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

信学技報
TECHNICAL REPORT OF IEICE.

高位合成システムCCAPの AMPマルチコアシステム設計のための拡張

石守　祥之† 　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
†† 名古屋大学 大学院情報科学研究科　〒464–8603 名古屋市千種区不老町
††† 京都高度技術研究所　〒600–8813 京都府京都市下京区中堂寺南町 134
E-mail: †ishimori@ksc.kwansei.ac.jp

あらまし　我々は，C プログラムを入力として，指定した関数をソフトウェアから呼出し可能なハードウェアに合成する高位合成システム CCAP (C Compatible Architecture Prototyper) の開発を行っている．本稿は，CCAP の AMP マルチコアシステム設計のための拡張を提案する．本手法では，ANSI-C で記述した逐次プログラムに対し，専用のプラグマによって関数単位でのコアへの割当てや関数の呼び出し方の制御を指定する．これまでの CCAP では，１つの関数に対して１つのハードウェアを合成していたが，本研究の拡張では，複数の関数を１つのハードウェアに合成すること，および１つの関数を実行するハードウェアを複数複製する構成も可能である．並列実行は非同期関数呼び出しの指定により行う．排他制御処理を行うことにより関数呼び出しを多段に行うことが可能である．本稿では，AMP マルチコアシステムのプラグマによる設計記述とその合成系の実装法について述べる．
キーワード　高位合成，AMP，非同期関数呼び出し

Extension of High Level Synthesis System CCAP for AMP Multi-Core System Design

Yoshiyuki ISHIMORI†, Nagisa ISHIURA†, Hiroyuki TOMIYAMA††, and Hiroyuki KANBARA†††

† Kwansei Gakuin University　2–1 Gakuen，Sanda，Hyogo, 669–1337 Japan
†† Nagoya University　Furo–cho，Chikusa–ku，Nagoya，Aichi，464–8603 Japan
††† ASTEM　134，Chudojiminami–cho，Shimogyo–ku，Kyoto–shi，Kyoto，600–8813 Japan
E-mail: †ishimori@ksc.kwansei.ac.jp

Abstract　We are developing a high-level synthesis system named CCAP (C Compatible Architecture Prototyper), which synthesizes functions in C programs into hardware callable from the other functions running on CPU as software. This article presents an extension of CCAP for AMP Multi-Core system design. We augment a given sequential ANSI-C program with dedicated pragmas to specify assignment of the functions to cores and how the calls to each function are implemented. While CCAP so far synthesizes a single hardware instance for each function, this extension allows multiple functions to be synthesized into a single hardware instance or a single function to be executed on multiple copies of a hardware module. Parallel execution is realized in terms of asynchronous remote procedure calls. Mutual exclusion control is automatically implemented to allow multilevel asynchronous calls. This article describes the pragmas for AMP multi-core design specification and how the systems are implemented from the specification.
Key words　High level synthesis, AMP, Asynchronous function call

1. はじめに

近年，VLSI の回路規模や複雑度は増加の一途をたどっており，その設計効率の向上は重要な課題となっている．高位合成技術は，ハードウェアの動作レベルの記述から自動的にレジスタ転送レベルの設計記述を生成する技術であり，設計効率化の課題の解決策の一つとして注目を集めている．入力となる動作記述には，ハードウェア記述言語 (VHDL，Verilog HDL 等) やシステム記述言語 (SpecC，SystemC 等) の他に，ソフトウェア資産の流用や言語習熟者数等の観点から C/C++ 等のプロ

グラミング言語が用いられるようになっている．

我々は，C プログラムを入力とし，その中の指定した関数をハードウェアに合成する高位合成システム CCAP（C Compatible Architecture Prototyper）[1] の開発を行っている．CCAP は，動作記述全体を単体のハードウェアに合成するのではなく，指定した関数を CPU で実行される他の関数から起動可能なハードウェアに合成する点を特長としている．現在の CCAP が合成するシステムの構成はハードウェアが複数で CPU が 1 つであるシングルコアのシステムである．

近年では 1 チップに複数のプロセッサコアや専用ハードウェアを搭載したマルチコア技術の開発が進んでいる．マルチコア技術は処理の高速化のみならず低消費電力化やリアルタイム処理の実装にも適しており，組み込みシステムへの適用が拡大している．特に，タスクによって分担するコアをあらかじめ決定しておく AMP（Asymmetric Multi-Processor）方式は，組込みシステムにおいて多く採用されている [2]．動作記述からマルチコアシステムを設計する手法やマルチコア技術を有効に利用するためのプログラミング手法等の研究も盛んに行われている [3] [5] [6] [7].

本研究は，高位合成システム CCAP の AMP マルチコアシステムの設計のための拡張を提案する．本研究の動作記述として，ANSI-C で記述した逐次プログラムを用いるが，並列実行が可能な部分をプラグマで指定することにより，関数を並列に実行することが可能である．並列実行は，関数呼び出しの排他制御を行うことで，多段に行うことも可能である．本研究では，プラグマを用いて，非同期関数呼び出し及び排他制御処理の有無，関数の多重化，及び関数のプロセッサ/ハードウェアへの割り当てを実現する．本論文では，この AMP マルチコアシステム設計の実現手法について述べる．

2. 関 連 研 究

マルチコアシステムには，タスクをプロセッサコアに均一に割り当てて負荷分散を行う SMP (Symmetric Multi-Processor) 方式と，プロセッサコア毎に役割が決まっており，タスクに応じて特定のプロセッサコアを用いて機能分散を行う AMP 方式がある．AMP 方式は，異種のコアから構成されるヘテロジニアスマルチコアと同種のコアから構成されるホモジニアスマルチコアの 2 種類に分類される．

動作記述からマルチコアシステムを設計する手法やマルチコア技術を有効に利用するためのプログラミング手法の研究には以下のようなものがある．

AMP マルチコア・プログラミングのフレームワークとして, [6] ではプロセス間通信に RPC（Remote Procedure Call）ベースの API を用いたソフトウェア開発手法を提案している．この研究では，元となる逐次プログラムに対し，関数の呼び出しを RPC に書き換えることにより，マルチコアによる並列処理を記述する．実行するコアを動的に割り付ける動的負荷分散機能をスケジューラに搭載しており，それぞれの関数は実行するべきコアを意識する必要がない．しかし，関数の呼び出しに排他制御処理が行われておらず，リモートプロシージャの呼び出しは一段に制限されている．

文献 [3] では，高位合成ツールと連携したハードウェア/ソフトウェア設計空間の探索が可能なマルチプロセッサ SoC（MPSoC）設計のフレームワークを提案している．入力としては，システム機能分割を単純な構文で記述した C による動作記述を用いる．ここではスレッドと呼ばれる単位で機能分割を行い，コンパイラが分割された並列スレッド間のデータ通信同期命令を自動挿入することを特長とする．このフレームワークを用いて合成されるシステムは分散メモリ型のマルチプロセッサモデルである．

SMP におけるスレッド並列化のためのディレクティブの共通規格として，OpenMP [4] がある．OpenMP では既存の逐次プログラムに，プラグマを用いて並列処理が可能な部分を指定する. [5] では，OpenMP を用いた動作合成手法を提案している．この研究では，OpenMP の section，for 等のディレクティブで並列化を指示した部分を並列動作する複数のハードウェアに合成する．SMP 上でシミュレーションを行うため，RTL シミュレーションと比較して並列化の効果の評価を高速に行うことが可能である．

SystemBuilder [7] は，C 言語及びシステム定義ファイル，または SpecC 言語を入力として，ソフトウェアとハードウェア及びその間のインターフェースからなるシステムを自動合成する．入力記述は，RTOS 上で並行に動作する複数のプロセスからなるプログラミングモデルであり，ソフトウェア/ハードウェア間のインタフェースの設計を考慮することなく設計が可能である点を特長としている．

本研究で提案する設計手法は，ANSI-C で記述した動作記述にプラグマを挿入することにより，ヘテロジニアス AMP マルチコアシステムを設計する手法である．プラグマにより，関数のプロセッサ/ハードウェアへの割り当て，非同期関数呼び出し，排他制御処理の有無，及び関数の多重化を指定する．排他制御を実装しているため，非同期呼び出し時に関数呼び出しを多段に行うことができる．また，システムの設計には OS のサポートを要しない．

3. 高位合成システム CCAP

3.1 CCAP の合成するシステム

図 1 に（拡張前の）CCAP が合成するシステムの構成を示す．CCAP は，入力となる ANSI-C プログラム中のいくつかの関数 (設計者が指定する) をハードウェアに合成する．合成したハードウェア関数は，ソフトウェア として CPU で実行される他の関数から呼び出すことが可能である．但し，再帰呼び出しは扱わないものとする．CPU とハードウェア関数はメモリ空間を共有し，主記憶上に配置するグローバル変数へのアクセスにより，データや制御の受け渡しを行う．各デバイスと主記憶の間にはアービタが介在しており，主記憶へのアクセス要求を調停する．

3.2 CCAP におけるハードウェア関数の呼び出し

CCAP における関数の呼び出しは，制御やデータの受け渡しのために主記憶上に確保したグローバル変数へのアクセスに

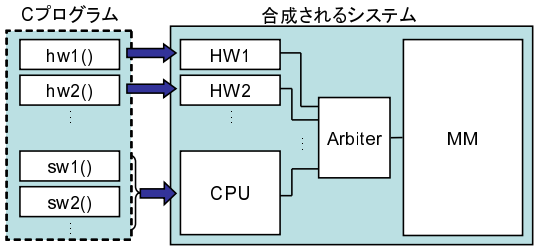
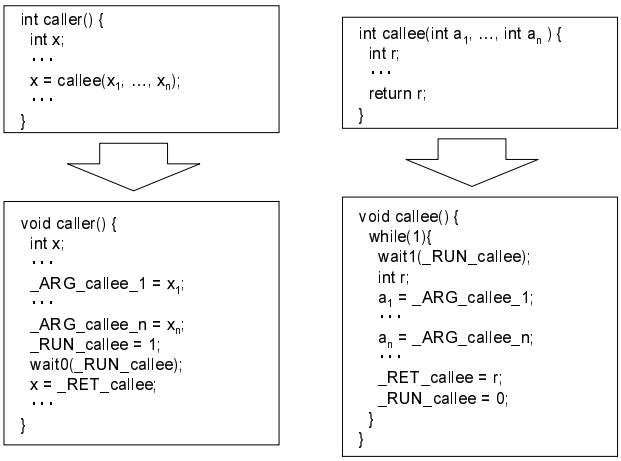


図 1　拡張前の CCAP が合成するシステム

変換して実現する．現時点では，関数の起動の制御はグローバル変数のポーリングによる実現のみをサポートしている．



(a) 呼び出し側の変換　　(b) 呼び出される側の変換

図 2　CCAP における関数呼出し

図 2 はソフトウェア関数 caller からハードウェア関数 callee を呼び出しの変換の例を C プログラムで表現したものである．

図 2(a) のように caller はまず引数 x_1，...，x_n の値を引数受渡し用グローバル変数 _ARG_callee_1，...，_ARG_callee_n に代入する．続いて実行制御用グローバル変数 _RUN_callee に 1 を代入すると，callee が処理を開始する．_RUN_callee の値は callee の処理が終了すると 0 になるので，caller はその値を監視しながら待機する（wait0）．その後，返り値用グローバル変数 _RET_callee から返り値を受け取り，残りの処理を続行する．

一方 callee 側では，図 2(b) のように，まず_RUN_callee を監視しながら，その値が 1 になるまで待機する．その後，_ARG_callee_1，...，_ARG_callee_n から引数を受け取り，関数本体の処理を実行する．処理が終了したら，_RET_callee に返り値を書き込み，_RUN_callee の値を 0 にして，処理が終了したことを caller に伝える．その後，再び _RUN_callee の監視に戻る．

4.　高位合成システム CCAP の拡張

4.1　本研究の特長

本研究では，AMP マルチコアシステム設計が可能となるよう CCAP の拡張を行う．コアとしては，複数種類の CPU および高位合成により合成したハードウェアを想定している．

動作記述は ANSI-C による逐次的なプログラムを基に，関数単位での処理するプロセッサやハードウェアへの割り当て，及び非同期での関数呼び出し，排他制御の有無等をプラグマを用いて指定する．

本研究の特長としては以下が挙げられる．

- 関数とハードウェアの多対多対応

拡張前の CCAP ではプロセッサは 1 つであり，また 1 つの関数に対し 1 つのハードウェアを合成していた．本研究ではプロセッサは複数あってもよく，関数とハードウェアは一対一に対応していなくても良い．即ち，複数関数を 1 つのハードウェアに合成することや，1 つの関数のハードウェアを複数に合成し，処理の多重化（並列処理）を行うことも可能である．

- 非同期関数呼び出し及び排他制御

処理の並列化は [6] 同様，非同期呼び出しにより実現する．ただし，非同期呼び出しを多岐に行った時のコア資源の競合を回避するために，関数呼び出しに対する排他制御処理を行う．

- シミュレーション

動作記述の検証のためのシミュレーションをマルチスレッドを用いて SMP 環境で高速に行うことができる．

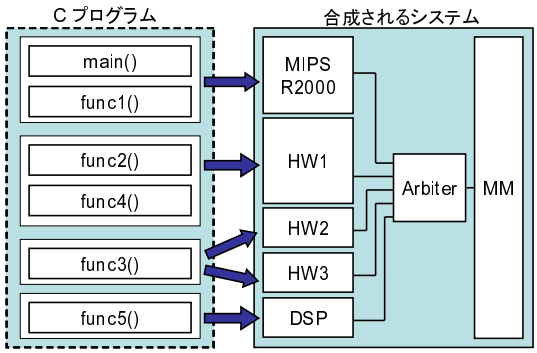


図 3　拡張した CCAP が合成するシステムの構成の例

4.2　セクション分割

拡張した CCAP が合成する AMP マルチコアシステムの構成の例を図 3 に示す．

ここでは，関数 main，func1 は MIPS R2000 で，関数 func5 は DSP で実行する．HW1 は合成されたハードウェアであるが，関数 func2 と関数 func4 を実行する．拡張後の CCAP は一つのハードウェアで複数の関数を実行することが可能である．

また，CCAP では「関数の多重化」が可能である．関数の多重化とは，関数を複製して，それぞれを別々のハードウェアに合成，あるいはプロセッサコアに割り当てることをいう．図 3 では，関数 func3 を多重化してそれぞれを別のハードウェア HW2，HW3 に合成する．

関数と合成するハードウェアの対応はプラグマで指定する．図 3 のシステムを設計するためのプラグマの記述例を図 4 に示す．本研究では，1 つのコア又はハードウェアに割り当てる関数群を「セクション」と呼ぶ．プラグマの中には，割り当てるコアの種類，セクション名，そのセクション内の関数が呼び出す他のセクションに属する関数を記述する．図 4 の 1 行目のプラグマは，割り当てるプロセッサコアは MIPS，セクション名

```
 1: #pragma ccap section(MIPSR2000 main)calls(func2,func5)
 2: int main(void) {
 3:    …
 4: }
 5: int func1(){
 6:    …
 7: }
 8: #pragma ccap section(HW h1) calls(func3)
 9: int func2(){
10:    …
11: }
12: #pragma ccap section(HW h2,HW h3)
13: int func3(){
14:    …
15: }
16: #pragma ccap section(HW h1) calls(func3)
17: int func4(){
18:    …
19: }
20: #pragma ccap section(DSP d4)
21: int func5(){
22:    …
23: }
```

図 4　セクション分割用プラグマの記述例

```
1: int func4(){
2:    …
3:   #pragma ccap call (h2)
4:   r1 = func3();
5:    …
6: }
```

図 5　多重化した関数の呼び出し

は main，セクション main に属する関数が呼び出す他のセクションの関数には関数 func2，func5 があることを表している．

ハードウェアに合成する場合には 5 行目のプラグマ内でコアの種類を「HW」と記述する．図 4 では，関数 func2，func3，及び func4 をハードウェアに合成するが，関数 func2 と関数 func4 は 1 つのハードウェアに合成するため，同一のセクション名 h1 となる．

呼び出す関数を多重化する場合には，いずれのセクションに属する関数を呼び出すのかをプラグマで指定する必要がある．図 5 は図 4 において関数 func 4 が多重化した関数 func3 の呼び出しにおいて，セクション h2 に属する関数 func3 を呼び出す場合の記述例である．

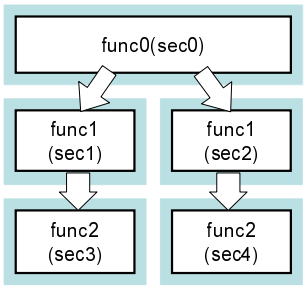


図 6　呼び出す関数自身が複製されている場合

呼び出す関数自身が多重化されている場合には，多重化した関数毎に呼び出す関数を指定しなければならない．例えば，図 6 の関数 func1 が関数 func2 を呼び出す場合の動作記述の記述例を図 7 に示す．2 行目では，セクション sec1 に属する

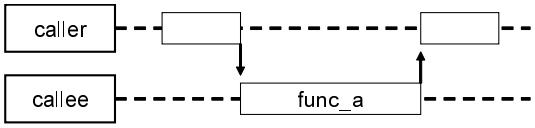
```
1: int func1(){
2:    …
3:   #pragma ccap call (sec1 => sec3, sec2 => sec4)
4:   r1 = func2();
5:    …
6: }
```

図 7　呼び出す関数自身が複製されている場合の関数呼び出し

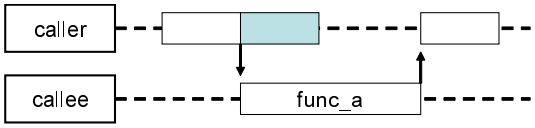
関数 func1 はセクション sec3 に属する関数 func2 をセクション sec2 に属する関数 func2 はセクション sec3 に属する関数 func2 を呼び出すことを指示している．

4.3　非同期関数呼出し

図 8(a) に従来の CCAP の関数呼び出しを示す．従来の CCAP では，関数呼び出しに同期呼び出しを採用しており，呼出し側の関数は呼び出された関数の実行が終了するまで待機していた．本研究では，非同期関数呼出しにより呼び出し側の関数と呼び出された関数の並列実行を可能となるよう拡張を行う．図 8(b) に非同期での関数呼出しの例を示す．非同期関数呼出しとは，呼び出す側の関数 caller が呼び出された関数 callee の実行の終了を待たずに，実行を継続することで関数の並列実行を行うことをいう．



(a) 従来の CCAP の関数呼出し



(b) 非同期関数呼出し

図 8　CCAP の関数呼出し

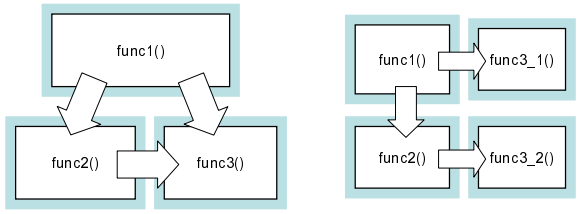
関数の非同期呼出しはプラグマにより指示する．入力記述例を図 9 に示す．6 行目のプラグマは次の行の関数呼び出しが非同期呼び出しであることを示し，呼び出された関数の完了待ち合わせを行う部分を 12 行目のプラグマで指示する．この例では，8 行目から 11 行目までの処理が関数 func1 の実行と並行して行われる．

```
 1: …
 2: int main(void) {
 3:   int a, b, r1;
 4:   int x, y, z, r2, r3;
 5:    …
 6:   #pragma ccap call async_begin func1
 7:   r1 = func1(a, b);
 8:   x = a + 1;
 9:   y = x * 2;
10:   z = 2 * (b + 1);
11:   r2 = func2(x, y, z);
12:   #pragma ccap call async_end func1
13:   r3 = func3(r1,r2);
14:    …
15: }
16: …
```

図 9　非同期関数呼出しの入力プログラム

4.4 排他制御

非同期関数呼出しを許可すると，1 つの関数が複数の関数から同時に呼び出される可能性が生じる．この例を図 10(a) に示す．この例では，関数 func3 は関数 func1，func2 の双方から同時に呼び出される場合が生じる．この時，関数 func3 の呼出しに対して排他制御を行わなければ正しい実行結果を保証することができない．



(a) 関数の多段呼出し　(b) 関数の多重化

図 10　排他制御を要する関数呼出し関係

本研究では，これを関数の多重化による回避又は関数の実行権取得による排他制御 [8] により解決する．関数の多重化による回避を図 10(b) に示す．この例では，関数 func3 を多重化することにより関数 func3 への重複した呼出し関係を解消している．実行権取得による排他制御では，関数の呼出し時にその関数の実行権を先に取得した関数が先にその関数を実行する（取得できなかった関数は待たされる）．実現手法は後述する．

本研究では全ての関数呼び出しに対して排他制御処理を行うことをデフォルトとする．しかし，セクションの数が 2 つの場合等関数呼び出しに排他制御が不要であることが自明である場合がある．その場合，プラグマにおいてそのセクション内の関数の呼び出しには排他制御が不要であることを記述する．図 11 は排他制御が不要なセクションの定義を行うプラグマの記述例である．この例では，セクション h1 内の関数 func1 の呼び出しに際しては，排他制御処理が不要であることを表す．

4.5 シミュレーション

動作記述が機能的に正しいかを検証するため，本研究では SMP 環境下においてシミュレーションを行う．シミュレーションでは，文献 [9] の手法を用いる．セクション毎に 1 スレッドを割り当て，スレッド間通信は，CCAP 同様にグローバル変数を用いて行う．動作記述に基づき，スレッドを同期呼び出し，非同期呼び出しにより実行制御を行うことで SMP マルチコア環境下においてシミュレーションを行うことができる．

5. 実　現　法

5.1 同期関数呼び出し

CCAP におけるソフトウェア/ハードウェア相互間の通信は通信用スタブ関数を生成し，その関数内でグローバル変数等を扱うことにより行う．

スタブ関数は関数呼び出し用と待機用の 2 種類を用意する．図 3 の関数 func2 を呼び出すために生成される通信用スタブ関数を図 12(a) に，関数 func2 を含むセクション h1 で実行の開始を待つためのスタブ関数の例を (b) に示す．

拡張前の CCAP では，関数毎に実行制御用グローバル変数を用意していたが，本研究では，各コアに 1 つ実行制御用グローバル変数を用意する．各コア内にてどの関数を実行するかは実行制御用グローバル変数の値を参照して決定する．

この例では，呼び出し側は関数 func2 を実行するために関数呼び出し用スタブ関数にて _RUN_h1 に 1 を代入する．待機側は _RUN_h1 が 1 の場合には関数 func2 を，2 の場合には関数 func4 を実行するため，関数 func2 の実行を開始する．

```
1: int func2(int a, int b){
2:   _ARG_func2_1=a;
3:   _ARG_func2_2=b;
4:   _RUN_h1=1;
5:   while(_RUN_h1){}
6:   int ret = _RET_func2;
7:   return ret;
8: }
```

(a) 呼び出し用スタブ関数

```
 1: int wait_stab(void){
 2:   while(1){
 3:     if( _RUN_h1 == 1 ){
 4:       _RET_func2 = func2();
 5:       _RUN_h1 = 0;
 6:     } else if(_RUN_h1 == 2 ){
 7:       _RET_func4 = func4();
 8:       _RUN_h1 = 0;
 9:     }
10:   }
11: }
```

(b) 待機用スタブ関数

図 12　通信用スタブ関数スタブ関数

5.2 排他制御

排他制御を要する場合には，文献 [8] の手法を用いる．図 3 の関数 func2 の呼び出しに排他制御を要する場合の通信用スタブ関数を図 13 に示す．実行権の取得には，排他制御用グローバル変数　_SEM_h1 を用いる．まず排他制御用グローバル変数に自身のセクション ID 2 をセットして，関数の実行権を取得する．セクション ID が書き込まれた場合にそのセクション以外に _SEM_h1 を更新できないようアービタは調停を行う．実行権の取得が確認できれば，引数の受け渡し等の関数を実行する処理を行う．

5.3 非同期関数呼び出し

図 14 はセクション sec1 に属する関数 func1 を非同期に呼び出す場合に用いる関数呼び出し用スタブ関数を示している．非同期で関数を呼び出す場合には，呼び出す関数自身も実行を継続する必要がある．そこで，同期関数呼び出し時とは異なり，実行権の取得，引数の受け渡し及び実行制御を行う通信用スタ

```
1: …
2: #pragma ccap section(no_exclusive_control HW h1)
3: int func1(){
4:    …
5: }
6: …
```

図 11　排他制御不要な場合のプラグマの記述

```
 1:  int func1(int a, int b){
 2:    do {
 3:      while( _SEM_h1 == 1 ){}
 4:        _SEM_h1=2;
 5:    } while(_SEM_h1!=2);
 6:    _ARG_func2_1=a;
 7:    _ARG_func2_2=b;
 8:    _RUN_h1 = 1;
 9:    while(_RUN_h1){}
10:   int ret=_RET_func2;
11:   _SEM_h1=0;
12:   return ret;
13: }
```

図 13　排他制御を要する場合の通信用スタブ関数

```
 1:  int main(void){
 2:    …
 3:    func1_start(a, b);
 4:    x = a + 1;
 5:    y = x * 2;
 6:    z = 2 * (b + 1);
 7:    r2 = func2(x, y, z);
 8:    r1 = func1_end();
 9:    r3 = func3(r1,r2);
10:    …
11:  }
12:  //引数の受け渡しと実行制御用通信スタブ関数
13:  void func1_start(int a, int b){
14:    do{
15:      while(_SEM_sec1){}
16:      _SEM_sec1 = 1;
17:    }while(_SEM_sec1 == 1);
18:    _ARG_func1_1 = a;
19:    _ARG_func1_2 = b;
20:    _RUN_sec1 = 1;
21:  }
22:  //実行結果取得用通信スタブ関数
23:  int func1_end(void){
24:    while(_RUN_sec1){}
25:    return _RET_func1;
26:  }
```

図 14　非同期呼び出し用スタブ関数

ブ関数 func1_start と，呼び出した関数の実行終了を待ち，結果を受け取る通信用スタブ関数 func1_end を用意する．

6. むすび

本研究では，高位合成システム CCAP の AMP マルチコアシステム設計のための拡張を行う．ANSI-C で記述した逐次プログラムを動作記述に用い，排他制御処理により，並列実行が可能な部分をプラグマで指定すれば，関数を多段に呼び出すことができる．さらに関数とハードウェアの多対多対応や関数の多重化も可能となる．今後は処理系の実装を行い，実際のシステムに適用して評価を行う予定である．今後の課題としては，親セクションに属する関数の呼び出しの扱いがある．文献 [10] では，ソフトウェア関数からのハードウェア関数の呼び出し手法を提案しているが，そのまま実装することはできない．そのため，今後さらに検討していく必要がある．

謝辞 本研究に多大な貢献をして頂きました，関西学院大学（現ルネサスソリューションズ）の西村啓成氏及び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，科学研究費補助金 (課題番号 20500058, 19700040) の支援による．

文　献

[1] M. Nishimura, et al: “High-level synthesis of variable accesses and function calls in software compatible hardware synthesizer CCAP,” in *Proc. SASIMI 2006*, pp. 29–34 (April 2006).

[2] TOPPERS プロジェクト: TOPPERS Project Newsletter, 第 6 号 (April 2005).

[3] Arif Ullah Khan, Kawachi Yosuke, Tsuyoshi Isshiki, Dongju Li, and Hiroaki Kunieda: “Hybrid MPSoC Design Framework Based on Tightly-Coupled Thread Model,” 情処 *DA シンポジウム 2008*, pp. 49–54 (Aug. 2008).

[4] 牛島省: “OpneMP による並列プログラミングと数値計算法,” 丸善株式会社 (May 2006).

[5] 中谷嵩之，松崎裕樹，山崎勝弘: “OpenMP によるハードウェア動作合成システムの設計と検証,” *FIT 2007* (Sept. 2007).

[6] 須賀敦浩，鈴木貴久: “RPC 技術を利用した関数並列型マルチコアプログラミング,” Interface, 2007 年 12 月号，CQ 出版社 (Dec. 2007).

[7] 本田 晋也，冨山 宏之，高田 広章: “システムレベル設計環境 : SystemBuilder,” 信学論，Vol.J88-D-I，No.2，pp. 163–174 (Feb. 2005).

[8] 西村啓成: “ハードウェアの高位合成におけるソフトウェア関数の呼び出し及び非同期関数呼び出しの実現,” 関西学院大学大学院 理工学研究科 情報科学専攻 2008 年度修士論文 (Mar. 2008).

[9] 石守祥之，奥野裕，石浦菜岐佐，西村啓成，神原弘之，冨山宏之: “C プログラムと中間表現の実行比較に基づく高位合成システム CCAP のテスト,” 情処 関西支部 支部大会 *2006*, pp. 181–182 (Sept. 2006).

[10] M. Nishimura, et al: “High-Level Synthesis of Software Function Calls,” IEICE TRANS. FUNDAMENTALS, vol.E91-E，No.12，pp. 3556–3558 (Dec. 2008).