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

2004－ARC－159　(15)
2004／7／30

C言語からVHDLへの変換における並列処理

下尾浩正[†]　山脇彰[††]　岩根雅彦[††]

再構成可能コンピューティングを利用した新しい並列処理パラダイムを提案する．提案するシステムでは，C プログラムをスレッドへ静的に並列化し，VHDL へ変換した後，再構成可能コンピューティングシステム上で直接並列実行する．ハードウェア化されたスレッドは，再構成デバイス上で，フラグを用いた高速な同期とレジスタを用いた高速な通信によって協調動作する．並列処理の効果と同期による性能への影響を評価するために，実機上でプログラムを用いて簡単な実験を行った．その結果平均で，1.14 倍の性能向上が見られ，再構成デバイス上での低オーバヘッドの同期通信は並列処理に良い効果がみられた．

Parallel processing in the translation to VHDL from the C language

Kosei Shimo,[†] Akira Yamawaki[††] and Masahiko Iwane[††]

This paper proposes a new parallel processing paradigm using a reconfigurable computing system. In proposed system, a C program is parallelized statically into threads. They are translated to VHDL, and then, they are executed by a reconfigurable computing system directly. The threads cooperate with a high-speed synchronization using flags on the reconfigurable device. To evaluate the effect of a parallel processing and the performance impact of synchronization, we have performed the preliminary experiments using some programs on a real machine. The result shows that extracting parallelisms improves the performance of 1.14 times at average. The low–overhead communication and synchronization on the reconfigurable device can achieve good performance improvement on a parallel processing.

1. はじめに

高性能パソコン，高機能組み込み機器（携帯端末，ユビキタスネットワークなど）の登場によって，更なるプログラムの高速化が必要となっている．プログラムの高速化技術は，アルゴリズムの改良などのソフトウェア側の試みや，マルチメディア向けの命令の高機能化やクロックアップなどのハードウェア側からの試みなどが行われている．その高速化技術の一つに並列処理がある．並列処理は，逐次プログラムから命令レベルの並列性を動的に抽出して行うスーパスカラ，逐次プログラムから静的に並列性を抽出して実行するVLIW，あるいは粗粒度の並列性を静的に抽出して実行するマルチプロセッサなどがある[1)]．いずれも逐次処理プログラムから並列性を抽出して，並列処理を行うことによってプログラムの高速化を図っている．

一方，半導体技術の進歩によって高集積なプログラマブルデバイスが登場し，これを用いたリコンフィギュラブルコンピューティングが盛んに行われている[2)]．リコンフィギュラブルコンピューティングは，プログラマブルデバイス上のハードウェアを容易に書き換えることが可能であり，プログラムに最適なハードウェアを提供できる．

そこで，並列処理とリコンフィギュラブルコンピューティングを融合した新しい並列処理パラダイムを提案する．具体的には，プログラムから様々な粒度の並列性を静的に抽出し，ハードウェア化してリコンフィギュラブルプロセッサ上で直接並列実行することによって，プログラムの高速化を図る．

本稿は，提案並列処理の実行方式の概要を示し，ハードウェア化の際に並列処理の協調動作に必要な機構と設計法について述べる．そして，設計法に基づいたC言語からVHDLへの変換について示す．次に，並列処理の効果と同期による性能への影響を評価するために予備実験を行った結果を示し，考察を行う．最後にむすびとする．

† 九州工業大学大学院 工学研究科 電気工学専攻 博士後期課程
Doctoral Course in Department of Electrical Engineering, Graduate School of Engineering, Kyushu Institute of Technology
†† 九州工業大学 工学部 電気工学科
Department of Electrical Engineering, Faculty of Engineering, Kyushu Institute of Technology

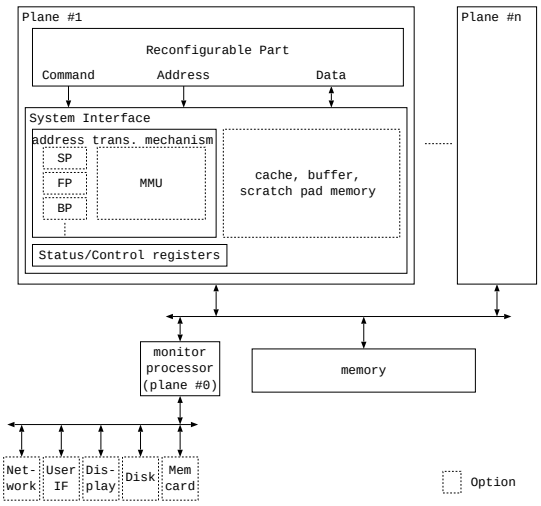

図 1 システムの概要
Fig. 1 Concept of system

2. 実行方式の概要

2.1 再構成可能コンピューティングシステム

図 1 に再構成可能コンピューティングシステムの概要を示す．システムは，n 個の再構成プレーン，モニタプロセッサ，メモリ，ディスク，およびユーザインターフェイスで構成する．再構成プレーンは，再構成部とシステムとの仲介を行うシステムインターフェイスを含む．ハードウェア化したプログラムはプレーンに割り当てられ処理を実行する．モニタプロセッサは，ユーザおよび入出力デバイスとのインターフェイスであり，プログラムに対するプレーンへの割り当てのスケジューリングも行う．

システムインターフェイスは，システムに伴ったアドレス変換機構を持つ．アドレス変換機構は，リニアアドレスを生成するためのベースレジスタ（MIPS プロセッサで示されるフレームポインタおよびスタックポインタ，あるいは x86 プロセッサで示されるベースレジスタおよびセグメントレジスタ）を備えることもでき，メモリ保護を行うための MMU（Memory Management Unit）を備えることもできる．また，バストラフィックを軽減するために，システムインターフェイスにはキャッシュ，バッファ，あるいはスクラッチパッドメモリのようなローカルのメモリデバイスを備えることもできる．

図 2 にプログラムのハードウェア化におけるコンパイルフローを示す．C 言語で記述したプログラムから並列性を抽出し，スレッドへ並列化する．そして，並列化した C 言語プログラムを VHDL に変換する．そ

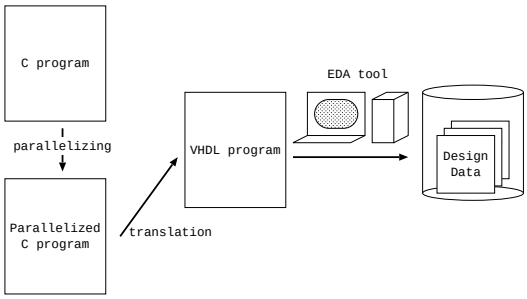

図 2 コンパイルフロー
Fig. 2 Compile flow

の際に，プログラムの変数に一意のアドレスを付ける．さらに，アドレス変換機構でのアドレス生成を指定するためのコマンドを付加する．EDA ツールを用いて再構成プレーンにロードする設計データを生成する．生成された設計データは，回路データ（汎用プロセッサのコード部に相当）およびデータ部を含む．

設計データはネットワークやディスクを介して，再構成プレーンにロードするときに，モニタプロセッサによってメモリに読み出される．バストラフィックを軽減するために，システムは設計データに対する専用のバスおよびメモリを備えることもできる．モニタプロセッサはシステムインターフェイス中の制御レジスタを通じて，プログラムが存在していないプレーンへ設計データをロードする．

再構成部上のプログラムが変数にアクセスする場合，再構成部は VHDL 変換時に付加されたアドレスとベースレジスタを指定するコマンドをシステムインターフェスに出力し，アドレス変換機構を用いて変数にアクセスする．

2.2 プログラムの並列化

プログラムは基本ブロックおよび繰り返しブロックを含んでいる．繰り返しブロックは，doall あるいは doacross として並列化することのできる最外周ループである．関数はインライン展開されているものとする．**図 3**（a）にプログラムの概要を示す．実線の矢印はデータ依存を表し，点線の矢印はは制御依存を表す．ブロックごとにデータ依存グラフを作成して，文レベルの並列性を抽出する．基本ブロック 1 において，S1 と S2 は依存関係がないため，それぞれ別々のスレッドに割り当てられ並列処理を行う．図 3（b）は，各基本ブロックの文がリストスケジューリング[3] を用いて，スレッドへ並列化される様子を示している．ハードウェア化するときには，並列処理を正確に実行するためにデータ依存および制御依存を保証する機構が必要である．

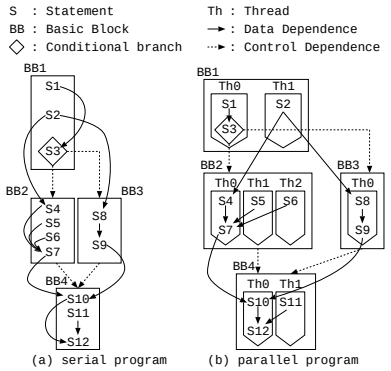

図 3 並列化の概要
Fig. 3 Image of parallelizing process

2.3 スレッドのハードウェア化

スレッドの各文の実行順序は，直線的に状態を遷移する単純な FSM（Finite State Machine）によって実現される．FSM のステートは，スレッドの文に対応する論理ステートである．**図 4** にハードウェア化の様子を示す．スレッド中の文は，ILP（Instruction Level Parallelism）[4] に相当する演算レベル並列性を抽出して実装する．文の演算を実行する演算器の遅延やハードウェア量とのトレードオフによって，実行に複数ステップを必要とする場合は，論理ステートを展開する．例えば，図 4 の S1 は S1a および S1b に展開し，S1a で a+b の演算を行って，S1b で a+b の演算結果と c との減算を行っている．また，乗算や除算のような演算完了までに複数ステップを必要とする場合は，演算回路からの完了を受けるまで，論理状態を繰り返す．例えば，図 4 の S2 の除算は，S2a で除算器に演算開始（start）を送信し，S2b で除算器からの演算完了（complete）を受けるまで S2b を繰り返し遷移する．

以上のような演算実行に必要なステートに加え，プログラムを正確に並列実行するためのステートを追加する．各ブロックにおいて，データ依存のある文はスレッド間で同期を取る必要がある．例えば，図 3（b）の S7 は S4，S5，および S6 の完了を待たなければならない．このため，図 4（b）に示す様に S4 と S7 の間に同期ステート（Sync）を挿入する．S5 および S6 はフラグによって演算の完了をスレッド 0（Th0）へ知らせる．

後続ブロックは，先行ブロックが完了し，条件が決定するまで実行することができない．このため，図 4（c）に示す様に各ブロック中の FSM の先頭に起動ステート（Invc）を挿入する．プログラムの先頭の基本ブロック（BB1）は，システムインターフェイスによってアサートされる信号によって起動する．それ以降のブロックは，直前の先行ブロック内のスレッドの完了と条件文の結果によって起動する．スレッドは実行終了後，起動ステートへ遷移し次の起動を待つ．

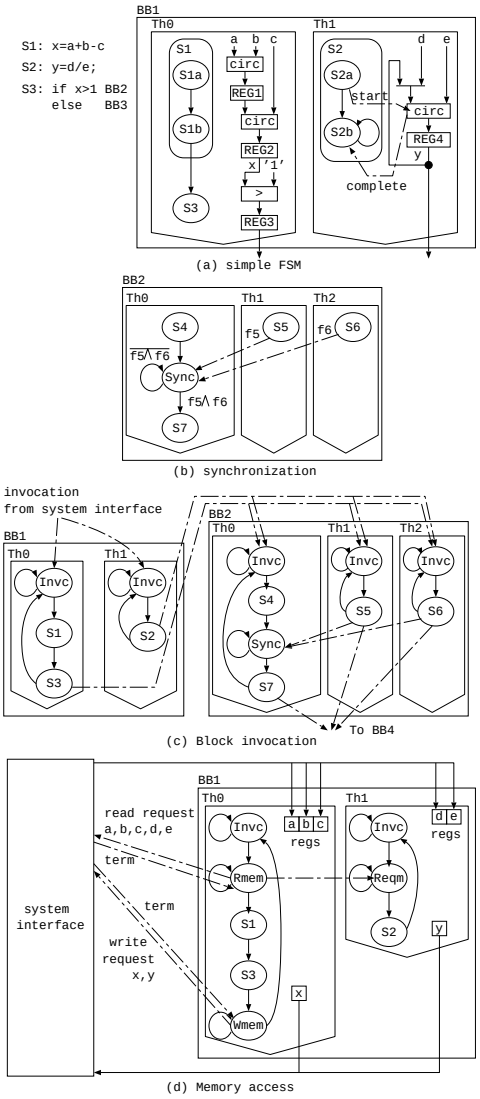

図 4 ハードウェア化の様子
Fig. 4 Image of creating hardware thread

スレッドは，演算の実行に必要なデータを読み書きするためにメモリ上の変数へアクセスする．メモリアクセスを軽減するために，Th0 の起動直後にそのブロックで使用する変数を読み出す．このため，メモリ読み出しステート（Rmem）は Th0 の起動ステート（Invc）直後に挿入する．Th0 は，読み出した変数を再構成部内のレジスタへ適切に格納する．他のスレッドは，レジスタへ変数が格納されるまでメモリ要求

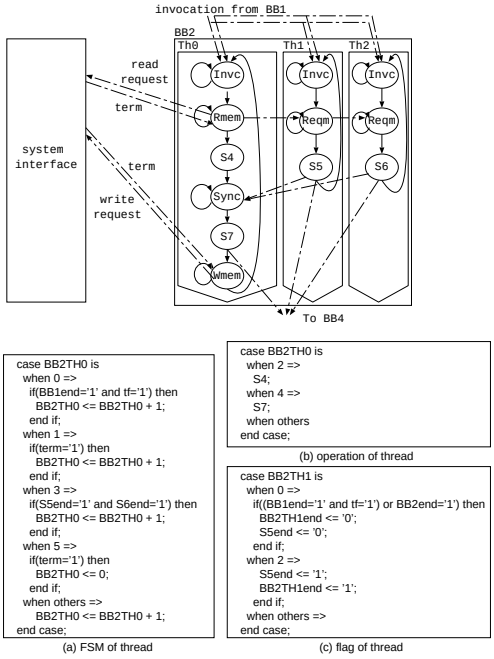

図 5 VHDL 記述
Fig. 5 VHDL description

ステート（Reqm）で待つ．アドレスがブロックを実行するまでに決定しない変数は，変数を使用する演算の直前に Rmem を挿入しメモリから変数を読み出す．メモリの一貫性を維持するために，Th0 は使用した変数をメモリに書き込む．メモリ書き込みステート（Wmem）は Th0 の末端に挿入し，ブロック内の変数をメモリに書き込む．

3. VHDL への変換

スレッド内の演算の実行順序を制御する FSM，式を実行する演算回路，およびスレッド間で協調動作を行うためのフラグを VHDL を用いて記述する．FSM はカウンタによって実現される．**図 5** に，VHDL の記述を示す．`BB2TH0` はカウンタ値であり，カウンタ値をインクリメントする（`BB2TH0<=BB2TH0+1`）ことによって次状態に遷移する．カウンタ値 0 は，起動ステートの記述である．BB2 の起動条件は制御フローグラフより，BB1 の完了かつ条件の成立であり if 文に記述する．カウンタ値 1 は，メモリ読み出しステートの記述である．システムインターフェイスからのデータ読み出し完了を待つ．カウンタ値 3 は，同期ステートの記述である．S7 はデータ依存グラフより，S5 および S6 の完了を待たなければならない．if 文には S5 および S6 の完了フラグを記述する．カウンタ値 5 は，メモリ書き込みステートの記述である．システムインターフェイスからのデータ書き込み完了を待つ．また，カウンタ値 5 はスレッドの最後に実行する文であるため，カウンタ値を 0 にセットし起動ステートに遷移する．BB1TH1 の S2b の様な文の実行に複数ステップを実行する場合も，同様に if 文を用いて記述する．

この FSM を基準にして，文を実行する演算を記述する．図 5（b）に文の実行の記述を示す．カウンタ値 2 のとき，S4 の演算を記述する．

図 5（c）に BB2TH1 のフラグの記述を示す．フラグは，スレッドの完了を表すフラグと文の実行完了を表すフラグを記述する．各フラグは，ブロックの起動時とブロックの完了時に 0 にリセットする．カウンタ値 2 で S5 を実行し，S5 の完了を表すフラグ（`S5end`）を 1 にセットする．また，スレッドの最後に実行する文であるため，スレッドの完了を表すフラグ（`BB2TH1end`）を 1 にセットする．

4. 予 備 実 験

4.1 実 験 環 境

これまでに示したように，ハードウェア化したスレッドでは文に対する実行ステートに加え，スレッド間の協調動作のための同期および起動ステートが必要である．メモリアクセスステートは，システムインターフェイス中のキャッシュメモリ，使用するメモリデバイス，あるいはシステムバス構成のような実装に依存する．予備実験では，実装に依存しない部分である同期および起動ステートの挿入による性能への影響を評価する．具体的には，プログラムに対し並列化を施したとき（パラレル版）の実行時間と並列化を施さないとき（シリアル版）の実行時間を比較する．シリアル版ではプログラムの最上ブロックにのみ起動ステートがあり，その後に続くブロックには同期および起動ステートは存在しない．

使用したプログラムは，GCD，FIBO および QSORT である．GCD は 2 個の整数の最大公約数を求めるプログラム，FIBO はフィボナッチ数列の第 n 項を求めるプログラムであり，QSORT はクイックソートアルゴリズムを用いた整数ソートプログラムである．実験では，プログラムから静的に文レベルの並列性を抽出し，リストスケジューリングによって文をスレッドへ割り当てた．そして，VHDL へ変換し，EDA ツールを用いて FPGA 向けのビットストリームを生成した．使用した FPGA は，Xilinx XC4020E である．FPGA は図 1 で示したモニタプロセッサと同等のスカラプロセッサと回路図を用いて別の FPGA に構成したローディングロジックとで構成している．

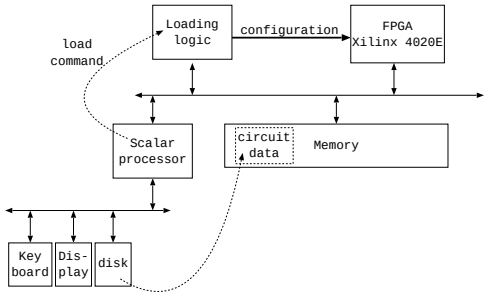

図 6 テストベッドの構成
Fig. 6 Organization of the testbed machine

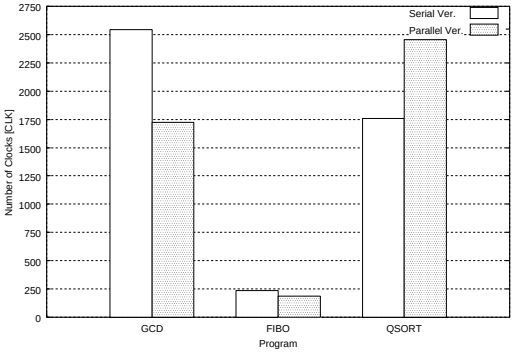

図 8 実験結果
Fig. 8 Result of the experiment

図 7 テストベッドの外観
Fig. 7 Picture of the testbed machine

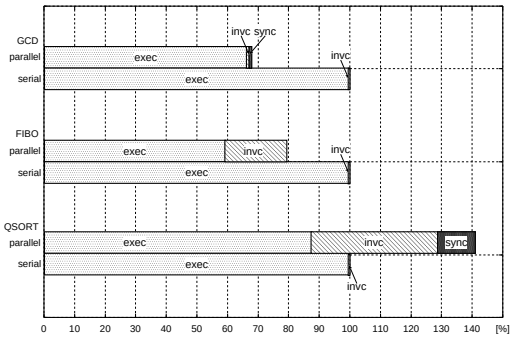

図 9 実行時間の内訳
Fig. 9 Breakdown of execution time

図 6 にテストベッドの構成を示し，**図 7** にその外観を示す．テストベッドは，スカラプロセッサ，ローディングロジック，FPGA，メモリおよびディスクで構成する．そして，ユーザインターフェイスとしてキーボードおよびディスプレイを持つ．

但し，予備実験では同期および起動ステートの性能への影響について評価するため，プログラムで使用するデータをあらかじめ FPGA 内の設計データに含めた状態で実験を行った．

4.2 結果および考察

図 8 に各プログラムの処理にかかったクロック数の並列版および逐次版を示す．逐次版に対する速度向上比は，GCD は 1.47 倍，FIBO は 1.26 倍，QSORT は 0.70 倍であった．GCD および FIBO は逐次版に比べて速度向上が見られたが，QSORT では並列版で速度が低下している．

ここで，同期および起動ステートが無い場合を理想のハードウェア（理想版）としたときの，理想版との速度向上比は GCD は 1.51 倍，FIBO は 1.68，QSORT は 1.15 倍である．GCD は，理想版と並列版では遜色ないが，FIBO および QSORT では並列処理による効果がかなり低減している．この原因を明らかにするために，**図 9** に逐次版を 100%としたときの実行時間の内訳を示す．sync は同期によって消費した割合，invc はブロックの起動によって消費した割合である．exec は文の実行に消費した割合である．GCD では全体の実行時間と比べて，同期および起動ステートの実行時間の割合が 2.7%であった．これが FIBO では 25.3%であり，QSORT においては 39.0%であった．同期及び起動ステートが全体の実行時間の数%と小さかったため GCD では理想的な速度向上が得られた．一方，FIBO は exec で速度向上したが，invc も増加したため速度向上が 1.26 倍に留まった．QSORT においても exec で速度向上したが，invc および sync が逐次版の実行時間よりも増加し，速度低下に繋がった．

速度向上に関して，それぞれのプログラムの特性を明らかにするために，**表 1** に設計したハードウェアにおける論理ステートの内訳を示す． 全ステート数に対する exec の割合は 3 種のプログラムとも同じぐらいであるが，GCD は 32 ビットの乗算および除算を含んでおり，演算を終えるまで論理ステートを繰り返し遷移する．これによって，同期および起動ステート

表 1 論理ステートの内訳
Table 1 Breakdown of logical states

Program	exec	sync	invc
GCD	8	2	4
FIBO	4	0	2
QSORT	21	5	10

表 2 ゲート数
Table 2 Gate count

Program	serial	parallel
GCD	18,101	18,610
FIBO	2,393	2,320
QSORT	9,462	12,656

表 3 フラグ，スレッドおよび基本ブロック数
Table 3 Flag, thread and basic block count

Program	flag	thread	basic block
GCD	14	12	4
FIBO	5	5	2
QSORT	27	19	11

のオーバヘッドが全体の実行時間に対して小さくなった．FIBO および QSORT は，乗算や除算のような論理ステートを繰り返す演算を含んでいないため，論理ステートの割合がそのまま実行時間の内訳と等しくなった．演算実行の粒度が大きいプログラムでは，同期によるオーバヘッドが小さくなり，理想的な並列処理による効果が得られる．一方，粒度が小さいプログラムにおいては，同期および起動ステートの挿入方法に関してより効率的な方法を検討する必要がある．

ハードウェア量の増加ついて，EDA ツール（Xilinx ISE）によって生成されたレポートを用いて逐次版と並列版のゲート数を**表 2** に示す．逐次版に対するゲート数の増加量は，GCD は 3%，FIBO は 3%，QSORT は 33%であった．GCD および FIBO は，少量のハードウェアの増加で速度向上が得られた．QSORT は，GCD および FIBO と比較してかなり増加している．**表 3** に実装したフラグ，スレッドおよび基本ブロックの数を示す．QSORT は，表 1 の論理ステート数，フラグ数，およびスレッド数が GCD および FIBO に比べて非常に多くなったためにゲート数が増加した．また，QSORT は基本ブロック数が多いためフラグ数が増加した．QSORT の様な基本ブロック数が比較的多いプログラムでは，ブロックを融合するような設計方法について検討する必要がある．それに伴って，invc の時間が小さくなり速度に対して良い結果をもたらすことが期待される．

5. むすび

本稿では，リコンフィギュラブルコンピューティングを利用した新しい並列処理パラダイムを提案した．提案パラダイムでは，C 言語プログラムから静的に並列性を抽出してスレッドに割り当てる．そして，並列化したプログラムを VHDL に変換して，再構成システム上で直接並列実行する．スレッドには文単位で割り当てられ，文に対応するステートを持った単純な FSM によって実現する．文に対するステートに加えて，独立に実行するハードウェアスレッドは他のスレッドと協調動作するために同期ステート，起動ステートおよびメモリアクセスステートを必要とする．同期ステートはハードウェアスレッド間のデータ依存を保証し，起動ステートはブロック間の制御依存を保証する．メモリアクセスステートは，メモリへの変数アクセスを待つためのステートである．

同期および起動ステートの性能への影響を明らかにするために，実機上でプログラムを用いて予備実験を行った．その結果，再構成デバイス上の低オーバヘッドの同期および通信において並列処理に良い効果が得られた．しかし，アプリケーションの実行粒度が同期によるオーバヘッドに対して比較的小さい場合，性能をより改善するためにハードウェアスレッドのより効果的な設計法の検討が必要であることが解った．

今後は，メモリアクセスを考慮した実験を行い，同期および起動ステートを含めて更に効果的な設計法について検討する．また，提案した並列処理パラダイムを実現する実システムの開発を行っていく．

参考文献

1) Patterson, D. A. and Hennessy, J. L.: *Computer architecture: a quantitative approach; third edition*, Morgan Kaufmann Publishers Inc. (2002).
2) Compton, K. and Hauck, S.: Reconfigurable computing: a survey of systems and software, *ACM Comput. Surv.*, Vol.34, No.2, pp.171–210 (2002).
3) El-Rewini, H., Lewis, T. G. and Ali, H. H.: *Task scheduling in parallel and distributed systems*, Prentice-Hall, Inc. (1994).
4) McFarland, M. C., Parker, A. C. and Camposano, R.: Tutorial on high-level synthesis, *Proceedings of the 25th ACM/IEEE conference on Design automation*, IEEE Computer Society Press, pp. 330–336 (1988).