.2 呼出し方法に起因する不要コードの削除

我々が開発した RTOS は μITRON4.0 のスタンダードプロファイルに準拠した 70 個のサービスコールがあるが，各サービスコール内にはそれぞれエラーチェックを行うコードが含まれている．μITRON4.0 の仕様ではメインエラーコードのエラークラス毎にエラーの検出を省略することができるが，エラーの検出を省略することで本来検出すべきエラーが検出できず，予期しない不具合となる場合がある．

我々の提案する手法では，システムコンフィギュレーションファイルおよび，アプリケーションプログラムの解析により，実行時にチェックする必要のないエラーをサービスコール毎に特定し，それらのコード断片をサービスコールのコードから削除し，必要なエラーチェックのみを行うコードを生成する．

それぞれのサービスコールに対しては通常ひとつまたは複数のエラーコードとその要因がある．以下に例としてセマフォの場合のサービスコールのチェック例を示す．表 1 から表 5 において，チェック内容が “–” となっているエラーは，アプリケーションプログラム実行時に動的に状況が変化するため静的なチェックができないが，それ以外のエラーについては，チェック内容に記載されたような確認をアプリケーションプログラム解析部で行い，その結果を最適化構成マクロ定義ファイルとして出力する．

表 1 サービスコールとエラー要因 (sig_sem)

サービスコール	エラー	意味	チェック内容
sig_sem	E_CTX	CPU ロック状態 or 非タスク状態からの呼び出し	–
	E_ID	セマフォ ID の範囲不正	セマフォ ID の範囲をチェック
	E_NOEXS	セマフォ ID が未登録	CRE_SEM で指定された ID と一致するかをチェック
	E_QOVR	最大資源数を超えた返却	–

例えば sig_sem サービスコールの呼び出しにおいて，与えられる引数の値を静的にチェックし，正しい範囲（0 以上，TMAX_SEMID 以下）であると判断される場合には，E_ID のためのチェックコードは不必要となる．一方，引数として変数が使用されていて値が判断できない場合などは，以下のマクロ定義を出力する．

表 2 サービスコールとエラー要因 (isig_sem)

isig_sem	E_CTX	CPU ロック状態 or タスク状態からの呼び出し	–
	E_ID	セマフォ ID の範囲不正	セマフォ ID の範囲をチェック
	E_NOEXS	セマフォ ID が未登録	CRE_SEM で指定された ID と一致するかをチェック
	E_QOVR	最大資源数を超えた返却	–

表 3 サービスコールとエラー要因 (wai_sem)

wai_sem	E_CTX	CPU ロック状態 or 非タスク状態からの呼び出し	–
	E_ID	セマフォ ID の範囲不正	セマフォ ID の範囲をチェック
	E_NOEXS	セマフォ ID が未登録	CRE_SEM で指定された ID と一致するかをチェック
	E_RLWAI	待ち状態の間に rel_wai 受付	rel_wai があるかをチェック †

表 4 サービスコールとエラー要因 (pol_sem)

pol_sem	E_CTX	CPU ロック状態 or 非タスク状態からの呼び出し	–
	E_ID	セマフォ ID の範囲不正	セマフォ ID の範囲をチェック
	E_NOEXS	セマフォ ID が未登録	CRE_SEM で指定された ID と一致するかをチェック
	E_TMOUT	ポーリング失敗	–

表 5 サービスコールとエラー要因 (twai_sem)

twai_sem	E_CTX	CPU ロック状態 or 非タスク状態からの呼び出し	–
	E_ID	セマフォ ID の範囲不正	セマフォ ID の範囲をチェック
	E_NOEXS	セマフォ ID が未登録	CRE_SEM で指定された ID と一致するかをチェック
	E_TMOUT	タイムアウト	–
	E_PAR	tmout が不正	tmout が -1 以外の負の場合をチェック
	E_RLWAI	待ち状態の間に rel_wai 受付	rel_wai があるかをチェック †

† チェックは行っているが，カーネルの実装上 wai_sem, twai_sem のコードに差分なし．

マクロ定義

```
#define CHK_SIG_SEM_E_ID
```

上記のようなアプリケーションプログラムの静的チェックの結果，最適化構成マクロ定義ファイルが出力される．出力されたマクロ定義に対応したエラーチェックのみが行われるように，サービスコールのソースコード内にあらかじめ以下のコード抜粋のようなディレクティブを記述しておく．

これらの定義が出力された最適化構成マクロ定義ファイルとカーネルのソースコードを合わせてビルドすることにより，アプリケーションプログラムに最適なオブジェクトコードを得ることができる．

sig_sem のコード抜粋

```
#ifdef CHK_SIG_SEM_E_CTX
    if(_KERNEL_SNS_LOC()||_KERNEL_SNS_CTX()){
    /* CPU ロック状態での呼出し or 非タスクコンテキスト */
...
        return E_CTX;
    }
#endif /* CHK_SIG_SEM_E_CTX */
...
#ifdef CHK_SIG_SEM_E_ID
    /* ID が範囲外 */
    if(semid < 0 || semid >= TMAX_SEMID){
...
        return E_ID;
    }
#endif /* CHK_SIG_SEM_E_ID */
...
#ifdef CHK_SIG_SEM_E_NOEXS
    /* ID が存在しない */
    if(_kernel_semid_tab[semid] != TRUE){
...
        return E_NOEXS;
    }
#endif /* CHK_SIG_SEM_E_NOEXS */
...
#ifdef CHK_SIG_SEM_E_QOVR
    /* キューオーバー */
    if(semcb -> semcnt > semcb -> maxsem){
...
    return E_QOVR;
    }
#endif /* CHK_SIG_SEM_E_QOVR */
```

3. 評価

各種タスク管理機能および通信機能等を使用する評価用テストアプリケーションと，μITRON4.0 の TOPPERS カーネルテストスイート [8] に対して，本環境を適用した．評価用テストアプリケーションは 9 個のタスク, 起動周期の異なる 3 個の周期ハンドラから成り，それぞれのタスク間で，固定長メモリプール，メールボックス，セマフォ，イベントフラグ，データキューによる同期・通信を行うアプリケーションプログラムである．対象とするプロセッサは ARM Cortex-A9[7]（ARM v7-A 命令セット）とした．バイナリサイズの評価では，コンパイル（gcc v4.9.3，“-O2” 指定）後の各サービスコールのバイナリコードからテキストセクションのサイズを取得してバイナリサイズとした．実行命令数の評価では，対象アプリケーションのシミュレータでの実行により実行された，各サービスコールの発行回数，アプリケーションプログラムに対する適応化前のサービスコール内の総実行命令数，本環境による適応化後のサービスコール内の総実行命令数を取得した．表 6〜表 16 に，評価用テストアプリケーションの結果と，カーネルテストスイートからの 10 個のアプリケーションの結果を示す．

また，表 6 における wai_sem サービスコールにおいて，属性固定に起因する不要コードの削減と呼び出し方法に起因する不要コードの削減によるそれぞれの内訳を表 17 示す．

3.1 バイナリサイズ

表 6 のアプリケーションプログラムでは，アプリケーションによる適応化により生成されたオブジェクトコードのプログラムサイズは 1980 バイト削減されている．実際に利用されている 34 種類のサービスコールのうち，適応化により, 29 種類のサービスコールにおいて平均 68 バイトのコードサイズの削減が確認できた．

表 7 から表 16 はカーネルテストスイートのプログラムに対する適応化の結果である．

表 7 のアプリケーションプログラムでは，アプリケーションによる適応化により生成されたオブジェクトコードのプログラムサイズは 708 バイト削減されている．実際に利用されている 13 種類のサービスコールのうち，適応化により, 11 種類のサービスコールにおいて平均 64 バイトのコードサイズの削減が確認できた．

同様に表 8 のプログラムにおいては，適応化により 324 バイト削減されており，実際に利用されている 7 種類のサービスコールのうち，適応化により, 5 種類のサービスコールにおいて平均 64 バイトのコードサイズの削減が確認できた．

表 9 および，表 10 はタスク関連のテストプログラムへの

表 6　評価用テストアプリケーション

SVC NAME	適応化前 (bytes)	適応化後 (bytes)	差 (bytes)	実行回数	適応化前総命令数	適応化後総命令数	差
cre_tsk	732	640	92	10	1384	1214	170
act_tsk	452	392	60	7	623	553	70
ext_tsk	316	316	0	2	116	116	0
chg_pri	748	688	60	3	194	170	24
get_pri	244	172	72	8	392	304	88
dly_tsk	180	180	0	1	43	43	0
slp_tsk	192	192	0	2	81	81	0
iwup_tsk	348	296	52	14	792	666	126
cre_sem	272	196	76	1	57	44	13
sig_sem	352	296	56	5	273	228	45
wai_sem	312	236	76	3	171	140	31
pol_sem	192	136	56	2	82	64	18
twai_sem	352	280	72	1	76	64	12
cre_flg	236	196	40	3	150	132	18
set_flg	444	388	56	5	397	352	45
clr_flg	184	128	56	1	39	30	9
wai_flg	440	384	56	5	383	343	40
pol_flg	264	204	60	1	47	36	11
twai_flg	484	428	56	1	100	91	9
cre_dtq	340	280	60	3	207	180	27
psnd_dtq	408	352	56	2	114	96	18
ipsnd_dtq	396	348	48	6	232	184	48
fsnd_dtq	460	384	76	1	59	47	12
rcv_dtq	684	632	52	1	61	53	8
trcv_dtq	720	672	48	1	80	72	8
cre_mbx	456	356	100	1	171	153	18
snd_mbx	468	324	144	5	325	240	85
rcv_mbx	444	332	112	5	447	388	59
cre_mpf	416	376	40	1	100	94	6
get_mpf	396	324	72	5	339	295	44
rel_mpf	384	292	92	5	292	207	85
get_tim	112	112	0	9	216	216	0
iget_tim	112	112	0	10	190	190	0
cre_cyc	368	284	84	3	243	198	45
合計	–	–	1980	–	–	–	1192

表 7　02spc01

SVC NAME	適応化前 (bytes)	適応化後 (bytes)	差 (bytes)	実行回数	適応化前総命令数	適応化後総命令数	差
cre_tsk	732	640	92	4	604	536	68
act_tsk	452	392	60	2	178	158	20
ext_tsk	316	316	0	3	174	174	0
chg_pri	748	688	60	1	66	58	8
slp_tsk	192	192	0	1	43	43	0
wup_tsk	388	328	60	1	61	51	10
iwup_tsk	348	296	52	10	630	540	90
sus_tsk	376	320	56	3	189	162	27
rsm_tsk	312	252	60	3	191	161	30
rel_wai	360	300	60	1	46	36	10
cre_sem	272	196	76	1	57	44	13
sig_sem	352	296	56	3	184	157	27
wai_sem	312	236	76	3	198	162	36
合計	–	–	708	–	–	–	339

表 8　02spc02

SVC NAME	適応化前 (bytes)	適応化後 (bytes)	差 (bytes)	実行回数	適応化前総命令数	適応化後総命令数	差
cre_tsk	732	640	92	6	864	762	102
act_tsk	452	392	60	4	356	316	40
ext_tsk	316	316	0	5	290	290	0
chg_pri	748	688	60	1	66	58	8
slp_tsk	192	192	0	2	86	86	0
wup_tsk	388	328	60	2	148	128	20
iwup_tsk	348	296	52	10	630	540	90
合計	–	–	324	–	–	–	260

表 9　03tsk01

SVC NAME	適応化前 (bytes)	適応化後 (bytes)	差 (bytes)	実行回数	適応化前総命令数	適応化後総命令数	差
cre_tsk	732	640	92	3	474	423	51
act_tsk	452	392	60	2	173	153	20
ext_tsk	316	316	0	2	116	116	0
ter_tsk	500	404	96	1	93	76	17
can_act	204	148	56	1	44	35	9
chg_pri	748	688	60	2	132	116	16
get_pri	244	172	72	1	49	38	11
can_wup	232	172	60	1	48	38	10
wup_tsk	388	328	60	1	53	43	10
iwup_tsk	348	296	52	10	630	540	90
sus_tsk	376	320	56	1	64	55	9
ras_tex	500	428	72	1	51	40	11
合計	–	–	736	–	–	–	254

表 10　03tsk07

SVC NAME	適応化前 (bytes)	適応化後 (bytes)	差 (bytes)	実行回数	適応化前総命令数	適応化後総命令数	差
cre_tsk	732	640	92	4	646	578	68
act_tsk	452	392	60	1	89	79	10
ext_tsk	316	316	0	3	174	174	0
chg_pri	748	708	40	8	363	331	32
get_pri	244	188	56	7	319	264	55
iwup_tsk	348	296	52	10	630	540	90
合計	–	–	300	–	–	–	255

表 11　06sem02

SVC NAME	適応化前 (bytes)	適応化後 (bytes)	差 (bytes)	実行回数	適応化前総命令数	適応化後総命令数	差
cre_tsk	732	640	92	5	734	649	85
act_tsk	452	392	60	3	267	237	30
ext_tsk	316	316	0	4	232	232	0
chg_pri	748	688	60	1	66	58	8
iwup_tsk	348	296	52	10	630	540	90
rel_wai	360	300	60	1	78	68	10
cre_sem	272	196	76	2	114	88	26
sig_sem	352	312	40	6	288	252	36
wai_sem	312	252	60	5	268	233	35
pol_sem	192	152	40	4	143	122	21
合計	–	–	540	–	–	–	341

適応化の結果である．表 9 においては，適応化により 736 バイト削減されており，実際に利用されている 12 種類のサービスコールのうち，適応化により, 11 種類のサービスコールにおいて平均 66 バイトのコードサイズの削減が確認できた．また，表 10 においては，適応化により 300 バイト削減されており，実際に利用されている 6 種類のサービスコールのうち，適応化により, 5 種類のサービスコールにおいて平均 60 バイトのコードサイズの削減が確認できた．

表 11 はセマフォ関連のテストプログラムへの適応化の結果である．適応化により 540 バイト削減されており，実際に利用されている 10 種類のサービスコールのうち，適

表 12 07flg02

SVC NAME	適応化前 (bytes)	適応化後 (bytes)	差 (bytes)	実行 回数	適応化前 総命令数	適応化後 総命令数	差
cre_tsk	732	640	92	3	474	423	51
act_tsk	452	392	60	1	89	79	10
ext_tsk	316	316	0	2	116	116	0
chg_pri	748	688	60	1	66	58	8
iwup_tsk	348	296	52	10	630	540	90
rel_wai	360	300	60	1	78	68	10
cre_flg	236	196	40	3	150	132	18
set_flg	444	404	40	9	513	457	56
clr_flg	184	144	40	3	97	84	13
wai_flg	440	404	36	11	670	615	55
pol_flg	264	220	44	12	539	476	63
合計	–	–	524	–	–	–	374

表 13 08dtq02

SVC NAME	適応化前 (bytes)	適応化後 (bytes)	差 (bytes)	実行 回数	適応化前 総命令数	適応化後 総命令数	差
cre_tsk	732	640	92	5	734	649	85
act_tsk	452	392	60	3	267	237	30
ext_tsk	316	316	0	4	232	232	0
chg_pri	748	688	60	2	137	121	16
slp_tsk	192	192	0	2	86	86	0
wup_tsk	388	328	60	2	148	128	20
iwup_tsk	348	296	52	10	630	540	90
rel_wai	360	300	60	1	78	68	10
cre_dtq	340	280	60	3	207	180	27
snd_dtq	532	496	36	14	947	869	78
rcv_dtq	684	632	52	11	858	777	81
合計	–	–	532	–	–	–	437

表 14 09mbx02

SVC NAME	適応化前 (bytes)	適応化後 (bytes)	差 (bytes)	実行 回数	適応化前 総命令数	適応化後 総命令数	差
cre_tsk	732	640	92	5	734	649	85
act_tsk	452	392	60	3	267	237	30
ext_tsk	316	316	0	4	232	232	0
chg_pri	748	688	60	1	66	58	8
iwup_tsk	348	296	52	10	630	540	90
rel_wai	360	300	60	1	78	68	10
cre_mbx	456	356	100	1	60	52	8
snd_mbx	468	316	152	8	417	349	68
rcv_mbx	444	276	168	5	299	252	47
prcv_mbx	320	160	160	6	249	200	49
合計	–	–	904	–	–	–	395

表 15 10mpf02

SVC NAME	適応化前 (bytes)	適応化後 (bytes)	差 (bytes)	実行 回数	適応化前 総命令数	適応化後 総命令数	差
cre_tsk	732	640	92	5	734	649	85
act_tsk	452	392	60	3	267	237	30
ext_tsk	316	316	0	4	232	232	0
chg_pri	748	688	60	1	66	58	8
iwup_tsk	348	296	52	10	630	540	90
rel_wai	360	300	60	1	78	68	10
cre_mpf	416	328	88	2	176	148	28
get_mpf	396	340	56	5	316	280	36
pget_mpf	268	228	40	3	118	104	14
rel_mpf	384	308	76	7	380	302	78
合計	–	–	584	–	–	–	379

表 16 13sys01

SVC NAME	適応化前 (bytes)	適応化後 (bytes)	差 (bytes)	実行 回数	適応化前 総命令数	適応化後 総命令数	差
cre_tsk	732	640	92	5	860	775	85
ext_tsk	316	316	0	3	174	174	0
chg_pri	748	688	60	4	264	232	32
dly_tsk	180	180	0	1	39	39	0
iwup_tsk	348	296	52	10	630	540	90
rot_rdq	196	176	20	7	297	287	10
get_tid	128	128	0	1	28	28	0
loc_cpu	208	208	0	2	100	100	0
unl_cpu	84	84	0	2	38	38	0
dis_dsp	104	104	0	2	48	48	0
ena_dsp	104	104	0	2	48	48	0
sns_dsp	28	28	0	4	28	28	0
sns_ctx	48	48	0	1	12	12	0
sns_loc	28	28	0	4	28	28	0
sns_dpn	88	88	0	3	59	59	0
合計	–	–	224	–	–	–	217

表 17 不要コードの内訳

SVC	属性固定に起因する不要コードの削減		呼び出し方法に起因する不要コードの削減	
	サイズ (bytes)	命令数	サイズ (bytes)	命令数
wai_sem	52	24	24	7

応化により, 9 種類のサービスコールにおいて平均 60 バイトのコードサイズの削減が確認できた.

表 12 はイベントフラグ関連のテストプログラムへの適応化の結果である．適応化により 524 バイト削減されており，実際に利用されている 11 種類のサービスコールのうち，適応化により, 10 種類のサービスコールにおいて平均 52 バイトのコードサイズの削減が確認できた.

表 13 はデータキュー関連のテストプログラムへの適応化の結果である．適応化により 532 バイト削減されており，実際に利用されている 11 種類のサービスコールのうち，適応化により, 9 種類のサービスコールにおいて平均 59 バイトのコードサイズの削減が確認できた.

表 14 はメールボックス関連のテストプログラムへの適応化の結果である．適応化により 904 バイト削減されており，実際に利用されている 10 種類のサービスコールのうち，適応化により, 9 種類のサービスコールにおいて平均 100 バイトのコードサイズの削減が確認できた.

表 15 はメモリプール関連のテストプログラムへの適応化の結果である．適応化により 584 バイト削減されており，実際に利用されている 10 種類のサービスコールのうち，適応化により, 9 種類のサービスコールにおいて平均 64 バイトのコードサイズの削減が確認できた.

表 16 はシステム状態管理関連のテストプログラムへの適応化の結果である．適応化により 224 バイト削減されており，実際に利用されている 15 種類のサービスコールのうち，適応化により, 4 種類のサービスコールにおいて平均 56 バイトのコードサイズの削減が確認できた.

これらの結果により，サービスコールのバイナリサイズ

は 1 サービスコールあたり平均で 66 バイト削減される．削減の対象となるサービスコールの元のサイズと比較すると平均で 14.8% の削減となっている．特に待ちキューにタスクの待ちとメッセージの待ちの両方の要因があるメールボックス関連での削減量が大きいことがわかる．また，表 17 に，表 6 における wai_sem サービスコールにおける削減サイズ 76bytes の内の，属性固定に起因する不要コードの削減量と呼び出し方法に起因する不要コードの削減量の内訳を示す．属性固定による削減量である 52bytes に対し，呼び出し方法に起因する削減量は 24bytes となっており，前者の削減効果が比較的大きな割合を占めることがわかる．

3.2 実行命令数

表 6 から表 16 の結果により，実行命令数の評価を行う．

表 6 のアプリケーションプログラムでは合計で 1192 命令削減され，1 サービスコールあたりで平均して 41 命令削減されている．同様に，表 7 のアプリケーションプログラムでは合計で 339 命令削減され，1 サービスコールあたりで平均して 30 命令削減, 表 8 では合計で 260 命令削減され，1 サービスコールあたりで平均して 52 命令削減, 表 9 では合計で 254 命令削減され，1 サービスコールあたりで平均して 23 命令削減, 表 10 では合計で 255 命令削減され，1 サービスコールあたりで平均して 51 命令削減, 表 11 では合計で 341 命令削減され，1 サービスコールあたりで平均して 37 命令削減, 表 12 では合計で 374 命令削減され，1 サービスコールあたりで平均して 37 命令削減, 表 13 では合計で 437 命令削減され，1 サービスコールあたりで平均して 48 命令削減, 表 14 では合計で 395 命令削減され，1 サービスコールあたりで平均して 43 命令削減, 表 15 では合計で 379 命令削減され，1 サービスコールあたりで平均して 42 命令削減, 表 16 では合計で 217 命令削減され，1 サービスコールあたりで平均して 54 命令削減されることがわかる．

これらの結果により，サービスコールの実行命令数は 1 サービスコールあたり平均で 40 命令削減され，削減の対象となるサービスコールの元の命令数と比較すると平均で 13.3 % の削減となっている．また，表 17 に，表 6 における wai_sem サービスコールにおける総実行命令数の削減数 31 命令の内の，属性固定に起因する不要コードの削減命令数と呼び出し方法に起因する不要コードの削減命令数の内訳を示す．バイナリサイズの評価と同様，属性固定による実行命令数削減効果が大きな割合となっている．

4. 関連研究

組込みシステム分野において，アプリケーションに合わせて RTOS をコンフィギュレーションする試みがなされてきた．商用の RTOS の例として，VxWorks[2] では，カーネルのコンフィギュレーションにより，小さなフットプリントとなるように RTOS コンポーネントを限定してバイナリを作成することができる．しかし，ユーザ／開発者が使用するプロファイルおよび付加コンポーネントを指定する必要がある．

研究分野における例として，PURE[9] は組込みオペレーティングシステムを提供するためのプラットフォームであり，アプリケーションの要求に合わせて最小構成の OS を提供することを目的としている．CORE と呼ばれるセットは最小機能を提供し，オブジェクト指向開発における継承技術により付加的なサービスを提供することになる．本手法は新たな機能を継承によって自由に付加することになるため，OS 機能の定義が曖昧であり，ITRON のような厳格に定義された (well-defined)OS に対しては適用が困難である．また，継承により，粒度の異なる，機能が類似したモジュール（クラス）が多数存在することになり，利用者はアプリ構築時に情報を整理して使い分ける必要がある．これに対し，我々の方法では，利用者は ITRON の API に従うのみであり，OS のカスタマイズは透過的に行われる．

EPOS[10], [11] は上記 PURE の拡張環境である．PURE において，多数のシステムオブジェクト（モジュール）から，アプリケーションのシナリオにあわせて，それらを組合せることにより OS を構築するが，EPOS はそのシナリオからの構築をサポートする “scenario adapters” を提供する．基本的に PURE のサポート環境であるため同じ特徴を持つが，コンフィギュレーションでは，割込みハンドラ，ハンドラ間通信機能，スレッド実行，スケジューリング機能などのように，OS としての基本機能をコンフィギュレーションの対象モジュールとしており，我々の対象（エラーチェックコードやサブ機能の選択）とは異なる．

文献 [12] では，ソースコードレベルでカスタマイズ可能なコンポーネントライブラリ DREAMS[13] を利用して，コンポーネント合成環境 TEReCS[14] によりアプリケーションに特化したシステムを生成（コンフィギュレーション）する方法を紹介している．サービスはコンポーネントを含むスケルトンベースで提供され，アプリケーションが使用しないコンポーネントを削除することにより最小構成の OS を生成する．しかし，コンフィギュレーションされるサービスは主に同期・通信機能を対象としており，我々の手法のような各サービスコール内の細かなコード断片の削除とは目的が異なる．また，アプリケーション内の同期・通信を表現するグラフを開発者が別途作成する必要があり，アプリケーションのソースコードのみによるコンフィギュレーションは実現していない．

我々の手法では，μITRON4.0 を対象としていることから，RTOS サービスの使用は自明であり，利用サービスのみ実行バイナリにリンクされ，サービス単位では最小構成なシステムが構築される．また，アプリケーションのソー

スコードとコンフィギュレーションファイルのみを入力とし，開発者に対して完全に透過的に最適化を行うことが特徴である．

5. おわりに

本稿では，μITRON4.0 のアプリケーションプログラムに対して，システムコンフィギュレーションファイルおよび C 言語で記述されたアプリケーションソースプログラムを解析して，使用する機能のみをコード断片レベルで抽出し, RTOS をコンフィギュレーションする手法を提案した．生成されたコードサイズの比較，および，実際に実行される命令数の比較を行った．比較にあたっては，ARM Cortex-A9 のシミュレーション環境を構築し，その環境上でアプリケーションプログラムを実行させ，実際に実行される命令数の比較を行った．本手法によりアプリケーションプログラムのバイナリのサイズおよび実行時間が短縮されることが確認できた．

謝辞 本研究の一部は JSPS 科研費 JP 15K00073 の助成を受けて行われた．

参考文献

[1] μITRON4.0 Specification Ver.4.00.00, ITRON Committee, TRON ASSOCIATION.
[2] WInd River VxWorks Platforms 6.9, "`http://www.windriver.com/products/product-notes/PN_VE_6_9_Platform_0311.pdf`," Wind River Systems, Inc.
[3] N. D. Jones, C. K. Gomard, and P. Sestoft, "Partial Evaluation and Automatic Program Generation," Prentice-Hall, Inc., 1993.
[4] K. Tanaka, "Real-Time Operating System Kernel for Multithreaded Processor," Proc. of Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA), pp.91–99, 2006.
[5] H. Spencer and G. Collyer, "#ifdef Considered Harmful, or Portability Experience With C News," Proc of USENIX Conf., pp.185–198, 1992.
[6] ARM., "ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition."
[7] ARM., "Cortex-A9 Technical Reference Manual." Revision:r2p2.
[8] TOPPERS カーネルテストスイート "`https://www.toppers.jp/testsuites.html`"
[9] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk, "Design Rationale of the PURE Object-Oriented Embedded Operating System," Proc. of Intl. Workshop on Distributed and Parallel Embedded Systems (DIPES), pp.231–240, 1998.
[10] A. A. Fröhlich and W. Schröder-Preikschat, "Tailor-made Operating Systems for Embedded Parallel Applications," Workshop held in conjunction with IPPS/SPDP'99, LNCS-1586, pp.1361–1373, 1999.
[11] A. A. Fröhlich and W. Schröder-Preikschat, "EPOS: an Object-Oriented Operating System," Proc. of 2nd ECOOP Workshop on Object-Orientation and Operating Systems, pp.38–43, 1999.
[12] C. Böke, M. Götz, T. Heimfarth, D. E. Kebbe, F. J. Rammig, and S. Rips, "(Re-) Configurable Real-Time Operating Systems and Their Applications," Proc. of Intl. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS), pp.148–155, 2003.
[13] C. Ditze, "A Customizable Library to support Software Synthesis for Embedded Applications and Micro-Kernel Systems," Proc. of ACM SIGOPS European Workshop on Support for Composing Distributed Applications, pp.88–95, 1998.
[14] C. Böke, "Software Synthesis of Real-Time Communication System Code for Distributed Embedded Applications," Proc. of IFIP Conf. on Parallel and Real-Time Systems, 1999.