一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
VLD2020-75,HWS2020-50(2021-03)

RTOS 利用システムのフルハードウェア化における サービス処理機能の集約

六車　伊織†　石浦菜岐佐†　安堂　拓也†　冨山　宏之††　神原　弘之†††

† 関西学院大学 理工学部　〒 669–1337 兵庫県三田市学園 2–1
†† 立命館大学 理工学部　〒 525–8577 滋賀県草津市野路東 1 丁目 1–1
††† 京都高度技術研究所　〒 600–8813 京都市下京区中堂寺南町 134 番地

あらまし　本稿では, RTOS 利用システムをフルハードウェア化する手法において, 回路規模削減と実行の高速化を目的とした回路構成法を提案する. 大迫らは, タスク/ハンドラ等のカーネルオブジェクトと RTOS カーネルの機能を全て 1 つのハードウェアに合成する手法を提案しているが, RTOS のサービス機能をタスク側にリンクして高位合成していたため, 同じサービス機能を実行するハードウェアが重複して生成されていた. 本稿では, カーネルオブジェクトの実行を制御するマネージャモジュールに RTOS のサービス機能を集約することにより, この重複を削除する. これを実現するために, カーネルオブジェクトからサービス処理を起動する方式と, 複数のサービス処理要求を調停する方式を新たに提案する. この方式では, 簡単なサービス処理は RTL で設計できるため, 処理の高速化も実現できる. TOPPERS/ASP3 カーネル付属のサンプル “sample1” を縮小したプログラムをフルハードウェア実装した結果, 大迫らの手法に比べて回路規模は大幅には削減できなかったが, ほとんどのサービスコールで処理に要する実行時間を 50% 以下に削減できた.

キーワード　リアルタイムシステム, RTOS, システム合成, ハードウェアアクセラレータ, TOPPERS/ASP3, 高位合成

Aggregating Service Functions in Full Hardware Implementation of RTOS-Based Systems

Iori MUGURUMA†, Nagisa ISHIURA†, Takuya ANDO†,
Hiroyuki TOMIYAMA††, and Hiroyuki KANBARA†††

† Kwansei Gakuin University,　Gakuen 2–1, Sanda, Hyogo, 669–1337, Japan
†† Ritsumeikan University, 1–1–1 Noji–Higashi Kusatsu, Shiga, 525–8577, Japan
††† ASTEM RI/KYOTO, 134 Chudoji Minamimachi, Shimogyo–ku, Kyoto, 600–8813, Japan

Abstract　This article presents a revised architecture for full-hardware implementation of RTOS-based systems. In the previous method by Oosako, where tasks and handlers along with kernel functions were all synthesized into a single hardware, multiple copies of the same hardware modules were generated because the bodies of the same service functions were linked with tasks and were synthesized into hardware. Our new architecture attempts to remove such duplication by migrating the bodies of the service functions from task modules to the manager module. For this purpose, a new mechanism for activating service functions from tasks and arbitrating multiple requests is proposed. This scheme enables design of service modules in RT level, which contributes to reduce execution time of the service functions. An experimental hardware design of “sample1” bundled with TOPPERS/ASP3 kernel shows that only small reduction in circuit size is achieved on this instance but the execution time is reduced by more than 50% for most of the service functions.

Key words　Real-Time Systems, RTOS, System Synthesis, Hardware Accelerator, TOPPERS/ASP3, High-Level Synthesis

1. はじめに

近年の情報通信技術の発展により, 新しい製品やサービスが次々に提供されるようになっており, これらのキーデバイスとなる組込みシステムには益々高い機能が要求されるようになっている. 特に, 車載機器, 無人飛行機, ロボットの制御には, 高い機能と同時に高い応答性が要求される.

このようなリアルタイムシステムの開発には, リアルタイム OS (RTOS) が用いられる. RTOS は, タスクやハンドラが制約時間内に実行を完了するようにシステムを実装するための機能を提供する. しかし, システムの高機能化が進むにつれ, RTOS を利用したシステムのリアルタイム性確保は難しくなってきている.

RTOS を用いたシステムの高速化手法としては, RTOS 機能の一部または全てをハードウェア実装する方法がある. 文献 [1], [2], [3] では, RTOS のスケジューラのハードウェア化による高速化を行っている. また, 文献 [4], [5], [6] では RTOS のほとんどの機能をハードウェア実装している. しかしこれらの手法では, タスクやハンドラはソフトウェアのままであり, CPU 待ちやコンテキストスイッチによる遅延が生ずる.

一方, 高位合成技術 [7] を利用してタスクやハンドラを専用ハードウェアに合成することによって応答性能の向上を図る手法がある. 指定したタスクをハードウェアに合成するためのシステムレベル設計手法 [8], [9] が提案されているが, これらの手法では, RTOS および一部のタスクはソフトウェアとして実行される. 文献 [10], [11] は割込みハンドラを含むシステム全体のハードウェア化を行っているが, 合成対象はベアメタルシステムに限られる.

この課題を解決する一手法として, 文献 [12] はタスクおよび RTOS のサービス機能全てをハードウェア化する手法を提案している. この手法は, RTOS 上で動作するプログラムを入力として, これを実行するプロセッサと機能等価なハードウェアを自動生成する. タスクはそれぞれ独立に動作するハードウェアに合成され, 全てのタスクは並列に実行できるため, タスクの切り替えオーバーヘッドが無い. TOPPERS/ASP3 および FreeRTOS を対象としたプロトタイプ実装 [13], [14] では, 高い応答性能を実現しているが, 回路規模が大きくなることが課題となっている.

本稿では, 文献 [12] のハードウェア構成において, タスクに分散している RTOS のサービス処理機能をタスク管理ハードウェア側によって集約することによって, 回路規模を削減する手法を提案する. このために, タスクからサービス処理を起動する方式や, 複数のサービス要求が同時に発生した場合の調停方式を新たに提案する. また, 基本的なサービス処理を RTL で実装することにより処理の高速化を図る.

2. RTOS を用いたシステムのフルハードウェア実装

文献 [12] では, RTOS のサービスコールを用いたプログラムを入力とし, これを実行するプロセッサと機能等価なハードウェアを合成する手法を提案している. その概念を図 1 に示す. 上方の TSK, CYC, ALM, INT はそれぞれのタスク, 周期ハンドラ, アラームハンドラ, 割込みハンドラ等のカーネルオブジェクトであり, RTOS の管理下で実行される. この手法では, 各タスク/ハンドラは高位合成によって独立したハードウェアモジュールに合成され, RTOS 機能はマネージャ (manager) というハードウェアが実現する.

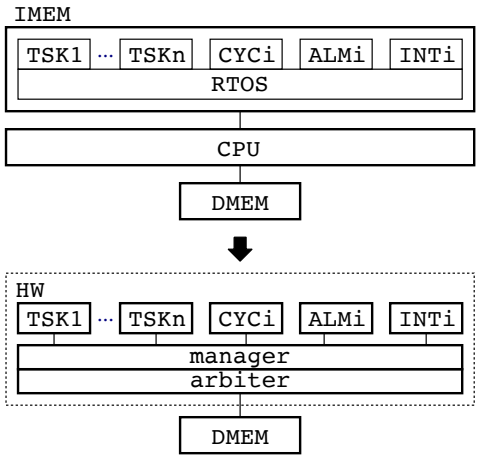


図 1: RTOS を用いたシステムのフルハードウェア実装 [12]

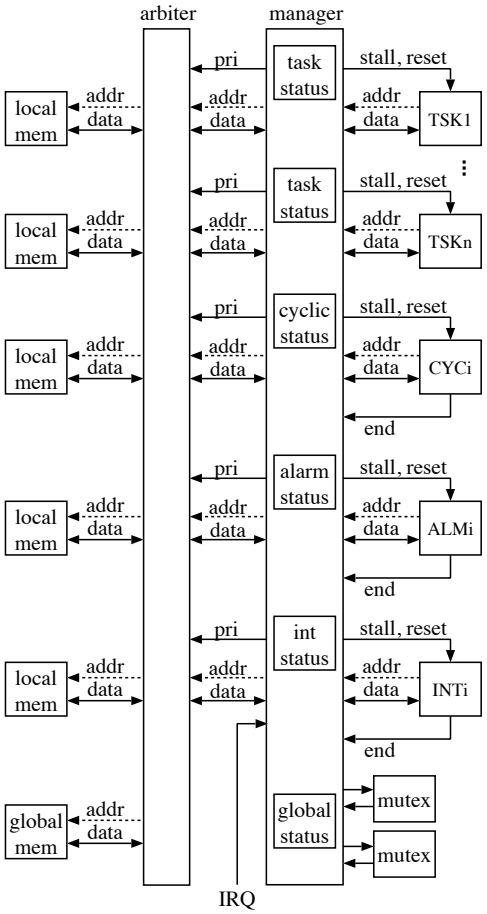
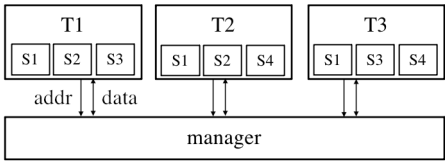
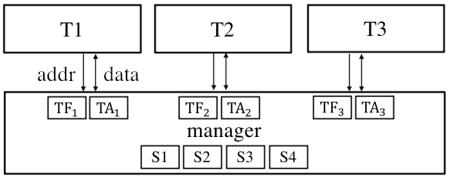


図 2: ハードウェアの構成 [12]

タスク/ハンドラは, 実行可能になれば全て並列に実行される. この制御はマネージャが各タスク/ハンドラの状態変数の値からタスク/ハンドラの実行/停止を制御する信号を生成することにより行う. データメモリ (DMEM) の同じバンクに同時にアクセスがあった場合には, 優先度に応じてアービタ (arbiter)



(a) 文献 [12]



(b) 本手法

図 3: 提案するハードウェア構成

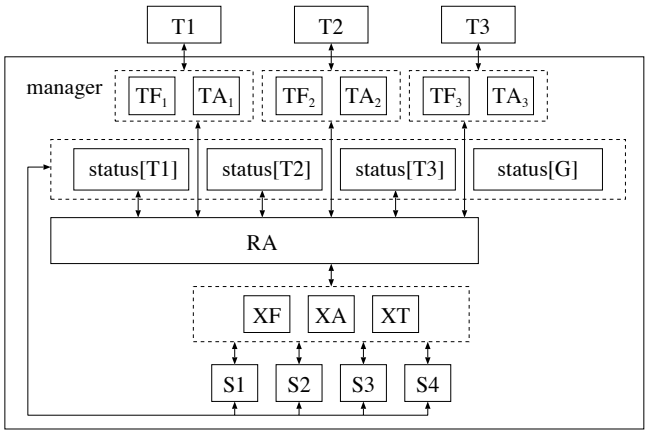


図 4: 合成するハードウェアモジュールの詳細構成

が調停を行う. この手法では, スケジューリングやコンテキストスイッチ, および CPU 待ちのオーバーヘッドがなくせる上, タスク/ハンドラをハードウェア化して高速実行できるため, システムの応答性能を飛躍的に向上させることができる.

合成されるハードウェアの構成を図 2 に示す. 各タスク/ハンドラの状態や優先度等の情報は, マネージャモジュール内の status レジスタに, カーネルの状態は global status レジスタに格納される. マネージャモジュールは, status レジスタの値に基づき, そのタスク/ハンドラが実行可能な場合には stall 信号を 0 に, 実行を停止させる場合には stall 信号を 1 にすることにより, タスク/ハンドラの実行を制御する. ロックの取得/解放などの処理は, ハードウェアにより実装されている. status レジスタと global status レジスタは, メモリ空間にマップされており, タスク/ハンドラからはメモリアクセスによって参照/更新できる. RTOS のサービスコールは基本的に status レジスタと global status レジスタおよび共有変数を参照/更新する処理として実装されているので, 各タスクとリンクする各タスク内にインライン展開することにより, タスクの一部として高位合成される.

なお, この手法では, 同時に実行できるサービスコールは一つに制限しており, サービスコールの入口と出口でロックを取得/解放することによりこれを実現している.

3. RTOS のサービス処理機能の集約

3.1 概　　要

本稿では文献 [12] の手法において, タスクに分散している RTOS のサービス処理機能をマネージャ側に集約することによって回路規模の削減を目指すとともに, 基本的なサービス処理を RTL で実装することによって実行の高速化を図る. このために, タスクからサービス処理を起動する方式や, 同時に複数のサービス要求があった場合の調停方式を新たに提案する.

文献 [12] の手法と本稿の提案手法で生成されるハードウェアの構成をそれぞれ図 3 (a)(b) に示す. 図中 T1∼T3 はタスクを実行するハードウェアを, S1∼S4 はサービス処理を行うハードウェアを表す. 文献 [12] の手法では, OS のサービス処理をタスク側のプログラムの一部として高位合成していたため, 同じサービス処理が複数のタスクモジュール中に重複して存在していた. 本手法ではサービス処理はすべてマネージャに集約することによってこの重複を排除する.

サービス処理の起動は, タスクがレジスタ TFi, TAi にそれぞれ起動したいサービスの番号と引数を書き込むことにより行う. 複数のサービスの起動要求が同時に発生した場合にはマネージャがタスクのその時点での優先度に従って最初に処理するサービスを決定し, 対応するサービスモジュールを起動する. サービスモジュールの実行が完了すると, マネージャはサービスモジュールが出力する返り値を TAi に格納し, TFi に完了値を書き込んでタスクに完了を通知する.

3.2 マネージャ

3.2.1 構　　成

本手法で合成するハードウェアモジュールの詳細構成を図 4 に示す. TFi および TAi はタスク Ti とマネージャを仲介するレジスタである. status[Ti] はタスク Ti の状態変数を格納するレジスタ群であり, status[G] はシステムの状態変数を格納するレジスタ群である. リクエストアービタ (RA) は複数のサービス処理要求があった場合の調停を行う回路である. Si はサービス処理を行うハードウェア (サービスモジュール) であり, XF, XA, XT はマネージャとサービスモジュールを仲介するレジスタである. サービスモジュールは処理の実行のために status レジスタ群の読み書きを行う.

TFi と XF には, サービス要求の有無, サービスを処理するサービスモジュールの番号, サービス機能の番号等が符号化されている. TAi と XA はレジスタ配列であり, 要求のあったサービスに必要な個数だけ引数が書き込まれる.

3.2.2 処理の流れ

マネージャの具体的な動作の流れは次の通りである.

0) タスク Ti は引数を TAi に, サービスの ID 等を TFi に書き込むことによってマネージャにサービスを要求する.

1) マネージャは TFi を監視し, タスク Ti からサービスの要求があれば, XF, XA, XT にそれぞれ TFi, TAi, i を書き込む. 複数のサービス要求が同時にあった場合, あるいは, あるサービス要求の処理待ちの間に別のサービス要求があった場合には, タスクの優先度等に応じてそのうちの 1 つを選択して XF, XA, XT を書き込む. それ以外のサービス要求は, 実行中のサービスの終了を待つ. また, 実行状

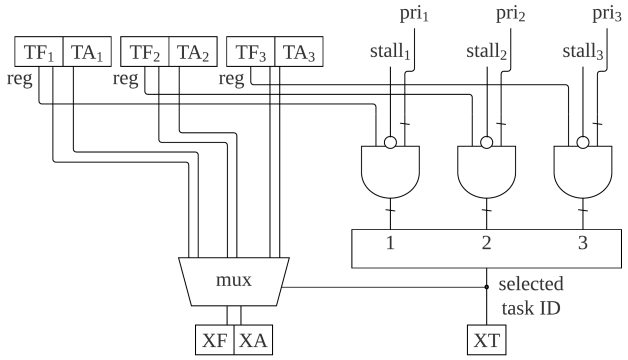


図 5: リクエストアービタの回路構成

態以外のタスクからの要求は, そのタスクが実行状態になるまで処理されない.

2) サービスモジュールは, XF, XA, XT に書き込みがあると, それを読んで要求されたサービスの処理を実行する. 処理が完了すると, XA に返り値を書き込んで完了信号を出す.

3) マネージャは XA の返り値を要求があったタスクの TAi にコピーし, タスク Ti に完了を通知する.

3.2.3 サービス要求の調停

リクエストアービタ (RA) は, 複数のサービス要求が同時に発生している場合に, 次に実行するサービスを決定し, XF, XA, XT レジスタを介して当該サービスモジュールを起動する. サービスの選択は, その時点でのタスクの状態と優先度に基づいて行う. すなわち, リクエストを出しているタスクのうち, その時点で実行中であり, かつその時点での優先度が最も高いタスクのリクエストを選択する. これは, タスクの状態や優先度は, リクエストを出してから処理が開始されるまでに変動する可能性があるためである.

RA の回路構成を図 5 に示す. TFi にリクエストがあって実行がストール状態でないタスクの優先度を比較し, 優先度最大のタスクの ID を XT に, 対応するリクエストと引数を XF と XA に書き込む.

3.3 サービスモジュールの動作

サービスモジュールは XF を監視し, XF に自分が担当するサービスの番号が書かれると処理を開始する. サービスモジュールは必要に応じて状態レジスタにアクセスし, タスクやシステムの状態変数を参照/更新する. 実行が完了すると (実行中にエラーが発生した場合を含む), サービスコールの返り値を XA レジスタに格納し, 完了信号 (DONE) を送出してマネージャに完了を通知する.

サービスコール act_tsk (引数で指定した ID のタスクを休止状態から実行可能状態に遷移させる) の処理を行うサービスモジュールの動作例を図 6 に示す. このサービスモジュールは, act_tsk 以外にも wup_tsk や slp_tsk 等, タスクの実行状態を管理するサービスコールの処理も行う. STATE はこのモジュールの状態レジスタであり, その初期値は S0 である. act_tsk の本質的な処理は, 指定されたタスクの状態を表す状態変数を書き換えるだけだが, エラー処理のために複数状態を要している. 具体的な動作の流れは次の通りである.

S0	DONE <= 0
	if (XF.mod == this) {
	  if(XF.func == ACT_TSK) {
	  tskid <= XA[0]
	  STATE <= A1
	  } else if(XF.func == WUP_TSK) {
	  tskid <= XA[0]
	  STATE <= W1
	  } else if(XF.func == SLP_TSK) {
	  ...
	  }
	}

A1	lock <= status[G][CPU_LOCK] STATE <= A2

A2	if (lock) {
	  XA[0] <= E_CTX
	  DONE <= 1
	  STATE <= S0
	} else {
	  actcnt <= status[tskid][ACTCNT]
	  STATE <= A3
	}

A3	if (actcnt >= TMAX_ACTCNT) {
	  XA[0] <= E_QOVR
	  DONE <= 1
	  STATE <= S0
	} else {
	  stat <= status[tskid][TSKSTAT]
	  STATE <= A4
	}

A4	if (stat != TTS_DMT) {
	  status[tskid][ACTCNT] <= act + 1
	} else {
	  status[tskid][TSKSTAT] <= TTS_RDY
	}
	XA[0] <= E_OK
	DONE <= 1
	STATE <= S0

図 6: サービスコール act_tsk の 処理の流れ

(状態 S0) XF を監視し, そこに書き込まれたサービスモジュールの ID (XF.mod) が自分のものであれば処理を開始する. 要求されているサービスの ID (XF.func) が ACT_TSK であれば, 引数 tskid を XA から読み出し, 状態 A1 に遷移する.

(状態 A1) CPU ロックがかかっていないかどうかを調べるためにシステム状態変数 status[G][CPU_LOCK] を読み出す.

(状態 A2) CPU ロックがかかっていればエラーコード E_CTX をレジスタ XA[0] に書き込んで, 完了信号 DONE を送出し, 待機状態 (S0) に戻る. そうでなければ, 対象タスクの起動要求キューイング回数を表す状態変数を読み出して, 状態 A3 に遷移する.

(状態 A3) 対象タスクの起動要求キューイング回数が上限を超えていれば, エラーとして E_QOVR を XA に書き込んで終了する. そうでなければ, 対象タスクの状態を読み出す.

(状態 A4) 対象タスクが休止状態であれば起動要求キューイング回数を更新し, 実行可能状態であれば実行状態に変更して終了する (待機状態に戻る).

なお, このように処理が簡単な場合は RTL の設計ができるが, 処理が複雑な場合には高位合成を用いても良い.

3.4 サービス呼び出し

レジスタ TFi と TAi はタスクのメモリ空間にマップされて

```
1 ER act_tsk(ID tskid) {
2   TA[TOPPERS_HW_SELF_ID][0] = tskid;
3   TF[TOPPERS_HW_SELF_ID] = F_ACT_TSK;
4   do {;} while ( T_EXEC( TF[TOPPERS_HW_SELF_ID] ) );
5
6   return TA[TOPPERS_HW_SELF_ID][0];
7 }
```

図 7: act_tsk の呼び出し

いるので, タスクはそのアドレスに対して読み書きすることにより, サービスの起動と返り値の受け取りを行う. サービスコールの本体は TFi と TAi の読み書きを行うスタブに変換する.

act_tsk のスタブの例を図 7 に示す. 2–3 行目で引数 (`tskid`) とサービス ID (`F_ACT_TSK`) を自タスク用の TAi[0] と TFi に書き込んでいる. 4 行目で完了を待ち (`T_EXEC` は TF 中の完了通知を読み出すマクロである), TAi[0] から読みだした引数を返り値として返している. (注1)

このスタブをタスク中にマクロ展開することにより, サービスコールを含めたタスクを高位合成することができる.

4. 実装と実験

本手法に基づき TOPPERS/ASP3 付属のサンプルプログラム “sample1” をハードウェア化した.

このプログラムは, 全体を制御するタスク MAIN_TASK, 例外を処理するタスク EXC_TASK および 3 つの並行実行タスク TASK1, TASK2, TASK3 からなる. MAIN_TASK はシリアル通信からのメッセージを受けとり, 以下のサービスコールを実行する.

act_tsk, can_act, ter_tsk, chg_pri, get_pri, wup_tsk,
can_wup, rel_wai, sus_tsk, rsm_tsk,
loc_cpu, unl_cpu

ただし, アラームハンドラ, サイクルハンドラと割り込みハンドラは未実装であり, 関連するサービスコールの呼び出しは削除している.

マネージャは Verilog HDL で設計した. サービスモジュールに関しては Verilog HDL で実装, TOPPERS/ASP3 のサービスコール 178 個中タスク操作に関する 28 のサービスコールをサポートした. タスクモジュールはバイナリ合成システム ACAP [16] により合成した.

これらの設計は, Xilinx Vivado 2016.4 で Xilinx FPGA Artix-7 (xc7a100tcsg324-3) をターゲットに論理合成した.

合成した各モジュールの回路規模を表 1 (a) に示す. TASK1, TASK2, TASK3 の回路規模は文献 [12] に比べて削減できている. ただし, その分サービスモジュール (serv_main, serv_other) とマネージャのリクエストアービタが必要になったため, 全体としては回路規模をあまり削減できなかった. これは, 今回の sample1 で使われているサービスコールの処理自体が軽量なため, 回路規模削減の効果が周辺のオーバヘッドの増加を大きく上回らなかったためと考えられる.

合成したシステムのクリティカルパス遅延を表 1 (b) に, 応答性能を表 1 (c) に示す. #cycle は, 各サービスコールが呼ばれてから状態が更新されるまでの実行サイクル数を表す. 例えば, 1 行目の act_tsk は, タスクを休止状態から実行可能状態に遷移させるまでのサイクル数であり, 文献 [12] の半分以下に削減できている. 処理の内容自体は変わっていないので, RTL で直接設計した影響と考えられる. latency は本手法では実行サイクル数とクリティカルパス遅延の積である. いずれのサービスコールも 200 ns 以内に実行することができた.

表 1: sample1 の合成結果

(a) 回路規模

module	文献 [12]		本研究	
	#LUT	#FF	#LUT	#FF
top	368	5	272	5
arbiter	383	5	366	5
serv_main	none	none	368	79
serv_other	none	none	3,098	163
manager	4,079	2,672	5,452	3,871
TASK1	5,520	929	5,219	929
TASK2	7,567	925	5,912	925
TASK3	7,189	965	6,035	965
MAIN_TASK	6,199	943	5,593	943
EXC_TASK	8,008	939	6,095	939
total	39,313	7,383	38,410	8,824

(b) クリティカルパス遅延 [ns]

文献 [12]	本研究
13.098	13.935

(c) 実行サイクル数と遅延

service call	文献 [12]		本研究	
	#cycle	latency [ns]	#cycle	latency [ns]
act_tsk	23	301.3	9	125.4
wup_tsk	28	366.7	9	125.4
ext_tsk	12	157.2	9	125.4
ras_ter	26	340.5	10	139.3
ter_tsk	23	301.3	8	111.5
slp_tsk	16	209.6	14	195.1

High-level synthesizer: ACAP [16]
Logic synthesizer: Xilinx Vivado (2016.4)
Target: Xilinx Artix-7 (xc7a100tcsg324-3)

5. むすび

本稿では, 文献 [12] の RTOS を用いたシステムのフルハードウェア合成における回路規模の削減と高速化を目的とした回路構成法を提案した. タスクに分散している RTOS のサービス処理機能をマネージャ側に集約し, 基本的なサービス処理を RTL で実装した. また, これを実装するためのサービス処理起動方式を提案した.

実験の結果, 本手法により生成されたハードウェアの回路規模は, 文献 [12] のシステムと比較して大きく削減することはできなかったが, 実行速度は向上した.

現在, 本稿の提案方式に基づいて MUTEX, イベントフラグ, キュー等の処理を行うハードウェアの設計を進めており, ハードウェア化可能な TOPPERS/ASP3 のサービス機能を拡大しているところである. また, 提案方式を FreeRTOS にも適用で

(注1) : 4 行目のように完了通知をポーリングするのではなく, 返り値の読み出しでタスクをストールさせても良い.

きるようにする計画である.

回路規模が大きいことが一つの課題であるが, バイナリ合成システム ACAP の合成方式にその一因があると考えられるため, Vivado HLS 等の汎用高位合成システムで合成できるようにアーキテクチャを調整することも今後の課題である.

謝　　辞

本研究に関して有益な御助言を頂いた元立命館大学の中谷嵩之氏, 元関西学院大学の田村真平氏, およびご支援を頂いた関西学院大学石浦研究室の諸氏に感謝致します. 本研究は一部 JSPS 科研費 19H04081 および 20H00590 の助成による.

文　　献

[1] Y. Cho, S. Yoo, K. Choi, N-E Zergainoh, and A. A. Jerraya: "Scheduler implementation in MPSoC design," in *Proc. ASP-DAC 2005*, pp. 151–156 (Jan. 2005)

[2] M. Vetromille, L. Ost, C. A. M. Marcon, C. Reif, and F. Hessel: "RTOS scheduler implementation in hardware and software for real time applications," in *Proc. RSP '06* pp. 163–168 (June 2006).

[3] P. Kohout, B. Ganesh, and B. Jacob: "Hardware support for real-time operating systems," in *Proc. CODES+ISSS '03*, pp. 45–51 (Oct. 2003). DOI:`http//doi.org/10.1145/944645.944656`

[4] T. Nakano, Y. Komatsudaira, A. Shiomi, and M. Imai: "Performance evaluation of STRON: A hardware implementation of a real-time OS," in *IEICE Trans. Fundamentals*, vol. E82-A, no. 11 pp. 2375–2382 (Nov. 1999).

[5] N. Maruyama, T. Ishihara, and H. Yasuura: "An RTOS in hardware for energy efficient software-based TCP/IP processing," in *Proc. SASP 2010*, pp. 58–63 (June 2010).

[6] 丸山修孝, 一場利幸, 本田晋也, 高田広章: "マルチコア対応 RTOS のハードウェア化による性能向上," 信学論 D, vol. J96–D, No. 10 pp. 2150–2162 (Oct. 2013).

[7] D. D. Gajski, N. D. Dutt, A. C-H Wu, and S. Y-L Lin: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).

[8] S. Shibata, S. Honda, H. Tomiyama, and H. Takada: "Advanced system-builder: A tool set for multiprocessor design space exploration," in *Proc. ISOCC 2010*, pp. 79–82 (Nov. 2010).

[9] Y. Ando, S. Honda, H. Takada, M. Edahiro: "System-level design method for control systems with hardware-implemented interrupt handler," *IPSJ Journal of Information Processing*, vol. 23, no. 5, pp. 532–541 (Sept. 2015).

[10] N. Ito, N. Ishiura, H. Tomiyama, and H. Kanbara: "High-level synthesis from programs with external interrupt handling," in *Proc. SASIMI 2015*, pp. 10–15 (March 2015).

[11] N. Ito, Y. Oosako, N. Ishiura, and H. Tomiyama, and H. Kanbara: "Binary synthesis implementing external interrupt handler as independent module," in *Proc. RSP 2017*, pp. 92–98 (Oct. 2017).

[12] Y. Oosako, N. Ishiura, H. Tomiyama, and H. Kanbara: "Synthesis of full hardware implementation of RTOS-based systems," in *Proc. RSP 2018*, pp. 1–7 (Oct. 2018).

[13] 大迫 裕樹, 石浦 菜岐佐, 神原 弘之, 冨山 宏之: "RTOS を用いたシステムのフルハードウェア実装とその自動化," 信学技報, VLD 2019-03, (Mar. 2019).

[14] 中野 和香子, 石浦 菜岐佐, 神原 弘之, 冨山 宏之: "FreeRTOS を用いたシステムのフルハードウェア合成," 信学技報, VLD 2020-01 (Jan. 2020).

[15] ITRON Committee, TRON association: *μITRON4.0 Specification* (1999, 2002). Available at `http://www.ertl.jp/ITRON/SPEC/FILE/mitron-400e.pdf` (accessed 2018-06-11).

[16] N. Ishiura, H. Kanbara, and H. Tomiyama: "ACAP: Binary synthesizer based on MIPS object codes," in *Proc. ITC-CSCC 2014*, pp. 725–728 (July 2014).