Vol.2017-SLDM-181 No.31
Vol.2017-EMB-46 No.31
2017/11/7

# FPGA アクセラレータ開発を支援するためのツール環境

富田 憲範[1,a)]　一場 利幸[1]　田宮 豊[1]　今里 賢一[2]　山下 公彰[2]　藤澤 久典[1]

**概要**：科学技術計算など大量のデータと計算を扱う分野では，処理速度や消費電力の面で CPU は効率が悪いため，アクセラレータの利用が広まっている．アクセラレータを FPGA で実装する場合，回路の開発作業を伴うため，開発期間の長さが問題となるが，高位合成ツールを利用すれば，C 言語を用いて高い抽象度で回路動作を記述可能となり，開発期間を短縮できる．しかし，ソフトウェアとして記述された C コードを高位合成すると，性能が低い，FPGA に入り切らない，といった問題が発生する．そのため，FPGA 設計者がソフトウェアの処理内容を理解した後，高位合成用の C コードを記述しなおす，という手順で開発されることがあった．本稿では，ソフトウェアを FPGA でハードウェア化することの難しさと，FPGA 設計者がソフトウェアを理解するとはどういうことか，について考察する．そして，メモリアクセスに基づいてソフトウェアの振る舞いを可視化する手法を提案し，ツールを実装してその評価を行う．

**キーワード**：FPGA アクセラレータ，ソフトウェア解析，メモリアクセス解析，動的解析，高位合成，回路設計支援

TOMITA Yoshinori[1,a)]　ICHIBA Toshiyuki[1]　TAMIYA Yutaka[1]　IMAZATO Kenichi[2]
YAMASHITA Hiroaki[2]　FUJISAWA Hisanori[1]

## 1. はじめに

FPGA がサーバ/PC に搭載され，従来ソフトウェアで実現されていたアプリケーション・ソフトウェア (アプリ) の一部処理が，ハードウェア回路化され FPGA へオフロードされる (CPU 負荷の軽減)〜すなわち FPGA アクセラレータ [1], [2] がごく普通に利用される未来を実現するため，我々は研究・開発に取り組んでいる．ここでの課題は，ソフトをハード化する開発作業が極めて高コストだということである．我々の経験では，こうやればスムーズにハード化できると保証できる一般化された開発手法が無くて，アプリごとの個別対応でコツコツとハード化を試みてきた．高収益が期待できる ASIC/SoC 開発であれば，十分な開発費用と時間をかけられる．一方 FPGA アクセラレータは，新領域の開拓/ニッチ分野への進出が向いていると我々は考えているため，FPGA アクセラレータの有効性を，性能向上と開発コストを指標値として，短期間で確認したい．ソフトの C コードを高位合成してハード化しても処理性能が出ないことも課題である．ハードで性能を出すためには多数の並列処理が必要があり，そのために，回路設計者がいったんソフトの処理内容を理解した後に，並列動作する回路として再設計をしている．これら課題を解決するために，現在，我々は FPGA アクセラレータ開発を支援するツールを開発している．ツールは，ソフト実行時のメモリアクセスを記録・解析・可視化することで，設計者がソフトの処理内容を理解するのを支援できる．

## 2. FPGA アクセラレータの概要

現在，GPU がアクセラレータとしてよく利用されるが，GPU には苦手な処理がある．そこをハードウェア回路で高速化できないだろうか? FPGA は，回路データを電気的に書き込むことで，何度でも自由に回路を変更できる半導体チップである．FPGA のチップ価格は高いが，チップ製造コストは不要なので，試作品や少量多品種開発のために使われてきた．半導体の微細化が進み，FPGA に搭載可能な回路規模が増大し，かなり複雑な処理を行う回路でも，1 チップの FPGA に入るようになった．そのため，ソフトの一部処理をハード化してオフロードする用途にも，FPGA が利用可能になった．しかし FPGA は回路設計で大きな

[1] 富士通研究所
211-8588 川崎市中原区上小田中 4-1-1
[2] 富士通九州ネットワークテクノロジーズ
814-8588 福岡市早良区百道浜 2-2-1 富士通九州 R&D センター
[a)] yoshin-t@jp.fujitsu.com

工数がかかるのがデメリットである．これについては，高位合成ツールが有効とされている．C 言語のようなプログラミング言語を用いてハードの動作記述の抽象度を上げることで，設計・検証の工数を削減できる．

**図 1** に FPGA アクセラレータのアーキテクチャを示した．(A) は CPU と FPGA を直接接続する形態 [3], [4], (B) は FPGA を搭載した拡張ボードを PCIe スロットに装着する形態である．拡張ボード上に DDR3 SDRAM などのメモリが搭載されている場合が多い．(C) は FPGA チップ内の回路アーキテクチャの例である．(D) はソフトからの FPGA の見え方である．アプリは高速化したいソフトである．その一部処理がハード化され FPGA 内部で動作する．アプリは，FPGA アクセラレータを制御するミドルウェアを介して，FPGA の機能を呼び出す．ミドルウェアは OS 内のデバイスドライバとやり取りし，デバイスドライバは DMA(Direct Memory Access) 転送や PIO(Programmed input/output ) 等の手段によって，直接ハードを制御する．

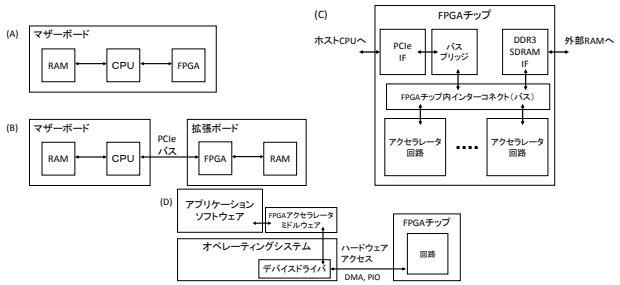


**図 1** FPGA アクセラレータのアーキテクチャ

FPGA アクセラレータの開発フローは以下の通りである．性能を左右する (2)〜(4) が本稿の主な対象である．

(1) ソフトの性能評価 (プロファイリング) を行い，ハード化したい処理の候補を探す
(2) ハード化候補周辺のソフトの処理内容を理解する
(3) アーキテクチャ設計を行う．ソフトとハードのインターフェイスと，ハードのアーキテクチャをラフに検討する．ハード部の処理を複数のブロックに分割し，ブロック図レベルで動作する様子を考える
(4) 性能をラフに見積もる．性能不足の場合は，アーキテクチャを修正して性能改善を検討する．このとき，ハード化候補の範囲を変更する場合もある
(5) 回路量をラフに見積もる．FPGA 内蔵メモリ (ブロック RAM) の容量, FPGA 内蔵の専用演算器 (DSP: Digital Signal Processing) の個数が収まるか確認する
(6) 回路の詳細設計を行う
(7) 動作を検証する

## 3. ソフトの FPGA 化とソフト解析

高位合成ツールを使う熟練ハード設計者によれば，設計フロー (2) でソフトを解析して処理内容を理解することが大事だという．ソフト開発の分野でもソフト解析は行われるが，ここでは「ソフトをハード化して高速化する」ことを目的としたソフト解析が必要なので，ソフト分野の解析ツール [5] は不十分である．本節では，ソフトの FPGA 化について，ソフト開発者にも知ってほしいことを述べる．

### 3.1 並列処理

結論から言えば，ソフト解析の主な目的は，並列実行可能な処理を探すことである．ハードウェア回路だから高速化する，という単純な話ではない．CPU のクロック周波数は FPGA の約 10 倍高速なので，FPGA は，ソフトと等価な処理を，CPU とは異なる方法でより効率的に行わなければならない．その効率化は並列処理で実現される．

並列処理には 2 種類の形態がある．1 つはデータ並列と呼ばれ，複数のデータに対して同時に演算を行う．もう 1 つはタスク並列と呼ばれ，その一種にパイプライン処理がある．データ並列は比較的容易に自動で行える．ソースコード内のディレクティブ指定やコンパイラ・オプションの指定で，データ並列が可能になる．ループ展開と呼ばれる技術がデータ並列では用いられる．一方，タスク並列は自動化が難しい．C 言語などは，命令文が 1 行ずつ逐次実行されることを前提としている．逐次処理で書かれたものを，タスク並列にして複数タスクを同時実行させようとしても，タスク同士に依存関係があって，同時実行を阻害してしまうケースが多い．ソフトの C を高位合成して性能が出ないのは，逐次動作する回路が生成されるためだろう．

ソースコードの修正で，この依存関係を無くせる場合がある．ソフト解析では，タスク間の依存関係を調べ，依存関係を解消できるかを検討する．現在の高位合成ツールは，1 つのタスクを回路へ自動変換するだけであり，多数のタスクを効率よく連携して並列動作するものを一括で自動変換するわけではない．そこは，設計者が考えてアーキテクチャを設計する必要がある．逐次処理するソフトのコードから，効率的にタスク並列で動作するアーキテクチャを考え出すのを，ツールで支援することを目指す．

### 3.2 CPU と FPGA のタスク並列の違い

CPU のキャッシュメモリを有効活用するのが，ソフトの最適化技術の 1 つである．FPGA でもブロック RAM を駆使して，性能向上を狙う．どちらも，メモリアクセスの工夫が必要なのは共通だが，メモリ容量と，特にタスク並列動作時のメモリの使い方の 2 点が，大きく異なる．なお，ここで言う FPGA でのタスクとは，回路アーキテクチャをブロック図で表現したときの 1 ブロックをイメージしており，人間にとって理解しやすい粒度で処理を整理したものを指す．

現行の CPU はマルチコア化により，数十のスレッドを

同時実行可能になっている．しかし共有キャッシュや主記憶へのアクセスが多発すると，メモリアクセス待ちが増えて性能低下する．そのためタスク (スレッド，プロセス) 数は適度に抑えておく．FPGA でのタスクの同時実行数は，回路が FPGA に収まる限り増やせる．ブロック RAM は 1 個のタスクで占有するか，タスク間でデータを受け渡すバッファとして利用する．例えばバッファサイズを 2 倍にしたダブルバッファ構成にすれば，バッファの読み出しと書き込みを同時実行可能である．このため，FPGA は CPU よりも多数のタスクを同時実行可能になる．直感的なイメージとしては，ソフトでは並列実行させない小規模の処理さえも，ハードでは並列実行させることになる．

### 3.3 ソフトを FPGA 化することの難しさ

並列動作は，Kahn process network(KPN) モデルを使って説明されることがある．KPN では，プロセス間が FIFO チャネルで接続され，複数のプロセスが連携して動作する．これは，FPGA におけるタスクとブロック RAM によく似ている．KPN(FPGA のブロック図) では，入力データが入り，プロセス (回路のブロック) 間をデータが流れていき，最終的に出力データとなって出ていく，というデータフローを自然にイメージできる．一方，C/C++言語の場合は，ソースコードをながめても，データフローをすぐにはイメージできない．KPN のような制約が無いため，どこがプロセスになりうるかも不明である．このあたりにソフトのハード化作業の難しさがあるのではなかろうか．

FPGA のブロック RAM は容量が少ない．データサイズが大きくてブロック RAM に収まりきらない場合，CPU のキャッシュのように外部 RAM とブロック RAM との間でデータを交換するか，もしくは，処理内容を見直して 1 回の処理で使用するデータサイズを小さくできないか検討する．前者の場合，FPGA が CPU に似た動作をすることになるが，CPU のほうがメモリ帯域が広いので，FPGA が CPU に勝つのが難しくなる．後者は，さらなるソフト解析が必要になる．

C/C++言語のポインタ変数は，ハード化で問題になる．ポインタ変数は，高位合成ツールで扱えない場合が多い．ポインタ変数は 32〜64 ビットのアドレス値を保持し，広大な主記憶空間の任意のメモリ位置を read/write 可能になる．もしも FPGA 化した後もポインタ値が回路に残っているとしたら，それは外部 RAM や CPU 側の主記憶メモリへのアクセスを行うことを意味しており，メモリアクセス待ちによる性能低下を起こす可能性がある．高位合成ツールを用いた設計では，ポインタ変数は使わずに，配列変数とインデックス変数を使って，ハードの動作を記述する．これは，主記憶空間から特定のメモリ領域を抜き出してきて，配列変数という小容量のメモリへ局所化する，という設計の工夫を行っていることだと言える．

**図 2** 左側が，CPU でプログラムを実行するときのメモリの使い方のイメージである．メモリ領域は主記憶上のどこに配置されていても，プログラマはあまり意識する必要はない．図 2 右側は，FPGA で性能が出るときのメモリの使い方である．メモリを局所化してブロック RAM だけで処理を行えるように工夫する．

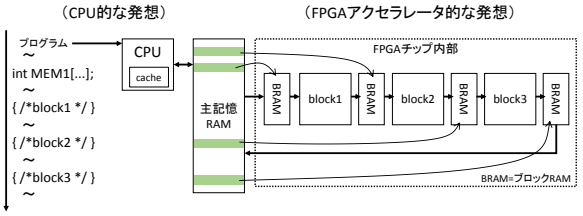


**図 2** CPU と FPGA のメモリ使用法の違い

### 3.4 ハード設計者がソフトウェアを理解するとは？

ソフトの動きを説明する図として，フローチャートやタスクグラフがある．ハード化目的のソフト理解では，多数のタスクを同時実行できるタスクグラフを描くことが，ゴールになるだろう．ソフト理解では，CPU/GPU 等デバイスに依存した最適化は不要であり，処理を実現するアルゴリズムの本質が見えれば十分である．性能は，回路アーキテクチャと関係するため，後で考慮すればよい．

## 4. メモリアクセスにもとづくソフト解析手法

動的解析にてメモリアクセスを検出することで，ソフトを解析する手法を提案する．ポインタ変数等の解析は，ソースコードを字面で追う静的解析では限界があり，実際にプログラムを実行して調べる動的解析が必要になる．可視化については 5 節で述べる．

### 4.1 なぜメモリアクセスに注目するのか？

高位合成ツールがポインタ変数をうまく扱えないこと，並列動作する回路ではメモリ配置が重要になることから，メモリの検討は必須である．ソフトの動作をアセンブリ言語レベルで考えると，メモリの読み書き，レジスタの読み書き，各種演算処理，ジャンプ命令やサブルーチンコールなどインストラクションポインタ (IP) の制御，が主な命令である．以下の仮説から，メモリアクセスの調査が必要だと考えた．

- レジスタと演算器は，回路ではデータパスと呼ばれる部分であり，データパス合成は高位合成ツールが得意なので，高位合成ツールに任せてしまえばよい．
- IP の制御は，プログラムを短く記述するための仕組みであり，ソフトの挙動の本質とは無関係である．
- 条件ジャンプ命令は，条件の評価処理と分岐先の処理内容を調べれば十分である．

### 4.2 ソフト解析処理の流れ

**図 3** に，ソフトを解析する処理の流れを示した．ツールへの入力は，解析対象ソフトの実行ファイルとソースファイル，および対象ソフトが使う入力データである．静的解析は，LLVM/Clang の AST parser の機能を利用してソースコードから情報を抽出したり，デバッグ情報 (DWARF) を利用してアドレス値とソースコードの対応付け等をすることである．

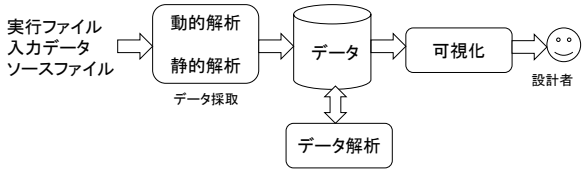


**図 3** ソフト解析処理の流れ

### 4.3 メモリアクセスを検出する手段

プロファイリングとは，ソフトの様々な特長を計測することであり [6]，メモリアクセスの検出もその一種と言え，プロファイリング技術を応用できる．対象ソフトに調査用コードを埋め込んで計測する手法があり，コードを埋め込むことを instrument と呼ぶ．実行ファイル (バイナリファイル) に instrument する手法は，Dynamic Binary Instrumentation(DBI) と呼ばれる．Valgrind[7] はメモリ関係のバグを調査するツールとして有名で，バイナリコードを中間コードに変換し，中間コードレベルで調査用コードを埋め込んだ後，中間コードを CPU のネイティブコードへ再コンパイルする．

メモリアクセスの検出は，CPU のデバッグ機能の一種である watch point 機能を利用できる．メモリアドレスを指定しておき，指定アドレスにアクセスがあったときに割り込みが発生する．しかし CPU のデバッグ機能には制限があり，Intel x86 プロセッサでは，1 度に設定できる watch point の数は 4 個まで，指定したアドレスから 1，2，4，8 バイトの範囲内のアクセスだけ検出可能である [9]．

今回は，Valgrind と GDB[8] を組み合わせて，メモリアクセスを検出し記録することにした．GDB にはリモートデバッグ機能があり，サーバ機能の GDB をターゲットデバイス上で実行し，クライアント機能の GDB を開発用マシン上で実行する．Valgrind は GDB サーバ機能を内蔵し，クライアント側 GDB を通じて，Valgrind の強力なメモリデバッグ機能を駆使しながら，対象プログラムを調査できる．たとえば Valgrind 下でプログラム実行すれば，x86 プロセッサの watch point のハード的な制約を受けずに，自由に watch point を設定できる．GDB は，Python スクリプトで処理を自動化できるので，ブレークポイントの設定，watch point の設定，メモリアクセスの検出と記録…，といった一連の処理をスクリプトにして自動化した．

メモリアクセスを検出して採取するデータは，(1) 時刻 (実時刻である必要はなく，メモリアクセスの発生順序を区別できればよい)，(2) 命令アドレス，(3) メモリアドレス，(4) 種別 (Read か Write か)，の 4 項目を基本とした．

### 4.4 監視対象メモリ領域のアドレスを求める方法

watch point を設定するときにメモリアドレスで指定するので，メモリアドレスを調べる必要がある．グローバル変数，static 変数は，コンパイル時 (実行ファイル作成時) にメモリ領域のアドレスが決定するので，実行ファイルのデバッグ情報からメモリアドレスを求められる．

関数内で宣言されるローカル変数は，関数実行時のスタックフレーム領域にメモリ確保されるため，実行時にアドレス値が決定される．そこで，関数の実行開始場所にブレークポイントを設定することで，ブレークポイントが発火したときにアドレス値を求められる．また，関数が終了するときにスタックフレームが破棄されるので，その地点にもブレークポイントを設定し，watch point を消去してメモリアクセスの監視を終了する．解放後は，同じメモリアドレスが別用途のメモリ領域として再利用されるので，監視終了は，必須である．

malloc/free などで動的メモリ確保を行う場合は，それが行われた場所でブレークポイントを設定して，監視の開始，終了を行う．しかし，関数の引数でポインタ値が渡されるような場合，ポインタ値が指すメモリ領域が，ソースコード中のどこで確保されたのか知るのは困難である．Valgrind 下でプログラムを実行して，GDB から Valgrind の GDB サーバにアクセスする場合，“`monitor check_memory addressable` アドレス” コマンドで，メモリ領域が確保された場所を求められる．

## 5. 可視化で得られる情報とその効果

可視化の基本アイデアを示した後，サンプル C コードを用いて，可視化の実例と効果を述べる．

### 5.1 可視化の基本アイデア

メモリアクセスを可視化すると，処理とデータの流れが見えてくる．その基本アイデアを**図 4** に示した．4.3 節の採取データから，次の手順でグラフを作成する．

- 「命令アドレス」と「メモリアドレス」をグラフの節点とする
- 種別が Read(R) のとき，メモリアドレスの節点から，命令アドレスの節点への向きの枝を追加する
- 種別が Write(W) のとき，命令アドレスの節点から，メモリアドレスの節点への向きの枝を追加する

つづいて，意味的に似た節点をグループ化して節点数を減らすことで，グラフを小さくして見やすくする．最後に，

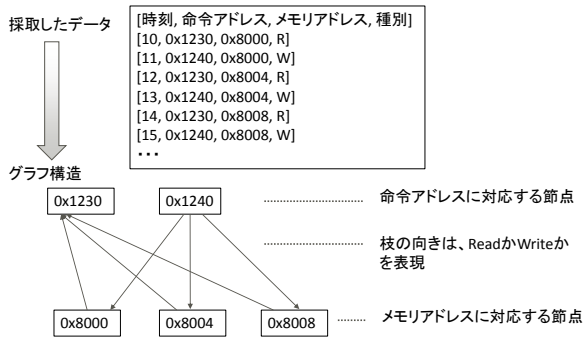


**図 4** メモリアクセスを可視化する

グラフの節点の表記を，アドレス値から人間にとって分かりやすいものへ変換する．ソースコードで使われている字句を用いて表記する．

- 「命令アドレス」節点については，命令アドレスと対応する，ソースファイル名，関数名，行番号を利用する
- 「メモリアドレス」節点については，変数名を使う．たとえば配列変数のときは，すべての配列要素を 1 グループにして，配列変数名をつける

最終的な可視化結果は**図 6** 上段のようになる．以下では**図 5** のサンプル C プログラムを用いて，可視化とその効果について述べる．

```
1  #include <stdio.h>
2
3  void ft01(int in[8], int out[8])
4  {
5          int j;
6          int mem1[8];
7          int t0=0;
8          for (j=0; j<8; j++) { /* block1 */
9                  t0 += in[j];
10                 mem1[j] = in[j] + 1;
11         }
12         for (j=0; j<8; j++) { /* block2 */
13                 mem1[j] = mem1[j] * 5 / t0;
14         }
15         for (j=0; j<8; j++) { /* block3 */
16                 out[j] = mem1[j] + 1;
17         }
18 }
19
20
21 int main(int argc, char* argv[])
22 {
23         int i;
24         int in[8] = { 1,1,2,2,3,3,4,4 };
25         int out[8];
26         ft01(in, out);
27         for (i=0; i<8; i++) printf("%d␣", out[i]);
28         printf("\n");
29         return 0;
30 }
```

**図 5** サンプル C プログラム

### 5.2 メモリアクセスの関係図

図 6(A) 上段は，5.1 節で述べたようにメモリアクセスの関係を可視化したものであり，タスクグラフや回路のブロック図に似た図形になる．楕円の節点は，命令アドレスをグループ化したものである．ここでは，関数よりも小さな単位で処理を見たかったため，コードブロックでグループ化した．コードブロックとは，ある行から連続する複数行の範囲のことで，基本的には，ループ文があったときに，そのループ文の範囲を 1 つのコードブロックにしている．また図 5 の 22〜25 行目など，ループ以外の行も 1 つのコードブロックにする．

命令アドレスの節点は「(関数名)/(種類)/(行番号)」と表記した．種類はループの場合は for，while，do などであり，ループ以外のコードブロックについては，STMT(statement の意) などは，LLVM/Clang 内部の用語から取っている．ft01/for/8 は，関数 ft01 の 8 行目から始まる for ループ文の意味で，main/STMT/22 は，main 関数の 22 行目から始まる文の意味である．

矩形の節点は，メモリアドレスをグループ化したものである．ここでは，配列変数の要素すべてをグループ化したので，配列変数の名前をつけた．

### 5.3 メモリ領域のライフタイム解析

メモリ領域の分割・結合といったメモリ使用方法の工夫のためには，ライフタイム解析が有効である．メモリアクセスの採取データを集計することでメモリのライフタイムを求めて，図 6(A) 中段のように可視化できる．横軸は時刻である．縦軸はメモリアドレスに対応していて，0〜7 は配列変数 in，8〜15 は mem1，16〜23 は out である．メモリアクセスが発生した箇所を濃い色にし，オレンジが write，グリーンが read である．write から最後の read までを薄い色にした．これは，その期間，メモリ素子が同じ値を保持しつづけていたことを表す．新たな値が write されたとき，薄い色を別の色に変えている (2 色を交互)．ここでは，関数 ft01 が 1 回実行されたとき，配列変数 mem1 は 2 回使われ，他は 1 回だけ使われていることがわかる．

### 5.4 処理性能の簡易見積り

図 6(A) 上段を回路のアーキテクチャと見なして，タスクがパイプライン動作するときの処理性能 (スループット) を簡易的に見積もる．今回は，タスク 1 回の処理サイクル数 (レイテンシ) をメモリアクセス回数としていて，タスク内部の詳細な処理，レジスタや演算器など回路リソースの動作にかかるサイクル数は無視した (ユーザ指定可能)．図 6(A) 下段のガントチャートは，タスクがパイプライン動作した様子を示している．横軸は時刻である．縦軸はコードブロックとメモリである．矩形は，コードブロックとメモリの活動期間を表す．この図では関数 ft01 を 2 回実行したときを表していて，横に矩形が 2 つ並ぶ．ガントチャートは，パイプライン動作時のスループットを理解し

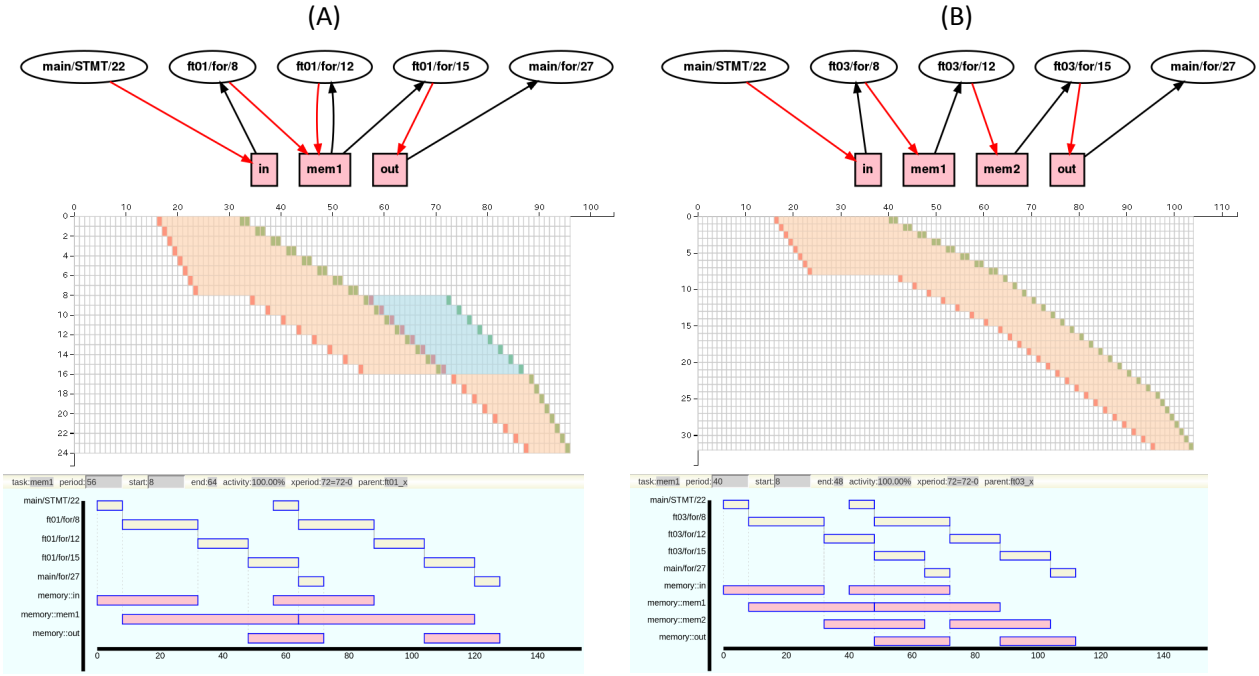


**図 6** ツールによる可視化 (上から，メモリアクセス関係図, ライフタイム図, ガントチャート)

やすい．矩形の先頭から次の矩形の先頭までの間隔を，タスクの実行開始間隔と呼ぶ．この間隔が短いほど，スループットは高くなる．

可視化は，アーキテクチャ検討にも有効である．変数 mem1 は，ライフタイム解析結果から 2 つに分割可能なことがわかるので，mem1 と mem2 の 2 つに分けた場合を図 6(B) に示した．(B) 上段から，タスクの間にバッファメモリを配置したような回路アーキテクチャを実現できることがわかる．(B) 中段からは，mem1 のライフタイムが短くなったことがわかる．ただし必要メモリ容量は増加している．ライフタイムの短縮は，タスクの実行開始間隔を小さくすることに貢献することになる．(A) では実行開始間隔が 56 だったのが (B) では 40 に減少し，スループットが向上している．

### 5.5 ツールによる効果の評価

現在開発中のツールを FPGA 設計者に使用してもらい，ツールの効果を評価したところ，ソフト解析の工数を約半分に削減できる見込みである．とくに，解析対象のソフトが複雑な場合や，解析の担当者がまだこの業務に習熟していない場合に，ツールの効果が大きくなると予想している．

## 6. まとめ

ソフトを FPGA でハード化するとき，性能向上のためには，タスク並列と FPGA に適したメモリ使用法が重要であり，そのためのソフト解析が必要であることを述べた．そして，メモリアクセス検出に基づくソフト解析手法を提案した．提案手法をツールとして実装し，その効果を確認した．処理とデータの流れが可視化され，回路のアーキテクチャ検討と簡易的な性能見積りにも役立つことがわかった．今後は，ツールの改良を継続しつつ，様々なアプリでトライアルを行い，有効性を確認する予定である．

## 参考文献

[1] A. K. Jain, D. L. Maskell, S. A. Fahmy, “Are Coarse-Grained Overlays Ready for General Purpose Application Acceleration on FPGAs?”, DASC/PICom/DataCom/CyberSciTec 2016
[2] M. S. Abdelfattah, A.Hagiescu, Deshanand Singh, “Gzip on a chip: high performance lossless data compression on FPGAs using OpenCL”, IWOCL ’14.
[3] Nathan Woods, “Integrating FPGAs in high-performance computing: the architecture and implementation perspective”, FPGA ’07
[4] P. Colangelo, R. Huang, E. Luebbers, M. Margala, K. Nealis, “Fine-Grained Acceleration of Binary Neural Networks Using Intel Xeon Processor with Integrated FPGA”, FCCM 2017
[5] Scientific Toolworks, Inc., “scitools Understand”, `https://scitools.com/features/`
[6] A. Neustifter, “Efficient Profiling in the LLVM Compiler Infrastructure”, Masters Thesis, Vienna University of Technology, 2010.
[7] N. Nethercote, J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation”, PLDI ’07
[8] Free Software Foundation, “GDB: The GNU Project Debugger”, `https://www.gnu.org/software/gdb/`
[9] Intel, “Intel 64 and IA-32 Architectures Software Developer Manuals”, Vol. 3: System Programming Guide