社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2005－SLDM－122 (26)
2005／12／2

ソフトウェア互換ハードウェアを合成する高位合成システム CCAP における変数と関数の扱い

西口 健一† 石浦菜岐佐† 西村 啓成† 神原 弘之†† 冨山 宏之†††
高務 祐哲†††† 小谷 学††††

† 関西学院大学理工学部 〒669-1337 兵庫県三田市学園 2-1
†† 京都高度技術研究所 〒600-8813 京都市下京区中堂寺南町 134
††† 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
†††† 京都大学大学院情報学研究科 〒606-8501 京都市左京区吉田本町
E-mail: †{scbc9034,ishiura,nishimura.m}@ksc.kwansei.ac.jp, ††kanbara@astem.or.jp,
†††tomiyama@is.nagoya-u.ac.jp, ††††{takatsukasa,kotani}@vlsi.kuee.kyoto-u.ac.jp

あらまし 我々は, ANSI-C のプログラム中の指定された関数を, 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高位合成システム CCAP (C Compatible Architecture Prototyper) を開発している. 合成したハードウェアは CPU とメモリ空間全体を共有することによりデータや制御の受け渡しを行う. このため, インターフェースを別途設計することなくソフトウェアからハードウェア化された関数を呼び出すことが可能である. また, ポインタを含むプログラムを自然な形でハードウェア化できるため, ソフトウェアが割り当てた配列や動的データにハードウェアがポインタを用いてアクセスすることも可能である. 本稿では, CCAP における変数の扱いと, グローバル変数を利用した関数呼び出しとそのスケジューリング手法を提案する.

キーワード CCAP, 高位合成, 動作合成, ハードウェア/ソフトウェア協調設計

Handling of Variables and Functions for Software Compatible Hardware Synthesizer CCAP

Kenichi NISHIGUCHI†, Nagisa ISHIURA†, Masanari NISHIMURA†, Hiroyuki KANBARA††,
Hiroyuki TOMIYAMA†††, Yutetsu TAKATSUKASA††††, and Manabu KOTANI††††

† School of Science & Technology, Kwansei Gakuin University, Sanda, 669-1337 Japan
†† ASTEM RI, 134 Minamimachi Chudoji, Kyoto, 600-8813 Japan
††† Graduate School of Information Science, Nagoya University, Nagoya, 464-8603 Japan
†††† Graduate School of Informatics, Kyoto University, Kyoto, 606-8501 Japan
E-mail: †{scbc9034,ishiura,nishimura.m}@ksc.kwansei.ac.jp, ††kanbara@astem.or.jp,
†††tomiyama@is.nagoya-u.ac.jp, ††††{takatsukasa,kotani}@vlsi.kuee.kyoto-u.ac.jp

Abstract We are developing a high-level synthesis tool named CCAP (C Compatible Architecture Prototyper), which synthesizes arbitrary functions in ANSI-C programs into hardware modules callable from the remaining functions executed on a CPU. The synthesized hardware shares the entire memory space with the CPU and transfers data and controls through global variables. This eliminates the necessity of designing an interface for each hardware module. Programs including pointers are synthesized in a natural way, so that arrays and dynamic data allocated in the software may be accessed from the hardware using pointers. In this paper, we present the key synthesis techniques employed in CCAP, including the handling of variables, the mechanism of function calls using global variables, and scheduling of the function calls.

Key words CCAP, high-level synthesis, behavioral synthesis, hardware-software codesign

1. はじめに

高位合成は, プログラミング言語等によるハードウェアの動作記述からレジスタ転送レベルの回路を合成する技術であり [1], 大規模化する VLSI の設計効率化の手法として実用化が進められている.

高位合成の入力には, Verilog HDL, VHDL 等のハードウェア記述言語, SystemC(注1), SpecC(注2), Bach-C [2], Handel-C(注3)等のシステム記述言語, ANSI-C/C++ 等のプログラミング言語が使用される. ANSI-C は, 応用のリファレンスが C プログラムで与えられることが多く, 実行 (シミュレーション) が容易かつ高速であり, また習熟者が多い等の理由から, 高位合成の入力の有力な選択肢となっている.

C/C++ を入力とする高位合成ツールには, スタンフォード大学の SpC [3], 市販用では Catapult C Synthesis(注4), eXCite(注5) 等があり, 合成できる範囲に制限があるものの C/C++ をほぼそのまま入力として高位合成が行える. しかし, これらのシステムは単体としてのハードウェアを設計, 合成することを指向したものである. C/C++ で書かれたプログラムを仕様として, ソフトウェアとハードウェアにより構成されるシステムを設計するためには, ハードウェアの合成だけでなく, CPU とハードウェア間 (あるいはハードウェアとハードウェア間) のインターフェースの設計が別途必要となる.

ハードウェアとソフトウェアから成るシステム全体の動作記述から, ハードウェア化する部分を決定して合成を行う手法に岡田らの「ソフトウェア/ハードウェア協調設計方法」 [4] がある. この手法は, CPU とハードウェア間に設けた専用バッファ RAM を介して関数呼び出しの引数, 返り値の受け渡しを行うというものであり, インターフェースを個々に設計することなくソフトウェアからハードウェア化関数を呼び出すことができる. しかし, データの参照渡しを行うことはできないので, 配列等の大きなオブジェクトを受け渡す時にはデータ全体のコピーが必要となり, 非効率的である. また, ハードウェア化関数からハードウェア化関数を呼び出すことはできないので, 扱えるプログラムの構成が制限される.

我々は, ANSI-C プログラム中の任意の関数をハードウェア化し, 元のソフトウェアによる関数をそのまま置き換えることができることを目的とした高位合成システム CCAP (C Compatible Architecture Prototyper) を開発している. CCAP は合成するハードウェアが CPU とメモリ空間全体を共有することを特長とし, ハードウェア化関数と CPU 間のデータや制御の受け渡しはグローバル変数を介して行う. このため, インターフェースを別途設計することなくソフトウェアからハードウェアを起動することが可能である. また, ポインタを含む文の合成や, ポインタを用いたデータの参照渡し, ハードウェア化関数からハードウェア化関数の呼び出しも可能である. 本稿では, CCAP で合成を行う際の変数の扱い, グローバル変数を利用した関数呼び出し, 及びそのスケジューリング手法を提案する.

次章で CCAP の合成するシステムの構成, 3 章で CCAP の構成について記述する. 以降の章で CCAP のグローバル変数, ローカル配列等, 変数の扱いについて説明した後, グローバル変数を用いた関数呼び出しの扱いと, そのスケジューリング法を提案する. 実験結果を紹介し, 最後にまとめと今後の課題について述べる.

(注1) : http://www.systemc.org/
(注2) : http://www.specc.gr.jp/
(注3) : http://www.celoxica.com/
(注4) : http://www.mentor.com/
(注5) : http://www.yxi.com/

2. 合成されるシステムの構成

本研究の高位合成システム CCAP では, C プログラム中の関数のうち, 一部 (設計者が指定する) をハードウェアとして合成し, 残りをソフトウェアとして CPU で実行する. 図 1 に CCAP で合成されるシステムの構成を示す. CCAP の特長は, これらのハードウェアとソフトウェアが同じメモリ空間にアクセスする仕組みを取ることである. CPU とハードウェアはメモリアクセスの調停を行う Arbiter を介して主記憶 (キャッシュはあってもなくてもよい) に接続され, メモリ空間を共有する. 各ハードウェアはロード/ストアユニットを用いて主記憶にアクセスする. CPU やハードウェアからのメモリアクセス要求は Arbiter が先着順 (同時の場合は CPU を優先) に処理する. Arbiter はアクセス要求を主記憶に送り, アクセスが完了するまで要求元に stall 信号を送る.

ハードウェアとソフトウェアが同じアドレス空間にアクセスできるため, グローバル変数を介してデータの受け渡しが可能である. また, ポインタを用いたメモリアクセスをそのままハードウェアに合成でき, ソフトウェアが割り当てたオブジェクトにハードウェアからアクセスすることが可能となる.

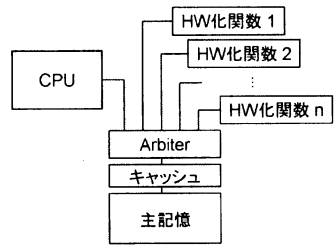


図 1 合成されるシステムの構成

3. 高位合成システム CCAP の構成

CCAP の処理の流れを図 2 に示す. 入力となる C プログラムの字句解析, 構文解析, およびハードウェア独立の最適化等のフロントエンド処理は SUIF(注6)で行い, SUIF の抽象構文木から CDFG (Control Data Flow Graph) を生成する. CDFG に次章以降で述べるグローバル変数, ローカル配列, 関数呼び出しの変換処理を行い, スケジューリング, バインディングの後, レジスタ転送レベルの中間表現を合成する. 最後に, この中間表現から Verilog-HDL によるハードウェア記述を生成する.

(注6) : http://suif.stanford.edu/

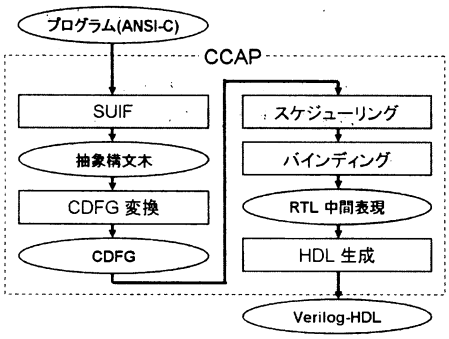


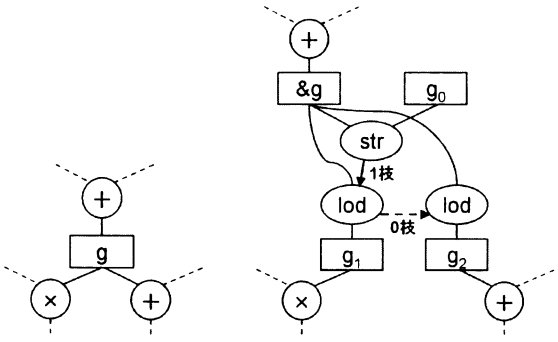
図 2　CCAP の構成

4. 変　　数

CCAP では, 関数中のローカル変数は従来の高位合成手法と同様にレジスタとして合成する. これに対し, グローバル変数は CPU とハードウェアが共にアクセス可能なメモリ空間に配置し, ロード/ストア演算によってアクセスする. ポインタアクセス, グローバル配列もメモリ空間に配置するが, 関数中でローカルに宣言された配列はアクセスの高速化のため, レジスタファイルに合成する.

4.1　グローバル変数の扱い

CPU/ハードウェア共通のメモリ空間上に配置するグローバル変数は全て "volatile" と考える. 即ち, あるアクセスにより読み書きした変数の値は, 次回のアクセス時にも同じとは限らないものと考える. これは, 一つのグローバル変数に対して複数のハードウェアやソフトウェアプロセスが同時にアクセスする可能性があり, また DMA や memory mapped I/O が実装されている可能性があるためである.



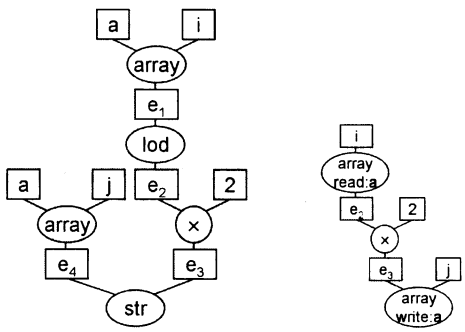
(a) 変換前 CDFG　　(b) 変換後 CDFG

図 3　グローバル変数へのアクセスの変換

グローバル変数へのアクセスは, 図 3 の例のようにロード/ストア演算に変換する. 図 3(a) は変換前の CDFG (SUIF の中間表現から作成した直後の CDFG) である. グローバル変数 g に代入された加算の結果を次の乗算と加算が参照している. この時点では, グローバル変数もローカル変数と同じデータ構造で表されている. グローバル変数に変換を施したものを図 3(b) に示す. $\&g$ はグローバル変数 g のアドレス, g_0, g_1, g_2 はそれぞれグローバル変数 g の値 (ローカル変数と同様, レジスタに割り当てられる) である. 演算 str, lod はそれぞれストアとロードを行う演算である. グローバル変数は "volatile" であるとするため, 同じグローバル変数への参照が複数回存在する時は, それぞれのアクセスに一つのロード/ストア演算を挿入する.

追加したロード/ストア演算の間には依存枝を挿入する. 図 3(b) の矢印が依存枝である. 本システムで扱う依存枝には「0 枝」と「1 枝」の 2 種類が存在し, スケジューリングの際に同時実行が許可される依存関係を「0 枝」で, 一方の演算の終了を待って次の演算を開始しなければならない依存関係を「1 枝」で表現する. 図 3(b) の実線の矢印は 1 枝, 破線の矢印は 0 枝を示す. 各演算はソースプログラム中の演算の実行順序を整数で保持しているので, この値に基づいて依存枝を張る. ロードとロードの間には 0 枝を, それ以外には 1 枝を設定する.

グローバル変数のアドレスは, ソフトウェアをコンパイル, リンクして得られる実行可能モジュールから取得する. 高位合成においてはレジスタ転送レベル中間表現の合成時までグローバル変数のアドレスをシンボル値で保持し, verilog 記述を出力する時にこれを具体的なアドレス値に置換する.



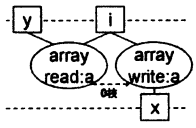
(a) 変換前 CDFG　　(b) 変換後 CDFG

図 4　a[j]=a[i]*2 に対する CDFG の変換

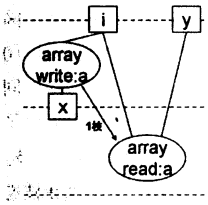
4.2　ローカル配列の扱い

ローカル変数として宣言した配列は, アクセスの高速化を図るため, レジスタファイルに合成する. 基本的に一つのローカル配列に対して一つのレジスタファイルを合成する. 図 4 に, それに伴う CDFG の変形処理を示す. 図 4(a) 中の array 演算は, 配列の先頭アドレスとインデックスから要素のアドレスを計算する演算であり, このアドレスに対してロード/ストアを行うことにより配列へのアクセスを行っている. 本システムではこれを図 4(b) に変換する. array_read:a, array_write:a はそれぞれインデックスを入力としてレジスタファイル a の読み出しと書き込みを行う演算である.

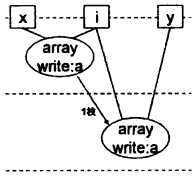
本システムでは, レジスタファイルは非同期読み出し同期書き込みで, 複数のポートを持つと仮定する. このため, 同一のレジスタファイルに対して複数のアクセスを一サイクルで実行することが可能である. 図 5 にそれぞれの場合に設定される順序枝を示す. WAR 依存の場合, array_write 演算が値を更新するよりも前に array_read 演算の結果が得られるので同時実行が可能である. RAW 依存及び WAW 依存の場合には, 後続の演



(WAR)Write After Read

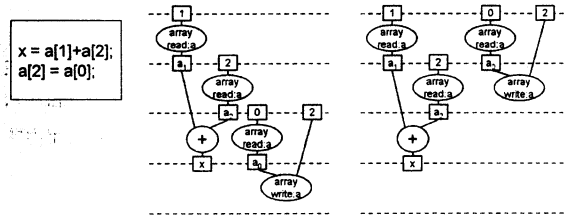


(RAW)Read After Write



(WAW)Write After Write

図 5　レジスタファイルアクセス演算の同時実行可能性



(a) プログラム　(b) 単一ポートの場合　(c) マルチポート (2R1W) の場合

図 6　レジスタファイルアクセスのスケジューリング

算を 1 サイクル遅らせる必要がある. レジスタファイルのスケジューリングの例を図 6 に示す. 図 6(a) のプログラムを単一ポートとマルチポート (読み出し 2 ポート, 書き込み 1 ポート) のレジスタファイルを使用してスケジューリングを行った結果をそれぞれ図 6(b), 図 6(c) に示す.

4.3　従来手法との比較と制限

SpC [3] では静的ポインタ解析とタグ付けにより, ローカル変数へのポインタによるアクセスの合成を実現している. また, メモリ管理ハードウェアの導入により, malloc/free の合成も実現している. ただし, SpC ではグローバル変数へのポインタ, ポインタを引数に持つ関数呼び出しは扱えず, その意味で単体のハードウェアの合成を指向したものと言える. これに対し CCAP では, グローバル変数へのポインタが使用可能であり, ソフトウェアがスタック領域及びヒープ領域に確保したデータにアクセスすることもできる. ただし, ハードウェア化される関数内でのローカル変数へのポインタ, malloc/free を扱うことはできない. また, 現時点ではローカルの構造体, コンパイル時にサイズの確定しないローカル配列を合成することはできないが, 今後スタック領域にこれらのデータを配置する等の方法により対応する予定である.

5. 関数呼び出しの合成

ハードウェア化される関数は, グローバル変数を通じてソフトウェア関数および他のハードウェア化関数と制御や値の受け渡しを行う. また, CCAP ではスケジューリングの際, 関数呼び出し, ローカル配列を扱うと効率的な結果を得ることができる.

5.1　グローバル変数を用いた関数呼び出し

関数 "callee" をハードウェアとして合成する際には, 次の 3 種類のグローバル変数を新たに導入する.

（1） _RUN_*callee*

関数の実行の開始・終了の制御に用いる. 1 が実行中, 0 が非実行中を表す.

（2） _ARG_*callee*_1, ... , _ARG_*callee*_n

関数の引数を格納する.

（3） _RET_*callee*

関数の返り値を格納する.

関数の起動や及び引数/返り値の授受は, これらグローバル変数への代入に変換する.

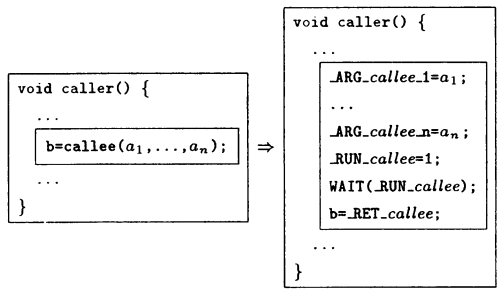


図 7　呼び出し側の変換

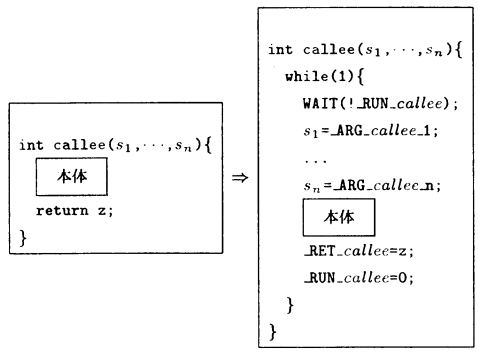


図 8　関数側の変換

図 7 に関数 caller から関数 callee を呼び出す文の変換を示す. まず, 引数 a_1, ..., a_n をそれぞれ_ARG_*callee*_1, ..., _ARG_*callee*_n に代入する. 次に, _RUN_*callee* に 1 を代入し, *callee* の実行開始を指示する. _RUN_*callee* の値は *callee* 実行中は 1 で, 実行が終了すると 0 になるので, 次の WAIT でこれを待つ. 最後に *callee* の返り値を受け渡すグローバル変数 _RET_*callee* から返り値を受け取る.

図 7 の WAIT の実装には, 割り込みの利用等も考えられるが, busy loop を用いればグローバル変数へのアクセス以外に特別なハードウェアもソフトウェアも必要としない実装が可能である. これは, WAIT(_RUN_*callee*) を while(_RUN_*callee*){} に

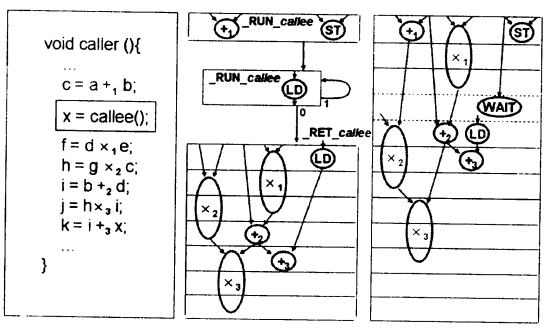
変換することにより実現できる．ただし，ハードウェアの場合にはスケジューリングの自由度を上げるために，後述する通りWAIT を 1 つの特殊な演算として扱う．

図 8 にハードウェア関数 *callee* 側の変換を示す．関数 *callee* は_RUN_*callee* の値が 1 になるのを待って実行を開始する．まず，関数 *callee* は，_ARG_*callee*_1, ..., _ARG_*callee*_n をローカル変数 s_1, ..., s_n に代入して引数を受け取り，本体での計算を実行する．本体での計算の終了後，返り値を _RET_*callee* に代入し，最後に _RUN_*callee* に関数の終了を示す 0 を代入する．

5.2 従来手法との比較および本手法の制限

プログラム中の一部の関数をソフトウェアから起動可能なハードウェアに合成する手法は [4] でも提案されているが，この手法では CPU とハードウェア間に設けた専用バッファ RAM を介して引数と返り値の受け渡しを行う．このため，配列等を受け渡すにはその全体のコピーが必要になる．また，ハードウェア化関数からハードウェア化関数を呼び出すことはできない．

これに対し，本研究の手法では，ソフトウェアとハードウェアが主記憶のアドレス空間を共有するため，ポインタを用いた値の受け渡しが可能である．データの参照渡しが行えるため，大規模なデータの一部だけを参照，更新する場合に特に効率的である．また，ソフトウェアが動的に割り当てたデータにハードウェアがアクセスすることも可能である．さらに，ハードウェア関数からハードウェア関数を呼び出すことも可能である．ただし，再帰呼び出し，およびハードウェア関数からのソフトウェア関数の呼び出しはできない．



(a) C プログラム　(b) while 文　(c) WAIT 演算

図 9　関数呼び出しのスケジューリング

```
s0 :  w = LSU.load(&_RUN_callee)
      if (w) → s0 else → s1
s1 :
```

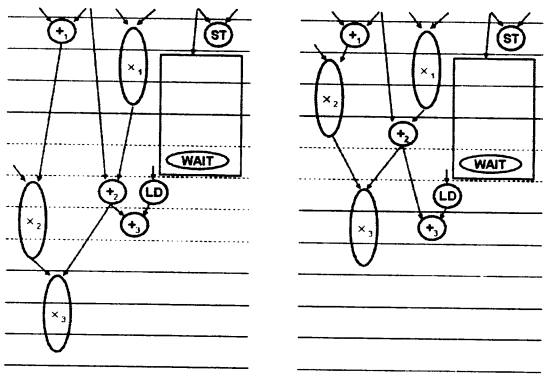
図 10　WAIT 演算の動作 (LSU はロード/ストアユニット)

5.3 関数呼び出し (WAIT 演算) のスケジューリング

ハードウェアから関数呼び出しを行った場合，呼び出した関数は呼び出された関数の終了を WAIT という特殊な演算で待つ．また，呼び出された関数側でも WAIT 演算で自身が起動されるのを待つ．

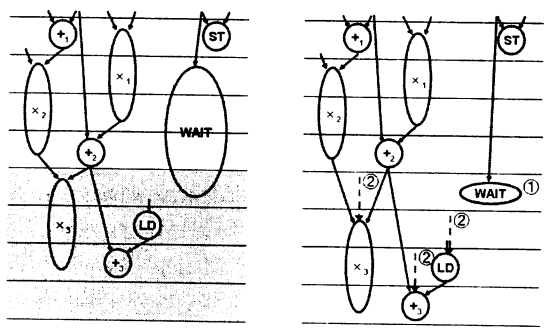
図 9 (a) の記述からハードウェアを合成する場合を考える．3 つの乗算 $\times_1$, $\times_2$, $\times_3$ は関数の引数や返り値とは独立であるものとする．WAIT を単純に while 文に変換して合成すると，図 9 (b) のように関数呼び出しの前後で基本ブロックが分断されてしまう．しかし，WAIT を演算として扱えば，図 9 (c) のようにこれらの基本ブロックを 1 つにできるため，スケジューリングの自由度を上げることができる．

WAIT 演算は，RTL 生成の際，図 10 に示すように，_RUN_*callee* のロードとその値のテストを値が 0 になるまで繰り返す状態遷移に展開する．WAIT はサイクル数不定のマルチサイクル演算となるため，スケジューリング時に他の演算とは独立の実行ステップを割り当てなければならない点に注意が必要である．



(a) 図 9(c) と等価なスケジューリング　(b) 最適化

図 11　関数の実行サイクルの下限が既知の時



(a) リストスケジューリング結果　(b) 再配置

図 12　スケジューリング・アルゴリズム

呼び出される関数の実行サイクル数は一般に不定であるが，スケジューリング結果からその下限を知ることができる．この情報を用いれば，呼び出し側の WAIT に関してさらに効率的なスケジューリングが可能である．例えば，関数 *callee* の実行サイクル数の下限が 4 とすると，図 9 (c) のスケジューリングは図 11 (a) と等価である．すると，WAIT が実行されないサイクルには他の演算をスケジューリングできるため，図 11 (b) のようなスケジューリングを行って総サイクル数を減らすことができる．

本稿ではこのような WAIT のスケジューリング法として，特

別なスケジューリングアルゴリズムではなく，既存のスケジューリング法をそのまま用いる方法を提案する．概要は次の通りである．

(1) WAIT 演算をマルチサイクル演算 (サイクル数は呼び出される関数のサイクル数の下限) と考え，通常のアルゴリズムでスケジュールする．(図 12(a) のスケジューリング結果が得られる)

(2) WAIT 演算をマルチサイクル演算の最後のステップに配置する．(図 12(b) の①)

(3) 次の処理により，WAIT 演算に独立した実行サイクルを割り当てるようにする．(図 12(b) の②)

(3-a) WAIT 演算より終了が遅い演算をマークする．(図 12(a) の網掛けされたサイクルにかかる演算のうち，WAIT を除いたもの全てがマークされる．)

(3-b) マークした演算の開始を一斉に s ステップだけ遅らせる．ただし，s はマークした演算の開始から WAIT 演算の終了までのサイクル数の最大値である．

6. 実験結果

CCAP は Perl (ver. 5.8.7) で実装しており，Linux 及び Windows 上の Cygwin 環境で動作する．フロントエンドには SUIF (ver. 1.3.0.1) を利用している．

DSPstone(注7) の convolution.c と fir.c の合成を行った結果 (サイクル数とレジスタ数) を表 1 に示す．合成は，加算器は 2 個，ALU が 2 個，乗算器が 1 個，ロード/ストアユニットが 1 個で，乗算は 2 サイクル，その他の全ての演算は 1 サイクルで行えるという制約で行った．プログラムにはループアンローリングを施している．convolution.c と fir.c の "LDST" の行はローカル配列を主記憶にバインディングをした場合，"RF" はレジスタファイルにバインディングをした場合である．サイクル数の減少の要因はレジスタファイルのアクセスにアドレスの計算が不要であるためと考えられる．c_fir.c は下限が 4 サイクルの関数を 2 回 fir から呼び出すものである．関数呼び出しの WAIT を while に変換したもの，そのまま WAIT 演算としてスケジューリングしたものをそれぞれ "while", "wait" の行に示す．提案のスケジューリング手法によりサイクル数が減少している．

表 1 合成結果

		サイクル数	レジスタ数
convolution.c	LDST	41	20
	RF	30	21
fir.c	LDST	96	38
	RF	28	18
c_fir.c	while	49	26
	wait	40	22

ADD × 2, ALU × 2, MUL × 1, LDST × 1

(注7): http://www.ert.rwth-aachen.de/Projekte/Tools/DSPSTONE/dspstone.html

7. むすび

本稿では，高位合成システム CCAP における変数の扱い，グローバル変数を用いた関数呼び出し，及び関数呼び出しのスケジューリング手法を提案した．CCAP では合成したハードウェアでソフトウェア関数を置き換えることができる．また，CCAP ではポインタを用いてソフトウェア/ハードウェア間のデータの授受が行える．今後は動作記述の制限の緩和，及び合成したハードウェアの性能評価を行う予定である．

謝辞

本研究に際し，御討論頂いた立命館大学の梅原直人氏，中谷嵩之氏，および関西学院大学石浦研究室の諸氏に感謝致します．

文献

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

[2] T. Kambe, A. Yamada, K. Nishida, K. Okada, M. Ohnishi, A. Kay, P. Boca, V. Zammit, and T. Nomura: "A C-Based Synthesis System, Bach, and its Application," in *Proc. ASP-DAC 2001*, pp. 151–155 (Jan. 2001).

[3] L. Séméria, K. Sato, and G. De Micheli: "Synthesis of Hardware Models in C With Pointers and Complex Data Structures," in *IEEE Trans. VLSI Systems*, vol. 9, no. 6, pp. 743–756 (Dec. 2001).

[4] 岡田和久: "ソフトウェア/ハードウェア協調設計方法," 公開特許広報, 特開 2003-114914 (2003).