RTOS利用システムのフルハードウェア化における管理ハードウェアの自動生成

南口 比呂[1,a)]　石浦 菜岐佐[1]　冨山 宏之[2]　神原 弘之[3]

概要：本稿では, RTOS を利用したシステムのフルハードウェア化において, RTOS の機能を提供するハードウェアの自動生成を行う. 六車・安堂らはリアルタイムシステムのタスクおよび RTOS の機能を全てハードウェア実装する手法を提案しているが, RTOS の機能を提供する管理ハードウェアは Verilog HDL で手動設計している. 本稿では, 設計対象のシステムの構成情報を記述したコンフィギュレーションファイルからこの管理ハードウェアを自動生成する. 使用する RTOS に依存する機能の情報をコンフィギュレーションファイルに記述することにより複数の RTOS に対応する. また, RTOS のサービスを提供するモジュールにおいて使用されない機能に対する記述の生成を省くことにより回路規模の削減を図る. 提案のシステムを Perl5 で実装した結果, TOPPERS/ASP3 および FreeRTOS それぞれの仕様に応じた管理ハードウェアを自動生成できた. タスク数 4, 8, 16 のシステムについてハードウェアを合成した結果, 回路規模と遅延はそれぞれ概ねタスク数とその対数に比例して増加した.

キーワード：システム設計技術, リアルタイムシステム, RTOS, ハードウェア実装

Automatic Generation of Management Hardware for Full Hardware Implementation of RTOS-Based Systems

Minamiguchi Hiro[1,a)]　Ishiura Nagisa[1]　Tomiyama Hiroyuki[2]　Kanbara Hiroyuki[3]

Abstract: This article presents an automatic scheme for generating hardware that provides RTOS's functions for full-hardware implementation of RTOS-based systems. Though Muguruma and Ando et al. have proposed a method for implementing both tasks of a real-time system and RTOS functions as hardware, the manager hardware to provide RTOS functions was designed manually in Verilog HDL. In our method, the manager hardware is automatically generated from a file that describes the configuration of the target system. Manager hardware for different RTOSes can be generated by describing RTOS dependent attributes in the configuration file. Generation of RTL description for unused function is omitted, which reduces the size of the resulting hardware. A prototype system based on the proposed method has been implemented in Perl5, which successfully generated manager hardware modules for 4 tasks for both TOPPERS/ASP3 and FreeRTOS. Manager hardware modules for 4, 8, and 16 tasks have also been generated, which suggests the circuit size and the critical path delay increases in proportion to the number of tasks and their logarithms, respectively.

Keywords: System design technology, Real-Time Systems, RTOS, Hardware Implementation

[1] 関西学院大学
Kwansei Gakuin Univ.
[2] 立命館大学
Ritsumeikan Univ.
[3] 京都高度技術研究所
ASTEM RI/KYOTO.
[a)] gtu14556@kwansei.ac.jp

1. はじめに

近年の情報通信技術の発展により, 日々様々な新しいサービスやデバイスが開発されつつあるが, これに伴って組み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, 車載機器や無人航空機の制御では機能性に加えて高い応答性能も要求される．このようなリアルタイムシステムの設計, 開発はリアルタイム OS (RTOS) を用いて行われる．RTOS は決められた期限内に処理を完了するリアルタイム性の実現をサポートする機能を提供するが, システムの高機能化・複雑化に伴う処理量の増加によりリアルタイム性の実現が困難になりつつある．

RTOS を用いるシステムの応答性能を向上させる一手法として, RTOS 機能のハードウェア化が提案されている．文献 [1], [2] は, RTOS のスケジューリング機能をハードウェア化し, 文献 [3], [4] は RTOS の機能の大部分をハードウェア化することによりシステムの応答性能の向上を図っている．しかし, タスク/ハンドラはソフトウェアであるため, タスク切り替えの際にコンテキストスイッチのオーバヘッドが発生する課題がある．

これに対し, 文献 [5] は RTOS の機能とタスク/ハンドラの全てをハードウェア実装することにより, タスク数があまり多くないシステムを対象に応答性能を飛躍的に向上させる手法を提案している．文献 [6] は文献 [5] の回路規模を削減するためにタスク内部に重複していた RTOS のサービス機能を管理ハードウェアに集約する新たなハードウェアアーキテクチャを提案した．また, 文献 [7] は文献 [6] のアーキテクチャにおいて, タスクを汎用的な高位合成ツールで合成できる制御手法を提案している．

しかし, 文献 [7] では RTOS の機能を提供する管理ハードウェアを Verilog HDL で手動設計しており, ハードウェアの自動生成は行えていない．また, RTOS として TOPPERS/ASP3 のみを想定している．

本稿ではこの課題を解決するため, 文献 [7] の管理ハードウェアを自動生成する手法を提案する．本手法では, タスク数やタスクが利用する RTOS のサービス等のシステム構成情報をコンフィギュレーションファイルに記述し, その構成情報に応じた管理ハードウェアの RTL 記述を自動生成する．また, RTOS によって異なる仕様の情報をコンフィギュレーションファイルに記述することにより, 複数の RTOS に対応した管理ハードウェアの生成を可能にする．さらに, 使用されていない RTOS の機能を省くことにより回路規模の削減を図る．

本手法を用いて TOPPERS/ASP3, FreeRTOS のそれぞれの仕様に応じた機能を提供するハードウェアを自動生成することができた．タスク数 4, 8, 16 のシステムについてハードウェアを合成した結果, 回路規模と遅延はそれぞれ概ねタスク数とその対数に比例することが判明した．

2. RTOS を用いたシステムのフルハードウェア化

2.1 概念

文献 [5] は RTOS の機能とタスクおよびハンドラ全てをハードウェア化する手法を提案している．その概念を図 1 に示す．上図はソフトウェアシステムによる実装であり, タスク (TSKi) は RTOS の管理下で実行される．下図は上図のシステムを全てハードウェア化したものである．各 TSKi は独立したハードウェアモジュールとなり, RTOS の機能は管理ハードウェア (manager) が提供する．

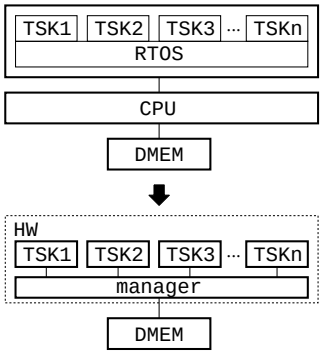

図 1: RTOS 利用システムのフルハードウェア実装 [5]

タスクは実行可能状態になれば全て並列に実行される．タスクの制御は管理ハードウェア (manager) が各タスクの状態から実行/停止の信号を出力することにより行う．この手法では, 各タスクモジュールは独立に並列に実行されるため, CPU 待ち, スケジューリングおよびコンテキストスイッチのオーバヘッドをなくせる．その上, タスクのハードウェア化により高速に処理が行えるため, 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．

2.2 前提とするアーキテクチャ

本稿では文献 [7] のアーキテクチャを前提とする．そのハードウェアの構成を図 2 に示す．T0∼T2 はタスクを, manager は RTOS 機能をハードウェア化したモジュールである．manager 内の STATUS レジスタは各タスクのステータス情報 (状態, 現在の優先度, ベース優先度, タイマー等) を, WAIT レジスタは各タスクのサービス待ちに関する情報を管理する．下部にある mutex, event flag 等は RTOS のサービス機能を提供するサービスモジュールである．shared_variable は共有変数の読み書きを, control_task はタスクの起動/休止や優先度の変更等のサービスを提供するモジュールである．サービス間の干渉を避けるため, サービスは同時に 1 つのみ実行される．Request Arbiter (RA) は複数のタスクからサービス要求があった場合に, タスクの優先度に基づいてその調停を行う調停回路である．

タスク (Ti) は実行可能状態になれば全て並列実行される．manager は STATUS レジスタ内の各タスクの状態に基づいて各タスクを実行/停止する制御信号を出力する．

タスク (Ti) がサービスを要求する場合, タスクはレジスタ TFi にサービスの ID を, TAi に必要な引数を書き込んでサービス処理の完了を待つ．RA は優先度最大のタスク

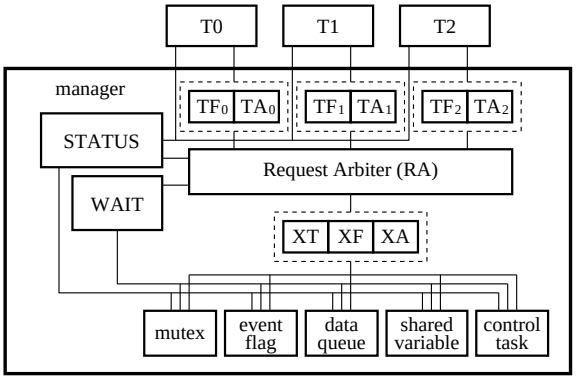

図 2: 文献 [7] のハードウェアの構成

番号を求め, レジスタ XT, XF, XA にそれぞれ優先度最大のタスク番号, TFi のサービスの ID, TAi の引数を書き込む. サービスモジュールは要求された処理を行い, 返り値を XA レジスタに書き込む. manager が XA の値を TAi にコピーしてタスクにサービスの完了を通知すると, タスクは TAi から返り値を読み取って自身の処理を再開する.

RA は比較回路のツリーで優先度最大のタスク番号を求めそのタスクの要求を XF, XA に書き込む.

タイムアウトの処理は 1 つのタイマーではなくタスク毎にタイマーを設け, それを manager が管理することにより行う. タスクがタイムアウト付きのサービスを要求すると, サービスモジュールはそのタスクのタイマーをセットする. タイマーを manager が毎クロックカウントダウンし, 0 になると manager がサービスモジュールにキャンセル処理を強制し, サービスモジュールはタイムアウトしたタスクの待ちを解除する.

RTOS の機能を提供するサービスモジュール (mutex, eventflag, dataqueue 等) は, システムで使用される複数のインスタンスを 1 つのモジュールで扱う. mutex モジュールの構成例を図 3 に示す. これはシステムに優先度上限プロトコルの mutex が 3 つ (ID=0, 1, 2) あり, 3 つのタスク (ID=0, 1, 2) が これらを使用している場合の例である. フラグの配列 lock は, どのタスクがどの mutex をロックしているかを表す. max priority はタスクが獲得している mutex の優先度上限の最大値を求めるテーブルである. タスクから mutex の獲得要求 (lock) があると, 当該 mutex が空いている場合はフラグをセットするとともにそのタスクが獲得している mutex の集合からテーブルで最大の優先度上限を求め, その値でタスクのカレント優先度を更新して正常終了を表す返り値を返す. mutex がすでに他のタスクによって獲得されている場合には, この要求を出したタスクを待ち状態にする. mutex の解放要求 (unlock) があった場合には, フラグのリセットとタスクのカレント優先度更新を行い, 当該 mutex を待っているタスクがあればその待ち解除処理を行う.

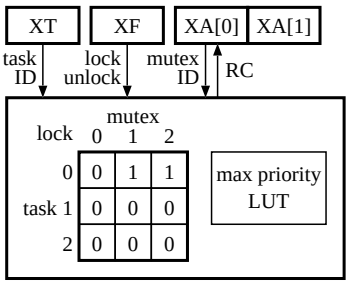

図 3: mutex モジュールの構成例

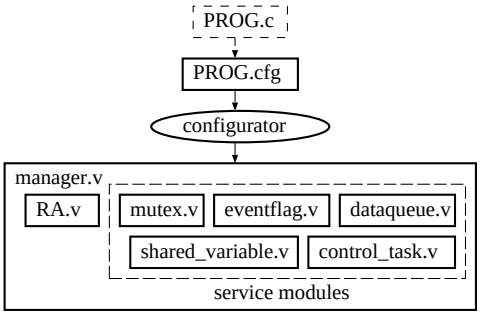

図 4: システムの自動生成の流れ

3. 管理ハードウェアの自動生成

3.1 概要

本稿では, 設計するシステムの構成情報から文献 [7] のアーキテクチャにおける manager モジュールを自動生成する.

コンフィギュレーションファイル内にシステムのタスク数およびタスクが利用する RTOS のサービスモジュール等の情報を記述し, そこから必要な数のポートとサービス機能を提供する manager の回路を生成する. RTOS によって異なるタスクの優先度の表現, エラーコード, サービスの仕様等をコンフィギュレーションファイルに記述することにより, 複数 RTOS に対応可能とする. また, 必要な RTOS のサービス機能のみを生成することにより回路規模の削減を図る.

本稿のハードウェア生成の流れを図 4 に示す. PROG.c は設計するリアルタイムシステムのソースコードであり, PROG.cfg はそこから抽出したシステムの構成情報を記述したコンフィギュレーションファイルである. configurator はコンフィギュレーションファイルの情報から管理ハードウェア (manager.v) とサブモジュールである調停回路 (RA.v) およびサービスを実行するモジュール (mutex.v, ..., shared_variable.v) を生成する. 現時点ではコンフィギュレーションファイルは手動で記述するが, 将来的にはソースコードを解析して生成する構想である.

3.2 システムの構成に応じた管理ハードウェアの生成

設計するシステムのタスク数とタスクが利用する RTOS のサービスに応じて必要となるポート, RA モジュール, サービスモジュールおよび STATUS レジスタや WAIT モジュールを生成する. コンフィギュレーションファイルにはこのために必要になる以下の情報を記述する.

- タスクの数
- タスクが利用するサービスとそのインスタンス毎の属性

コンフィギュレーションの記述例を図 5 に示す. 2 行目の tasks はタスク数で, 3 行目の service_modules は使用されるサービスモジュールのリストである. それぞれのサービスモジュールについて, そのモジュールあるいはサービスのインスタンス毎に必要な属性情報を記述する. 例えば, 4~8 行目はシステムで使用される 2 つの mutex の情報である. 5 行目は mutex の優先度変更プロトコルであり, 6, 7 行目はインスタンス毎の ID, 使用するサービスコール, mutex を利用するタスク等の情報を記述している.

使用するタスク数とサービスモジュールの情報から数に対応したポートや RA の回路を生成する. また, タスク数とサービスモジュール数の情報から対応したサイズの STATUS レジスタ, WAIT モジュール内のレジスタを生成する.

サービスモジュールに関しては, 使用するタスク, インスタンスの数および使用されるサービスコールでレジスタのサイズや個数と実装する処理が決まる. 例えば mutex では, mutex の優先度変更プロトコル, インスタンス数, およびそれを使用するタスクのリストから, 必要なビット数のフラグレジスタとルックアップテーブルを生成し, タスクが利用するサービスコールの情報からサービス (ロック, アンロック, タイムアウト) の処理を生成する.

```
1  system => {
2    tasks => 3,
3    service_modules => [
4      mutexes => [
5        protocol => ceiling
6        {id => 0, services => [loc_mtx, unl_mtx,
             tloc_mtx], tasks => [0, 2], ceiling
             => 2},
7        {id => 1, services => [loc_mtx, unl_mtx
             ], tasks => [0, 1, 2], ceiling => 1}
8      ],
9      eventflags => {
10       {id => 0, services => [set_flg, wai_flg,
              clr_flg], and_or => 1, clear => 0}
11     },
12     dataqueues => {
13       {id => 0, services => [snd_dtq, trcv_dtq
             ], data => 10}
14     },
15     shared_variables => {
16       {services => [read, write], mem => 32}
17     },
18     control_task => {
19       services => [wup_tsk, sus_tsk, get_pri,
             chg_pri]
20     }
21   ]
22   ...
23 }
```

図 5: コンフィギュレーションの記述例 (PROG.cfg)

3.3 複数の RTOS への対応

本稿では, RTOS によって異なる仕様の情報をコンフィギュレーションファイルに記述することによって複数の RTOS に対応する.

(1) タスクの優先度

一般にタスクの優先度は整数値で表現されるが, 優先度が数値の昇順であるか降順であるか, および優先度の上限値と下限値は RTOS によって異なる. 例えば, TOPPERS/ASP3 の場合, 優先度は数値が小さいほど高く, 優先度の取り得る値は 1 から 16 であるのに対し, FreeRTOS の場合は優先度は数値が大きいほど高く, 優先度の取り得る値は 0 から 31 である.

本稿では, 優先度最大を表す値 (MAX_PRI) と優先度最小を表す値 (MIN_PRI) をコンフィギュレーションファイルに記述し, その情報から RA の比較回路と優先度を扱うのに必要なビット数のレジスタおよび配線を生成する. MAX_PRI が MIN_PRI より小さい場合は, 優先度が数値の昇順であり, 逆の場合は降順であると解釈する.

図 6(a) は TOPPERS/ASP3 用の記述例であり, 11 行目が優先度の情報である. 優先度最大は 1, 最小は 16 であり, 数値が小さいほど優先度が高い. (b) の FreeRTOS 用の記述例も同様である.

優先度で調停を行う RA モジュールは優先度が昇順か降順で比較回路の大小順を決定して調停回路を生成する. また, RA に入力する優先度のビット数は優先度の取り得る値から決定する.

(2) サービスの内容

同じサービスであっても具体的な仕様は RTOS によって異なることがある. 例えば, mutex の場合 TOPPERS/ASP3 は優先度上限プロトコルを採用しているのに対して, FreeRTOS は優先度継承プロトコルを採用している.

使用するプロトコル名をサービスモジュールの情報に記述することによりプロトコルに合ったハードウェア記述を生成する.

例えば, TOPPERS/ASP3 用の記述例で図 6(a) では, 5 行目でプロトコルが ceiling であることを記述し, 6 行目と 7 行目で各インスタンスの優先度上限の値を記述している. (b) の FreeRTOS 用の記述例も同様である. これらの情報から protocol が ceiling の場合, 優先度上限プロトコル, inheritance の場合, 優先度継承プロトコルの mutex モ

ジュールの RTL 記述を生成する.

(3) エラーコード

サービスに対するエラコードも RTOS 毎に異なる.

本稿では, サービスモジュールがリターンコードを返す場合ごとにオリジナルネームを付け, それに対応するエラーコードをコンフィギュレーションファイルに記述することにより対応を行う.

例えば, TOPPERS/ASP3 用の図 6(a) では, 12 行目以降に mutex モジュールに対するエラーコードの記述例を示す. (b) の FreeRTOS 用も同様である. 例えば, mutex_unlock_ok は mutex のアンロックの正常終了, mutex_other_locked_unlock_error は他のタスクがロックしている mutex をアンロックした時のエラー処理に対するオリジナルネームである. 各オリジナルネームに対して, TOPPERS/ASP3 であれば E_OK や E_OACV 等, FreeRTOS であれば pdTRUE, pdFALSE 等のマクロ定数を記述することにより複数の RTOS に対応する.

3.4 最小限の機能の生成

本稿では, サービスモジュールの内の実装に不要な処理を省き必要最小限の機能を生成する. 不要な処理が発生する場面は以下が想定される.

- タイムアウトが不要な場合のタイムアウト処理
- タスクが使用しないサービスコールの処理

これらに対して, 最小限の機能を生成することにより回路規模削減を図る.

タイムアウト処理に対してはタイムアウトが必要な場合のみタイムアウト処理を生成し, 不要な場合は生成しないことにより最小限の機能を生成する. 例えば, 図 5 の 12~14 行目はデータキューの情報である. データキューの services である snd_dtq はデータ送信のサービスであるため, タイムアウトの処理は生成しない. trcv_dtq はタイムアウト付きのデータ受信のサービスであるため, タイムアウト処理を生成する.

タスクが使用しないサービスコールの処理に対する RTL 記述を省くことにより回路規模の削減を図る. 例えば, 図 5 の 18~20 行目のコントロールタスクに実装するサービスコール (wup_tsk, sus_tsk, get_pri, chg_pri) が services に記述されているので, これらの実行に必要な RTL 記述のみを生成する.

4. 実装と実験

提案手法に基づいて, 管理ハードウェアを生成するコンフィギュレータを Perl5 で実装した. 生成された Verilog HDL を Xilinx 社の Vivado 2016.4 を用いて FPGA (Artix-7) をターゲットに論理合成行った.

コンフィギュレータを用いて TOPPERS/ASP3 および FreeRTOS の仕様に応じた manager を生成した. 生成した manager モジュールの詳細は以下である.

- タスク数 4 つ
- data queue は $32B \times 10$ のデータ領域を保持
- shared_variable は $32B \times 32$ ワードメモリを保持
- control_task がサポートしたサービスコールは TOP-

```
1  system => {
2    tasks => 3,
3    service_modules => [
4      mutexes => [
5        protocol => ceiling
6        {id => 0, services => [loc_mtx, unl_mtx,
              tloc_mtx], tasks => [0, 2], ceiling
              => 2},
7        {id => 1, services => [loc_mtx, unl_mtx
              ], tasks => [0, 1, 2], ceiling => 1}
8      ]
9      ...
10   ],
11   priority => {MAX_PRI => 1, MIN_PRI => 16},
12   error_code => [
13       {mutex_unlock_ok => 'E_OK'},
14       {mutex_other_locked_unlock_error => '
             E_OACV'},
15       {mutex_unlocked_unlock_error => 'E_OACV'
             },
16       {mutex_timeout => 'E_TMOUT'},
17       {mutex_lock_ok => 'E_OK'},
18       {mutex_locked_lock_error => 'E_OBJ'},
19      ...
20   ]
21 }
```

(a) TOPPERS/ASP3

```
1  system => {
2    tasks => 3,
3    service_modules => [
4      mutexes => [
5        protocol => inheritance
6        {id => 0, services => [loc_mtx, unl_mtx,
              tloc_mtx], tasks=>[0, 2]},
7        {id => 1, services => [loc_mtx, unl_mtx
              ], tasks=>[0, 1, 2]}
8      ]
9      ...
10   ],
11   priority => {MAX_PRI => 31, MIN_PRI => 0},
12   error_code => [
13       {mutex_unlock_ok => 'pdTRUE'},
14       {mutex_other_locked_unlock_error => '
             pdFALSE'},
15       {mutex_unlocked_unlock_error => 'pdFALSE
             '},
16       {mutex_timeout => 'pdFALSE'},
17       {mutex_lock_ok => 'pdTRUE'},
18       {mutex_locked_lock_error => 'pdFALSE'},
19      ...
20   ]
21 }
```

(b) FreeRTOS

図 6: RTOS の差異の記述例

表 1: manager の回路規模とクリティカルパス遅延 (タスク数 4)

RTOS	#LUT	#FF	delay [ns]
TOPPERS/ASP3	3243	2039	7.581
FreeRTOS	3431	2042	7.693

表 2: タスク数を変化させた場合の合成結果

#TASK	#LUT	#FF	delay [ns]
4	4463	2188	7.449
8	7970	3812	9.121
16	14662	7110	12.155

PERS/ASP3 では wup_tsk, sus_tsk, get_pri, chg_pri の 4 つ, FreeRTOS では, xTaskResume, vTaskSuspend, vTaskPriorityGet, vTaskPrioritySet の 4 つである.

生成した manager モジュールの回路規模とクリティカルパス遅延を表 1 に示す. 回路規模を表す #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. 両 RTOS 仕様で生成したサービス処理やタスク数が同じであるため, 回路規模とクリティカルパス遅延は同程度になった.

本手法を用いてタスク数 4, 8, 16 について TOPPERS/ASP3 仕様の manager を生成した. その manager 内のサービスモジュールは以下の通りとした.

- mutex はインスタンス 2 つ
- event flag はインスタンス 2 つ
- data queue は $32B \times 10$ のデータ領域を保持
- shared_variable は $32B \times 32$ ワードメモリを保持
- control_task がサポートするサービスコールは以下の 12 である.

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

生成した manager モジュールの回路規模とクリティカルパス遅延を表 2 に示す. 回路規模と遅延はそれぞれ概ねタスク数とその対数に比例して増加している. タスク数に対応したポート, サービス処理, レジスタを生成するため, 回路規模はタスク数に比例して増加したと考える. また, クリティカルパス遅延の増加は RA の比較回路のツリーの段数の増加によるものと考える. タスク数 16 では回路規模, クリティカルパス遅延とも実用には少し大きく, その削減が必要と考えれる.

5. おわりに

本稿では, システムの構成情報に対応した管理ハードウェアを生成する手法を提案した. RTOS によって異なる仕様の情報をコンフィギュレーションファイルに記述することにより複数の RTOS に対応するとともに, 必要な RTOS の機能のみを生成することにより回路規模の削減を図った.

本手法に基づき, TOPPERS/ASP3 および FreeRTOS 仕様の manager を生成することができた. また, タスク数 4, 8, 16 の manager モジュールの回路規模とクリティカルパス遅延はそれぞれ概ねタスク数とその対数に比例することが判明した.

管理ハードウェアの自動生成によって, タスク数に対する回路規模やクリティカルパス遅延が判明したため, ボトルネックの分析や回路の最適化が次の課題として挙がられる. また, 現時点ではコンフィギュレーションファイルは人手で記述しているため, プログラムのソースコードを解析してコンフィギュレーションファイルを生成するトランスレータの実装が今後の課題として挙げられる.

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. 本研究の一部 JSPS 科研費 19H04081, 20H00590, 21K19776 の助成による.

参考文献

[1] P. Kohout, B. Ganesh, and B. Jacob: “Hardware support for real-time operating systems,” in *Proc. International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS ’03)*, pp. 45–51 (Oct. 2003).

[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. International Symposium on Rapid System Prototyping (RSP 2006)*, pp. 163–168 (June 2006).

[3] 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).

[4] 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).

[5] 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).

[6] 六車, 石浦, 安堂, 冨山, 神原: “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,” 信学技報, VLD2020-75 (Mar. 2021).

[7] 安堂, 石井, 石浦, 冨山, 神原:“RTOS 利用システムの汎用高位合成系を用いたフルハードウェア化,” 信学技報, VLD2021-51 (Jan. 2022).