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
VLD2018-122,HWS2018-85(2019-02)

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

大迫　裕樹† 　石浦菜岐佐† 　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 を用いたプログラムを高位合成によりフルハードウェア実装する手法を提案する. 本手法では, タスク/ハンドラ等のカーネルオブジェクトおよび RTOS カーネルの機能を, これらを実行する CPU と機能等価なハードウェアに合成する. 全タスク/ハンドラを独立したハードウェアモジュールに合成することにより, 実行可能状態のタスク/ハンドラは並列に実行を行うことが可能になる. これにより, 処理性能を向上させるとともに, コンテキストスイッチの時間を大きく削減する. また, タスク/ハンドラの実行/停止を, その状態に基づいて stall 信号で制御することにより, RTOS のタスクスケジューリング機能を軽量なハードウェアで実現する. 本手法に基づく合成システムを実装し, TOPPERS/ASP3 カーネル付属サンプル “sample1” をハードウェア実装した. 実験の結果, タスクの起動を 23 サイクル, 割込みハンドラの起動を 1 サイクルで行えることを確認した.

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

Synthesis of Full Hardware Implementation of RTOS-Based Systems

Yuuki OOSAKO†, Nagisa ISHIURA†, 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, 134 Chudoji Minamimachi, Shimogyo–ku, Kyoto, 600–8813, Japan

Abstract　This paper presents a method of automatically synthesizing a hardware design from a set of source codes for a real-time system utilizing an RTOS. It generates a full hardware implementation where all the tasks and handlers in the system as well as all the necessary services provided by the RTOS kernel are implemented as hardware. Every task and handler is synthesized into an independent hardware module so that it may run in parallel with the other tasks/handlers as soon as it is ready. This leads to task switching with extremely low overhead and reduced computation time both by parallel and hardware execution. Moreover, this eliminates the necessity of the task queue management; task scheduling is realized by a relatively simple manager hardware which instructs each task/handler to run or stall based on the values of its status variables. Since most of the API calls from tasks/handlers are reduced to reads/writes of these status variables, they can be expanded inline into the tasks/handlers' source codes which are compiled into hardware designs by a high-level synthesizer. We have implemented a prototype synthesis system which assume the use of the TOPPERS/ASP3 real-time kernel. A hardware implementation synthesized from a `sample1.c` code, bundled in the TOPPERS/ASP3 release, took 23 cycles for waking up a waiting task and only 1 cycle for activating an interrupt handler.

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って, 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いるが, これらに必要な機器を実装するために, 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能が要求される.

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

This article is a technical report without peer review, and its polished and/or extended version may be published elsewhere.
Copyright ©2019 by IEICE

OS (RTOS) が不可欠である．RTOS は, 制約時間内のタスクの実行完了を実現するための機能を提供するが, システムの複雑化が進んでいるため, リアルタイム性の実現は難しい課題となっている．

RTOS を用いたシステムを高速化する一手法として, RTOS 機能のハードウェア実装がある．文献 [1] [2] [3] では, ハードウェアにより RTOS のスケジューラの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/ハンドラはソフトウェアのままであり, プロセッサのタスク切り替えのオーバーヘッドが発生する．また, タスクの処理の負荷が大きい場合は応答時間が改善されない場合もある．

一方, 高位合成技術 [7] を用いることにより, タスク/ハンドラを専用ハードウェアに合成し, 高速化することが可能である．指定したタスク/ハンドラ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のみである．

本稿では, この課題を解決する手法として, RTOS 機能とタスク/ハンドラの全てをハードウェア化する手法を提案する [12]．従来の RTOS の高速化手法と異なり, 本手法では, スケジューリングキューを廃している．タスク/ハンドラは独立したハードウェアモジュールとして実装し, 実行可能状態になったものは全て, 即座に実行を開始することができる．従来のスケジューリング機能は, タスク/ハンドラの実行/停止を制御する回路に置き換える．

本手法に基づき, TOPPERS/ASP3 カーネルを用いて C 言語で記述されたリアルタイムシステムから, Verilog HDL で記述されたハードウェア設計を生成するための合成システムを実装した．予備実験では, タスクの起動が 23 サイクル, 割込みハンドラの起動が 1 サイクルで行えることを確認した．

2. リアルタイム OS

RTOS (Real-Time Operating System) は, 複数の逐次処理を並行に動作させる．本稿では, この処理単位を “タスク” と呼ぶ．タスクは特定の条件を満たすと実行可能となる．実行可能状態のタスクが存在する時, そのうちの 1 つが実行され, その他のタスクは CPU 待ち状態となる．この実行タスクの選択は RTOS のスケジューラが行う．リアルタイムシステムでは, タスク毎にデットラインが定義され, 各タスクの処理はこのデッドラインまでに処理を完了することが求められる．RTOS はこれを可能にする仕組みを持が, そのうちの 1 つが, タスクの優先度設定とそれに基づくスケジューリングである．各タスクには優先度が設定され, 最も優先度の高いタスクが実行される．優先度を動的に変更したり, 実行中のタスクを中断してより優先度の高いタスクに実行を切り替えるプリエンプションが行えるものもある．“ハンドラ” は, イベントを処理するための処理単位である．周期ハンドラは, アクティブ状態の時, 特定の周期で実行され, アラームハンドラは, 指定した時間が経過した後, 実行される．割込みハンドラは, 割込み信号もしくはイベントにより起動される．一般的に, これらのハンドラは, タスクよりも高い優先度で実行される．また, RTOS は排他制御やタスク間通信等の機能も提供する．

リアルタイムシステムにおいては, 以下の 3 つのオーバーヘッドが存在する．

1) スケジューリング: 新しいタスクもしくはプリエンプトされたタスクをレディーキューに追加し, 次に実行されるタスクを選択する処理．

2) コンテキストスイッチ: 実行中のタスク/ハンドラのコンテキスト (レジスタの値) を保存し, 次に実行されるタスク/ハンドラのコンテキストを読込む処理．

3) CPU 待ち: 実行可能状態のタスクが, 自身より高優先度のタスクの実行完了を待機する時間．

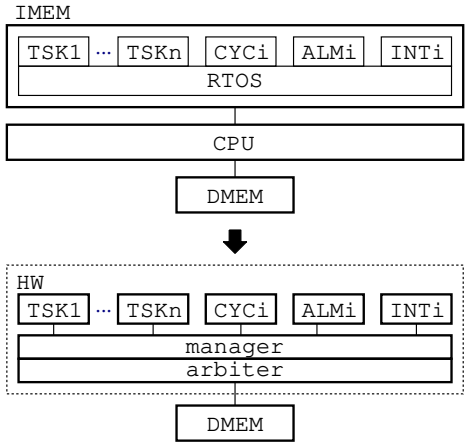

図 1: RTOS を用いたシステムのフルハードウェア実装

これらのオーバーヘッドの改善手法として, RTOS のスケジューラのハードウェア化 [1] [2] [3] や, RTOS 機能のハードウェア化 [4] [5] [6] が提案されているが, これらの手法で改善できるのはスケジューリングに関するオーバーヘッドのみである．コンテキストスイッチのオーバーヘッドの改善には, CPU 側でのハードウェアによる高速化が必要であり, CPU 待ち時間の改善には, CPU の並列処理機能の改善が必要となる．

3. RTOS を用いたシステムのフルハードウェア実装

3.1 概　　要

本稿では, リアルタイムシステムを高位合成によりフルハードウェア実装する手法を提案する．RTOS 上で動作するアプリケーションプログラムを入力することにより, これを実行する CPU と機能等価なハードウェアを自動生成する．

本手法で生成するハードウェアの概観を図 1 に示す．入力となるシステムは, タスク (TSKi) および周期/アラーム/割込みハンドラ (CYCi, ALMi, INTi) からなり, タスク/ハンドラおよび RTOS のプログラムは命令メモリ (IMEM) に格納されている．本手法では, タスク/ハンドラはコンパイル時に静的生成されるものとし, 動的生成は行わないものとする．本手法は命令メモリおよび CPU を機能等価なハードウェアに合成する．生成される各タスク/ハンドラは独立したハードウェアモジュールであり, 実行可能状態になったタスク/ハンドラはその優先度によらず, 即座に実行を開始できる．また, 従来のタスクスケジューラやレディーキューは不要となり, manager モジュールがタスク/ハンドラの実行を制御する．複数のモジュールから同時にメモリ (DMEM) へのアクセス要求があった場合は, arbiter モジュールが調停を行う．この調停は, タスク/ハンドラの優先度を用いて行う．

なお本稿では, 入力となるプログラムは, 複数のタスク/ハンドラが並列に動作可能であるように排他制御が行われていることを前提とする．すなわち, 同時に 2 つ以上のタスク/ハンドラが同時に動作しないことを前提とした排他制御を行っているシステムの合成は本手法の対象外とする．

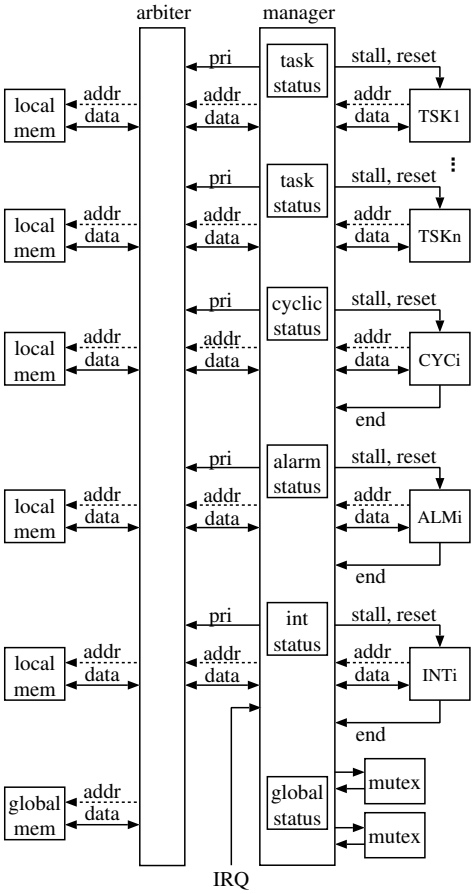

図 2: 合成されるハードウェアの構成

3.2 ハードウェア構成

本手法で合成するハードウェアの構成を図 2 に示す.

TSKi, CYCi, ALMi, および INTi は, 図 1 のタスク/ハンドラである. 各タスク/ハンドラモジュールは stall, reset の入力ポートを持つ. stall が 1 の時, タスク/ハンドラの動作は停止する. また, reset が 1 の時, タスク/ハンドラはリセットされ, 初期状態に戻る.

manager モジュールは stall と reset の値を, 各 status レジスタ (task status, cyclic status, alarm status, int status) の値を基に決定する. status レジスタは, 各タスク/ハンドラモジュールごとの実行状態, 優先度, 待ち要因等を保存する. また, 周期ハンドラ/割込みハンドラの status レジスタは, 実行開始までの時間の値も保持する. 例えば, タスクが休止状態の時, stall の値は 1 であり, 実行状態になると同時に 0 に変化する.

status レジスタはメモリ空間にマッピングし, タスク/ハンドラからは特定のアドレスに対するメモリアクセスで読み書きできるようにする. それ以外のアドレスに対するメモリアクセスの場合, manager モジュールは arbiter モジュールへメモリアクセス要求信号を送信する. RTOS の API の大部分は status レジスタを参照/更新する処理として実装できる.

周期ハンドラ/割込みハンドラ等のタイムイベントの管理には, status レジスタ内のクロックカウンタを用いる. カウンタの値はクロック毎に減じられ, 値が 0 になると stall が解除されて, ハンドラが実行される. 割込みハンドラは, manager への IRQ 信号によって起動される. IRQ 信号が送られた時, 割込みが禁止されていなければ, 割込みハンドラへの stall が解除され, ハンドラが起動する.

本手法で合成するハードウェアでは 全タスク/ハンドラが並列に実行可能であるため, 同一のメモリバンクに対して複数のアクセスが発生する可能性がある. arbiter モジュールは, これらのアクセス要求をタスク/ハンドラの優先度を用いて調停する. つまり, 最も高い優先度を持つタスク/ハンドラのアクセスを優先し, その他のモジュールに対しては stall 信号が送られる.

また, タスク/ハンドラの並列実行により, 処理内容が競合する API 要求が同時発生する可能性がある. この調停は manager モジュールが行う. 同時に複数の API 要求が発生した場合, 1 モジュールからの要求のみが処理され, その他のモジュールには stall 信号が送られる. この調停はミューテックスを用いて実現する.

ミューテックスはハードウェアモジュールとして実装する. 複数の取得要求があった場合, そのうちの 1 つがミューテックスを獲得し, その他のミューテックス待ちのモジュールには stall 信号が送られる.

表 1: TOPPERS/ASP3 におけるタスクの状態

状態名	意味
休止状態	タスクが実行すべき処理がない状態
実行可能状態	タスク自身は実行できる状態にあるが, それよりも優先順位の高いタスクが実行状態にあるために, そのタスクが実行されない状態
実行状態	タスクが実行されている状態. (タスクの実行中に発生した割込みまたは例外処理中かつ, タスクコンテキストに戻った後に, そのタスクの実行を再開する状態を含む)
待ち状態	タスクが何らかの条件が揃うのを待つために, 自ら実行を止めている状態
強制待ち状態	他のタスクによって, 強制的に実行を止められている状態
二重待ち状態	待ち状態と強制待ち状態が重なった状態

4. TOPPERS/ASP3 を用いたシステムからの合成

本章では, TOPPERS/ASP3 カーネル (注1) に基づいた合成手法の詳細について述べる.

TOPPERS/ASP3(注2) は, μITRON4.0 [13] 準拠の第 3 世代リアルタイムカーネルである. TOPPERS/ASP3 では, タスク/ハンドラはコンパイル時に静的生成される. 各タスクは, 表 1 の 6 つのうち, どれかの状態をとり, 初期状態はコンパイル時にコンフィギュレーションファイルで指定する. 全タスクは単一のメモリ空間を共有しており, スタックのサイズおよび開始アドレスはタスク/ハンドラ毎にコンフィギュレーションファイルで指定することが可能である. 表 2 はタスク/ハンドラの状態制御に関する主要なサービスコールである. 例えば, `act_tsk` は指定した ID のタスクを休止状態から実行可能状態に遷移させる.

4.1 タ ス ク

タスクへの stall 信号は実行状態の時のみ 0 となり, その他の状態 (休止, 実行可能, 待ち, 強制待ち, 二重待ち) の時は 1 となる. また, メモリアクセス要求もしくはミューテックス取得要求が待たされている場合にも stall 信号は 1 となる.

全タスクは並列に動作することが可能なので, 実行可能状態になったタスクは, ディスパッチ禁止状態でなければ即座に実行状態に遷移する. この状態遷移の管理は manager モジュールが行う. あるタスクが実行可能状態になった時, ディスパッチが禁止されていなければ, manager モジュールは, 次のクロックでタスクの状態を実行状態に更新し, stall 信号を解除する.

動作中のタスクはサービスコール `ext_tsk` を呼ぶことによ

(注1) : `https://www.toppers.jp/en/asp-kernel.html`

(注2) : `http://www.toppers.jp`

表 2: 実行制御と排他制御のためのサービスコール

(a) タスク実行制御

サービスコール	機能
act_tsk(ID tskid)	指定したタスクに対して起動要求を行う
slp_tsk()	自タスクを起床待ちさせる
tslp_tsk(TMO timeout)	自タスクをタイムアウト付きで起床待ちさせる
wup_tsk(ID tskid)	指定したタスクを起床する
rel_wai(ID tskid)	指定したタスクを, 強制的に待ち解除する
sus_tsk(ID tskid)	指定したタスクを強制待ちにする. 対象が待ち状態であれば, 二重待ち状態にする
rsm_tsk(ID tskid)	指定したタスクを, 強制待ちから再開する. 対象が二重待ちであれば, 待ち状態にする
dly_tsk(RELTIM time)	指定した時間, 自タスクを待ち状態にし, 遅延させる
ext_tsk()	自タスクを終了させる
ras_ter(ID tskid)	指定したタスクに終了要求を行う
ter_tsk(ID tskid)	指定したタスクを終了させる

(b) ハンドラ実行制御

サービスコール	機能
sta_cyc(ID cycid)	指定した周期通知を動作開始する
stp_cyc(ID cycid)	指定した周期通知を動作停止する
sta_alm(ID almid, RELTIM almtim)	指定したアラーム通知を動作開始する. 通知時刻は, sta_alm を呼び出してから, almtim で指定した相対時間後に設定される
stp_alm(ID almid)	指定したアラーム通知を動作停止する

(c) 排他制御

サービスコール	機能
loc_mtx(ID mtxid)	指定したミューテックスをロックする
unl_mtx(ID mtxid)	指定したミューテックスをロック解除する

表 3: タスクの状態変数

変数	意味
tskstat	現在の実行状態
tskpri	現在の優先度
tskbpri	ベース優先度
tskwait	待ち要因
wobjid	待ち対象のオブジェクトの ID
lefttmo	タイムアウトするまでの時間
actcnt	起動要求キューイング数
wupcnt	起床要求キューイング数
raster	タスク終了要求状態
dister	タスク終了禁止状態

り終了する. **ext_tsk** は, 自タスクを休止状態に遷移させるものである. このサービスコールが呼ばれると, manager は reset 信号と stall 信号を送り, タスクの初期化を行う.

各タスクの状態は, manager モジュール内の status レジスタに保存する. ASP3 カーネルでは, タスクの状態変数 (C 言語の構造体) は表 3 の 10 個のメンバからなり, 本手法では, これらに対応する 10 個のレジスタを用いる. status レジスタはメモリ空間にマッピングされており, タスクモジュールからはメモリのロード/ストア命令でアクセスできる. さらに, global status レジスタもタスクモジュールの制御に使用する. global status レジスタは, 全割込みロックフラグ, CPU ロックフラグ, 割込み優先度マスク, ディスパッチ禁止フラグを持つ.

ASP3 カーネルのほとんどのサービスコールは status レジスタの参照/更新処理として実装する. 図 3 に, タスクを休止状態から実行可能状態に遷移させるサービスコール act_tsk の処

```
1  ER act_tsk(ID tskid) {
2
3    if (IS_TASK_CONTEXT && tskid == TSK_SELF) {
4      tskid = TOPPERS_HW_SELF_ID;
5    }
6
7    if (tskid <= 0 || TNUM_TSKID < tskid) {return E_ID;}
8    volatile T_RTSK *const target =
9      &(task_status[tskid-1].rtsk);
10
11   _loc_service_call();
12
13   const uint_t actcnt = target->actcnt;
14
15   ER rc = E_OK;
16   if (glob_status.f_cpu_locked) {rc = E_CTX;}
17   else if (actcnt >= TMAX_ACTCNT) {rc = E_QOVR;}
18   else if (target->tskstat != TTS_DMT) {rc = E_QOVR;}
19   else {target->tskstat = TTS_RDY;}
20
21   _unl_service_call();
22
23   return rc;
24 }
```

図 3: サービスコール act_tsk の C 言語実装

理内容を示す. 19 行目がタスクを実行可能状態に遷移させる処理, 3-5, 8, 9 行目がサービスコール内部の前処理, 7, 15-18 行目がエラーチェックである. これらは, 本質的に ASP3 カーネルのソースコードと同様である. 11, 21 行目はサービスコールのシリアライズのためのミューテックスの取得/解放処理であり, これが本手法で新たに追加した部分である.

高速化のため, 各サービスコールはインライン展開した後に高位合成する.

4.2 ハンドラ

周期ハンドラ/アラームハンドラは, 基本的にタスクと同様の方法で制御する. 相違点は, ハンドラは end 信号を manager に送ることにより, 実行完了を通知する点である.

manager は各周期ハンドラに対して 2 つの変数を持つ. **cycstat** はハンドラの停止/動作中の状態を表し, 値が 1 であれば動作中, 0 であれば停止中である. **lefttim** はタイマ変数であり, 次の実行までの時間をミリ秒単位で保存する. 周期ハンドラが起動するとタイマには周期時間がセットされる. また, タイマに付随する変数として, サブカウンタが存在する. サブカウンタは ASP3 で扱うことのできないミリ秒未満の値を保持し, クロック毎に減算される. サブカウンタが 0 になると, タイマの値も減算される. タイマが 0 になると, ハンドラへの stall が解除され, また, 同時にタイマも再度セットされる. manager モジュールは, 動作開始のリクエストを保存するためのフラグ変数として f_reset を補助的に用いる.

周期ハンドラの具体的な動作の流れは以下の通りである.

1) (task) sta_cyc() を呼ぶ. このサービスコールは, f_reset, cycstat に 1 をセットする.

2) (manager) f_reset が 1 ならば, lefttim に周期時間をセットし, f_reset を 0 にする.

3) (manager) lefttim をデクリメントする

4) (manager) lefttim が 0 なら, stall を 0 にし, ハンドラの実行を開始する. また, (周期ハンドラの場合のみ) lefttim に周期時間を再セットする.

5) (handler) 実行が完了すれば, end 信号を 1 にする.

6) (manager) end 信号が 1 ならば, stall, reset をともに 1 にし, ハンドラを初期状態に戻す.

7) (manager) 3 に戻る.

割込みハンドラは, 周期ハンドラ/割込みハンドラと同様に制御する. manager は, 割込み要求信号を受け取ると, 割込みが禁止されていなければ割込みハンドラへの stall 信号を 0 にする.

TOPPERS における割込みは, 複数のデバイスが接続した“要求ライン” と呼ばれる信号線のモデルを用いてグループ化される. 各割込み要求ラインには動的に変更可能な優先度が設定され, この値が割込み優先度マスクより大きい場合は割込み処理は行われない. また, 各割込み要求ラインは割込み禁止フラグを持つ.

本手法では, 各割込み要求ラインに対して 1 つの割込みハンドラのハードウェアモジュールを合成し, このモジュールが割込み要求ラインに接続している全てのデバイスの割込みサービスルーチンを実行する.

割込み要求が発生した時, manager モジュールは以下の条件を満たしているかを確認する.

- 割込み要求ラインに対する割込み要求禁止フラグがクリアされている
- 割込み要求ラインに設定された割込み優先度が, 割込み優先度マスクの現在値よりも高い (優先度の値としては小さい)
- 全割込みロックフラグがクリアされている
- 割込み要求がカーネル管理の割込みである場合には, CPU ロックフラグがクリアされている

これらの条件を満たしていた場合, manager モジュールは, ハンドラの実行状態を保存する `exec` レジスタを 1 に, stall 信号を 0 にし, ハンドラの実行を開始する. ハンドラの実行が完了するとハンドラモジュールは, manager モジュールへの end 信号値を 1 にし, 実行完了を通知する. その後, manager モジュールは, exec レジスタを 0 に, stall を 1 にする.

4.3 ミューテックス

本手法では, ミューテックスをハードウェアモジュールとして実装する. 複数のタスク/ハンドラから同時にミューテックスの取得要求があった場合, ミューテックスモジュールはそのうちの 1 つのみを許可し, 他のタスク/ハンドラには stall を送る.

ミューテックスの取得/解放処理の流れは以下の通りである. タスク/ハンドラは, ミューテックスの取得にはサービスコール loc_mtx, 解放には unl_mtx をそれぞれ用いる. ミューテックスを取得した場合, ミューテックスオブジェクト内のメンバ acquired の値が 1 になる.

1) (task/handler i) サービスコール loc_mtx を呼ぶ.
2) (manager) ミューテックスモジュールへの i 番目のロック要求信号 LCKi の値を 1 にする.
3) (mutex) i 番目のタスク/ハンドラへの取得成功信号 ACQi を 1 にする. 複数の取得要求が同時発生した場合, i の値が最も小さいものを選択する.
4) (manager) LCKj==1 and ACQj==0 である j 番目のタスク/ハンドラに対して stall の値を 1 にする.
5) (manager) i 番目のタスク/ハンドラからの, ミューテックスオブジェクトのメンバ変数 `acquired` に対する load 命令の結果として LCKi の値を返す
6) (task/handler i) acquired==1 ならば, クリティカルセクションの実行を開始する.
7) (task/handler i) サービスコール unl_mtx を呼ぶ.
8) (manager) ミューテックスモジュールへの i 番目のロック解放信号 RELi の値を 1 にする.
9) (mutex) i 番目のタスク/ハンドラへの取得成功信号 ACQi を 0 にする.

5. 実装と実験

タスク/ハンドラは, 高位合成によりレジスタ転送レベルのハードウェア記述に合成する. 一方, manager や arbiter はシステムの構成に適したポート数のものをスクリプトで生成する. その他, mutex 等のモジュールは手設計する. タスク/ハンドラ, ミューテックスの数等はコンフィギュレーションファイル中の定義から抽出する.

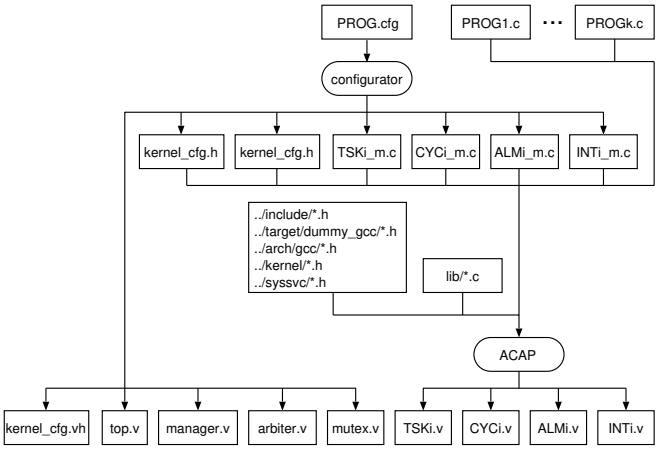

図 4: 合成の流れ

5.1 実　　装

本手法に基づく合成システムを, 高位合成システム ACAP [14] を用いて Perl5 で実装した.

ACAP は, C プログラムもしくは MIPS R3000 の機械語プログラムを入力として, これを 32bit の整数演算命令を並列に実行可能なハードウェアに変換する. まず, 入力となるプログラムは GCC を用いてコンパイル/リンクされる. 得られた MIPS の機械語プログラムを CDFG (control dataflow graph) に変換し, 最適化, スケジューリング, バインディングの処理を施して, Verilog HDL を生成する. これにより ACAP は, C プログラムの修正なしに, タスク/ハンドラを CPU と機能等価なハードウェアに合成できる. Vivado HLS(注3) 等の高位合成システムの場合でも, ソースコードへ少しの修正を加えることにより同様に合成することが可能であると考えられる.

本稿で実装したシステムによる合成の流れを 図 4 に示す. まず, コンフィギュレーションファイル PROG.cfg から各種カーネルオブジェクトの構成を読み取り, C の定義ファイル kernel_cfg.c, kernel_cfg.h および status レジスタの構成を記述した kernel_cfg.vh を生成する. また, 各タスク/ハンドラのメイン関数を含む C ファイル (TASKi_m.c)(注4) を生成する. これらのソースコードおよびアプリケーションプログラム (PROG1.c, ···, PROGk.c) から, ACAP を用いて Verilog HDL で記述されたハードウェア設計を生成する. これと同時に, manager/arbiter/mutex の Verilog HDL 記述を, システム構成に適したポート数で生成する.

現時点で, TOPPERS/ASP3 のサービスコール 178 個中 38 個を実装している.

5.2 実 装 結 果

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 と, アラームハンドラ ALMHDR1, 周期ハンドラ CYCHDR1, 割込みハンドラ INTNO1 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, ras_ter, rot_rdq,

(注3) : `https://www.xilinx.com/products/design-tools/vivado.html`
(注4) : ACAP は各ハードウェアモジュールに対して main 関数が必要である

表 4: `sample1` の合成結果

(a) 実行サイクル数と遅延

service call	#cycle	latency [ns]
act_tsk	23	301.3
wup_tsk	28	366.7
ext_tsk	12	157.2
ras_ter	26	340.5
ter_tsk	23	301.3
slp_tsk	16	209.6
loc_mtx	25	327.5
unl_mtx	10	131.0
sta_cyc	19	248.9
sta_alm	20	262.0
interrupt	1	13.1

(b) 回路規模

module	#LUT	#FF
top	96	1
manager	5,305	3,027
arbiter	573	10
mutex0	29	16
mutex1	29	16
MAIN_TASK	5,382	632
EXC_TASK	4,703	399
TASK1	5,320	950
TASK2	6,620	649
TASK3	6,258	649
ALMHDR1	8,252	850
CLCHDR1	6,714	852
INTNO1	5,669	639
total	54,950	8,689

High-level synthesizer: ACAP (2016.10)
Logic synthesizer: Xilinx Vivado (2016.4)
Target: Xilinx Artix-7 (xc7a100tcsg324-3)

sta_cyc, stp_cyc, sta_alm, stp_alm, loc_cpu, unl_cpu

各並行実行タスクは, メモリに実行されたことを示すデータを書き込み, 周期ハンドラ, アラームハンドラ, 割込みハンドラはレディーキューを回転させる. EXC_TASK は, ログを残した後にシステムコール ext_ker を呼ぶことによりシステムを終了させる.

生成された Verilog HDL は, 論理合成ツール Xilinx Vivado 2016.4 によって, Xilinx FPGA Artix-7 (xc7a100tcsg324-3) をターゲットに論理合成した.

合成したハードウェアにおけるレスポンス性能を表 4 (a) に示す. #cycle, latency はそれぞれ, 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, 時間を表す. 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に 23 サイクル必要である. (そして次のサイクルで実行状態に遷移する.) このうち, 状態の更新に要するのは 2 サイクルであり, それ以外の 21 サイクルはエラーチェックやシリアライズのためのミューテックス操作を行っている. latency は実行サイクル数とクリティカルパス遅延 13.098 ns の積である. いずれのサービスコールも 400 ns 以内に実行することができた. また, 割込みハンドラの起動は 1 サイクルで行うことができた.

表 4 (b) は各ハードウェアモジュールの回路規模を示す. #LUT はルックアップテーブル数, #FF はフリップフロップ数である. MIPS R3000 互換のソフトコアプロセッサが 3200 LUT であるため, 高位合成により生成したタスク/ハンドラの回路規模はやや大きいといえる. これについては, ハードウェアの最適化の余地があると考えられる.

6. むすび

本稿では, RTOS を用いたシステムをフルハードウェア実装を自動生成する手法を提案した. 本手法により生成されたハードウェアは, 応答性能が大きく向上していることを確認した.

現在, 生成したハードウェアの回路規模がやや大きいため, 高位合成の入力となるソースコードの改善, 生成されるハードウェアの最適化を進めている. また, 汎用の高位合成システムを用いることも検討している.

他の課題として, 複数のタスク/ハンドラが同時に実行されないことを前提にした排他制御を行っているプログラムからの合成が挙げられる. 解決手法として, 最も高い優先度を持つタスク/ハンドラのみを並列動作させるモード, 単一のタスク/ハンドラのみが動作するモードを導入することを検討している.

また, イベントフラグ, データキュー, メッセージバッファ等に関わる TOPPERS/ASP3 の他のサービスコールの実装も今後の課題である. さらに, 本手法の FreeRTOS(注5) 等の他の RTOS への適用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に感謝致します. 本研究は一部 JSPS 科研費 16K00088, 16K01207 および 15H02680 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).

[4] T. Nakano, Y. Komatsudaira, A. Shiomi, and M. Imai: "Performance evaluation of STRON: A hardware implementation of a real-time OS," *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] Renesas Electronics Corp.: "Issues with Real Time Performance in Conventional RTOS and Performance Improvements through HW-RTOS," (Sep. 2018). available at `https://www.renesas.com/kr/en/doc/DocumentServer/009/r70wp0002ej0100-rtos.pdf` (accessed 2019-01-30).

[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 (Mar. 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] ITRON Committee, TRON association: *μITRON4.0 Specification* (1999, 2002). Available at `http://www.ertl.jp/ITRON/SPEC/FILE/mitron-400e.pdf` (accessed 2019-01-30).

[14] 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).

(注5) : https://www.freertos.org