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.

CPU 密結合型アクセラレータの機械語プログラムからの自動合成

田村　真平† 　石浦菜岐佐† 　神原　弘之†† 　冨山　宏之†††

† 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 番地
††† 立命館大学 理工学部 〒 525–8577 滋賀県草津市野路東 1 丁目 1-1

あらまし　本稿では, 機械語プログラムの指定区間を CPU 密結合型アクセラレータに合成する手法を提案する. CPU 密結合型アクセラレータは, CPU のプログラムカウンタが特定番地に達すると起動し, 処理が終わるとプログラムカウンタに復帰番地を書き込むことにより CPU に制御を戻す. また, アクセラレータは CPU のレジスタファイルやメモリに直接アクセスすることにより CPU とのデータの授受を行う. 本手法では, 機械語プログラムの指定部分を制御付きデータフローグラフに変換し, これにレジスタファイルアクセス演算やプログラムカウンタ更新演算を挿入した後, 高位合成エンジンによりハードウェアを合成する. 本稿では, 機械語プログラム全体にデータフロー解析を行って, 不要なレジスタファイルアクセス演算の挿入を抑制するとともに, アクセラレータと CPU の実行切り替え時のパイプラインの状態まで考慮した効率的な演算のスケジューリングを行う. 本手法を高位合成システム ACAP に実装し, 評価実験を行った. その結果, CPU にその約 0.5 倍から約 1.4 倍のハードウェアを追加することにより, プログラム全体の実行速度を約 1.5 倍から 3 倍に高速化することができた.

キーワード　高位合成, 機械語プログラム, ハードウェア/ソフトウェア協調設計, CPU 密結合型アクセラレータ

Binary Synthesis of Hardware Accelerator Tightly Coupled with CPU

Shimpei TAMURA†, Nagisa ISHIURA†, Hiroyuki KANBARA††, and Hiroyuki TOMIYAMA†††

† Kwansei Gakuin University, 2-1 Gakuen, Sanda, Hyogo, 669–1337, Japan
†† ASTEM RI/KYOTO, 134 Chudoji Minamimachi, Shimogyo-ku, Kyoto, 600–8813, Japan
††† Ritsumeikan University, 1-1-1 Noji-Higashi Kusatsu, Shiga, 525–8577, Japan

Abstract　This article presents a method of synthesizing hardware that accelerates specified sections of binary programs. The accelerator is tightly coupled with a CPU; it watches the program counter of the CPU to start execution when the specified addresses are reached, it returns control to the CPU by rewriting the program counter. It also shares data with CPU by directly accessing the register file and the main memory. In our method, operations for accessing the register file and the program counter are added to a control dataflow graph (CDFG) derived from the specified machine code segments, which is fed into the conventional high-level synthesis flow. CDFGs are optimized by 1) removing redundant register file access operations based on dataflow analysis of the entire machine program, and 2) by scheduling operations considering the pipeline status of the CPU. The proposed method has been implemented on top of the ACAP high-level synthesizer. The experimental results show that the entire program execution speed was accelerated by 1.5 to 3.0 times at the cost of 50% to 140% increase in the hardware size.

Key words　High-Level Synthesis, machine language program, hardware/software codesign, hardware accelerator tightly coupled with CPU

1. はじめに

近年, 組み込みシステムには, ますます高い機能が要求される一方で, ハードウェアの面積や消費電力等の制約はより一層厳しくなりつつある. そのため, いかに面積や消費電力を抑制しつつ高機能かつ高速なシステムを設計するかが重要な課題となっている. これに対するアプローチとして, 既存のソフトウェアの全体あるいは一部の処理を高位合成 [1] 等によりハードウェア化し, システムの高速化や低消費電力化を図る研究が行われている.

このアプローチに基づく研究として, 特殊レジスタにより起動制御を行うハードウェア [2], 通信用インタフェースを介して起動制御を行うハードウェア [3], メインメモリを介してデータ授受やハードウェア制御を行うハードウェア [4] が提案されているが, ポーリングやデータハザードの回避のためのオーバーヘッドが課題として挙げられる. また, プロセッサに組み込み可能なカスタム命令等を合成する手法 [5], [6] も提案されているが, ハードウェアに合成可能な動作記述の制約が課題となる. これらいずれの手法においても, 機械語プログラムやコンパイラを修正することが必要となる.

これに対し, 戸田ら [7]〜[9] は CPU と密に結合するアクセラレータに基づくハードウェア/ソフトウェア協調設計手法を提案している. この手法は機械語プログラムの指定の区間をハードウェアアクセラレータ化するものである. このアクセラレータはプログラムカウンタの監視および更新により, ソフトウェアとの制御を素早く切り替える. また, メインメモリだけでなく CPU のレジスタファイルにアクセスすることにより, CPU とデータを高速に授受する. アクセラレータの元となったプログラムやコンパイラは更新せずにそのまま利用できる. しかし, 文献 [7]〜[9] では, アクセラレータの設計は自動化されておらず, 手動で行われていた.

そこで本稿では, 与えられた機械語プログラムの指定区間を CPU 密結合型アクセラレータに自動合成する手法を提案する. 本手法では, 機械語プログラムから生成した CDFG (Control Data Flow Graph) に対し, アクセラレータから CPU のレジスタファイルにアクセスするための演算と, プログラムカウンタを更新するための演算を挿入した後, 高位合成によりハードウェアを合成する. さらに本研究では, ソフトウェア部分を含むプログラム全体の解析を行うことにより, 不要なレジスタファイルアクセス演算の削除, およびソフトウェア部分を考慮したスケジューリングを行う. これにより, CPU とアクセラレータの間のデータの授受や制御の切り替えによるオーバーヘッドの削減を図る.

本手法を高位合成システム ACAP [10] に実装し, いくつかのプログラムの一部をアクセラレータ化する実験を行った. その結果, CPU 単体で実行した場合に比べて, CPU の約 0.5 倍から約 1.4 倍の回路規模の増大で, プログラム全体の実行速度を約 1.5 倍から 3 倍に高速化することができた.

2. CPU 密結合型アクセラレータ

戸田らが提案した CPU 密結合型アクセラレータ [7]〜[9] (以下, アクセラレータ) に基づくハードウェア/ソフトウェア協調設計手法は, 機械語プログラムの指定した区間の処理をハードウェア化するものである. アクセラレータの構成を図 1 に示す.

アクセラレータは CPU のプログラムカウンタ (PC) の値を常に監視しており, プログラムカウンタの値がハードウェア化した区間の先頭アドレスに到達すると処理を開始する. アクセラレータの処理中, アクセラレータはプログラムカウンタの値を固定し, CPU の命令レジスタ (IR) に NOP 命令を供給することによって, CPU の動作を停止させる. アクセラレータは処理の終わりにプログラムカウンタに復帰番地を書き込み, これにより, CPU は処理を再開する.

アクセラレータはメインメモリや CPU のレジスタファイル (RF) に直接アクセスすることにより, CPU とデータ共有を行う. また, フォワーディングユニットへのアクセスにより, ソフトウェア依存のデータハザードを軽減し, データ取得に要する時間をさらに短縮する [7], [9].

アクセラレータは, 起動/終了やデータ授受のための特殊な命令や特殊なレジスタを用いることなく, CPU との実行を高速に切り替えることができる. また, 機械語の複数区間をアクセラレータ化することや, ソフトウェアの分岐命令によりアクセラレータ化した区間の途中に遷移することも可能である [8]. アクセラレータを CPU と連動させるために, 機械語やコンパイラに一切変更を加えなくてもよいことも 1 つの利点である.

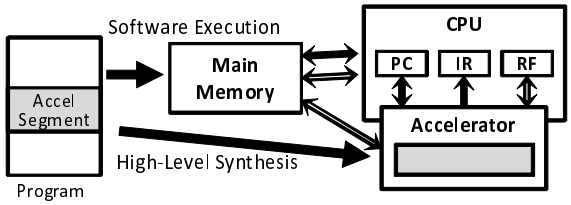


図 1　CPU 密結合型アクセラレータの構成

3. アクセラレータの自動合成

3.1 概　　要

本稿では CPU 密結合型アクセラレータを自動合成する手法を提案する.

本手法では, まず, 与えられた機械語プログラムの指定区間を切り出して CDFG に変換する. この CDFG では, プログラムに書かれたレジスタファイルへのアクセスは全てハードウェアのローカルレジスタへのアクセスに置換されているため, アクセラレータが CPU とデータを授受できるように, 「レジスタファイルアクセス演算」を CDFG に挿入する. 次に, アクセラレータの終了時にプログラムカウンタの値を復帰番地に更新する「プログラムカウンタ更新演算」を CDFG に挿入する. その後, その CDFG に対して一般的な高位合成のスケジューリングとバインディングを行って, ハードウェアを合成する.

3.2 レジスタファイルアクセス演算

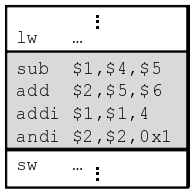
レジスタファイルアクセス演算の挿入の例を図 2 に示す. 図 2 (a) は機械語プログラムで, 灰色の区間をアクセラレータ化するものとする. 図 2 (b) は (a) の指定区間を CDFG に変換したものであり, 1 つの DFG (Data Flow Graph) からなる. この時点では, 図 2 (a) の レジスタファイル `$1` 〜 `$6` へのアクセスは, 図 2 (b) のようにローカルレジスタへのアクセスに置換される.

この DFG を解析することにより, `$4`, `$5`, `$6` のデータが DFG の入口で必要であることが分かる. そこで, 図 2 (c) のように CPU のレジスタファイルを読み出す演算 (R 演算) を挿入する. `$1`, `$2` は DFG 内で定義されるため, `$1`, `$2` のレジスタファイル読み出し演算は挿入しない.

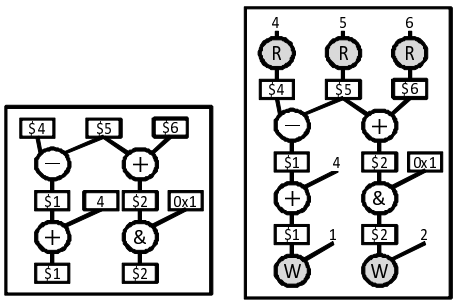
また, DFG の解析から, この DFG が `$1`, `$2` の値を定義していることが分かる. これらのデータを CPU に返すため, 図 2 (c) のようにレジスタファイル書き込み演算 (W 演算) を挿入する. 同じレジスタの値が複数回定義されている場合には, 最後の定義のみを書き込む.

3.3 プログラムカウンタ更新演算

プログラムカウンタ更新演算の挿入の例を図 3 に示す. PW はプログラムカウンタ更新演算を表し, 入力された値を CPU のプログラムカウンタに書き込む. 例えば, 図 3 (a) のプログラムは, 灰色の部分をアクセラレータで実行した後, A 番地から CPU の実行を再開するものとする. この場合には, 図 3 (b) のように, アクセラレータの CDFG の最後の DFG に A の番地を書き込むプログラムカウンタ更新演算を挿入する. 挿入した



(a) 機械語プログラム

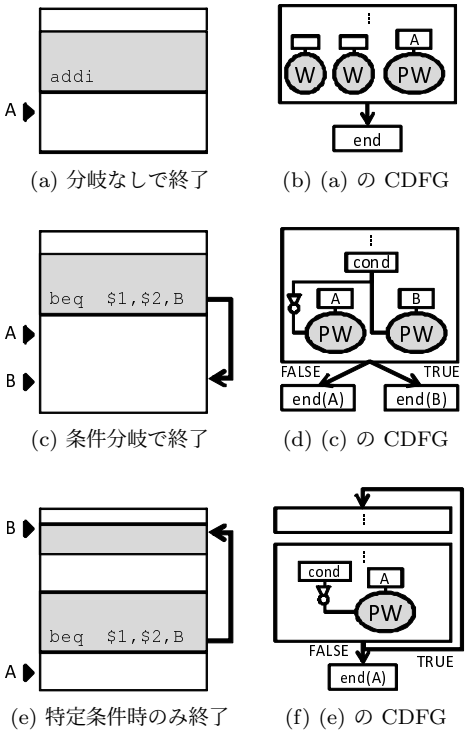


(b) DFG (演算挿入前)　(c) DFG (演算挿入後)

図 2　レジスタファイルアクセス演算の挿入

プログラムカウンタ更新演算は, アクセラレータの処理終了段階で実行するようにスケジューリングするが, その詳細は 3.4 節で述べる.

アクセラレータの処理が条件分岐で終了する場合 (図 3 (c)) には, 図 3 (d) のように, 条件によって異なる復帰番地を書き込むプログラムカウンタ更新演算を 2 個挿入する. 分岐先がアクセラレータ化区間の場合には, CPU の動作を再開させないため, プログラムカウンタ更新演算を挿入する必要はない. 従って, 分岐先がアクセラレータ化区間とソフトウェア区間に分かれる場合 (図 3 (e)) には, 図 3 (f) のように, アクセラレータが終了する条件に限って復帰番地を書き込むプログラムカウンタ更新演算を挿入する.



(a) 分岐なしで終了　(b) (a) の CDFG

(c) 条件分岐で終了　(d) (c) の CDFG

(e) 特定条件時のみ終了　(f) (e) の CDFG

図 3　プログラムカウンタ更新演算の挿入

3.4　スケジューリング

3.4.1　アクセラレータ用演算の扱い

各 DFG に挿入したレジスタファイルアクセス演算やプログラムカウンタ更新演算は, 他の演算と同様にスケジューリングする. レジスタファイル読み出し演算は, 1 サイクルにレジスタファイルの入力ポートの数まで, レジスタファイル書き込み演算はレジスタファイルの出力ポートの数まで同時実行が可能である. スケジューリングおよびバインディングでは, 各種演算専用の擬似的な演算器があるものとして処理する.

3.4.2　アクセラレータ起動時の制約

CPU が命令パイプラインを採用している場合, スケジューリング時に CPU との干渉を考慮する必要がある.

アクセラレータの起動直後は, CPU でパイプラインに残っている命令がレジスタファイルやメインメモリにアクセスしていたり, 実行結果がライトバックされていなかったりする可能性がある. このため, レジスタファイルアクセス演算およびロード/ストア演算は, これを考慮してスケジューリングする必要がある.

ロード/ストア演算のスケジューリング制約を図 4 に示す. ここでは, 5 段パイプラインの CPU を想定している. I1, I2, I3 はアクセラレータが起動する直前の 3 命令を表す. CPU は 命令 I3 を読み込んだ後, その IF ステージでプログラムカウンタの値を更新する. アクセラレータは, 命令 I3 の ID ステージで, プログラムカウンタの監視 (PR) によりその値を検知し, その次の s1 サイクルから処理を開始する. s1 のサイクルでは 命令 I3 が, s2 のサイクルでは命令 I2 が MEM ステージの処理を行っている. これらのサイクルでは CPU がメインメモリにアクセスしている可能性があるため, ロード/ストア演算は s3 サイクル以降にスケジューリングする必要がある. I2 や I3 がメモリアクセスを行わない命令であっても, アクセラレータ化区間の DFG だけを見る限りではそれを確認できないため, 悲観的にスケジューリングを行わなければならない.

レジスタファイルアクセス演算にも同様の制約が生じる. アクセラレータ起動時のスケジューリング制約は表 1 のようになる.

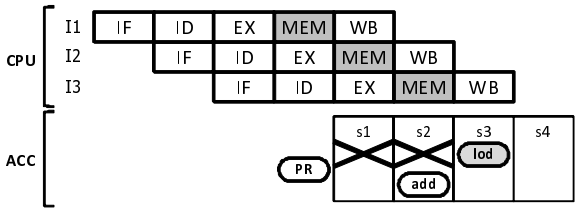


図 4　ロード/ストア演算のスケジューリング制約

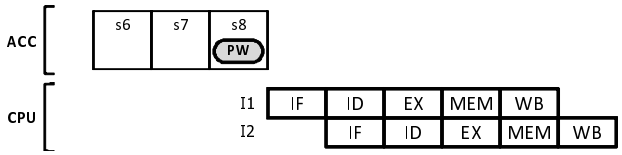
表 1　アクセラレータ起動時のスケジューリング制約

演算	制約
ロード/ストア演算	3 サイクル目以降
レジスタファイル読み出し演算	2 サイクル目以降
レジスタファイル書き込み演算	4 サイクル目以降

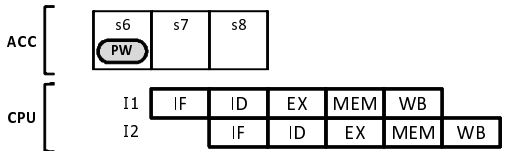
3.4.3　アクセラレータ終了時の制約

プログラムカウンタ更新演算 (PW) をアクセラレータの最後のサイクルで実行すると, 図 5 (a) のように, CPU はその次のサイクルから動作を再開する. これに対し, 図 5 (b) のように, プログラムカウンタ更新演算を最後より前のサイクルに実行すると, より早く CPU の動作を再開することができる. しかし,

この場合, s8 サイクルが I1 の ID ステージと同じタイミングになり, レジスタファイル読み出しの競合が発生する可能性がある. プログラムカウンタ更新演算は, 他の命令の競合が発生しない範囲で, できる限り早期に実行できるようスケジューリングする必要がある.



(a) プログラムカウンタ更新演算を最後に実行した場合



(b) プログラムカウンタ更新演算を途中で実行した場合

図 5　プログラムカウンタ更新演算と CPU の動作再開

レジスタ書き込み演算やプログラムカウンタ更新演算に対しても同様の制約が生じる. プログラムカウンタ更新演算に関連するスケジューリング制約は表 2 のようになる.

表 2　プログラムカウンタ更新演算のスケジューリング制約

演算	プログラムカウンタ演算の制約
ロード/ストア演算	最後の該当演算の 2 サイクル前以降
レジスタファイル読み出し演算	最後の該当演算の 1 サイクル前以降
レジスタファイル書き込み演算	最後の該当演算の 3 サイクル前以降

3.5　制御部の生成

アクセラレータは有限状態機械で制御する. 機械語の 2 つの区間をアクセラレータ化した場合の制御フローの例を図 6 に示す. 状態 1, 2, 3 が 1 つ目のアクセラレータ化区間, 状態 4, 5, 6, 7 が 2 つ目のアクセラレータ化区間を制御するものであるとする. addr1, addr2 はそれぞれ区間の先頭アドレスである. 状態 0 ではプログラムカウンタを監視しており, その値が addr1 に等しければ状態 1 に遷移し, addr2 に等しければ状態 5 に遷移する. また, アクセラレータの最終状態 (状態 3, 7) でも, 状態 0 と同様の判定で遷移する.

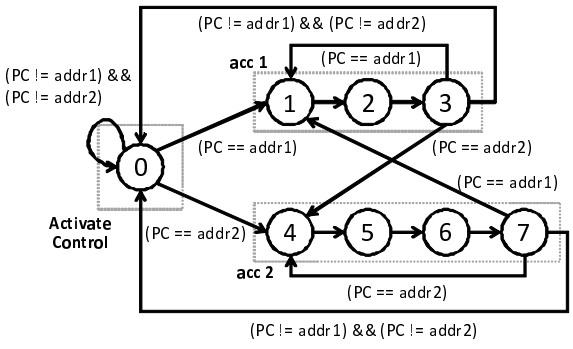
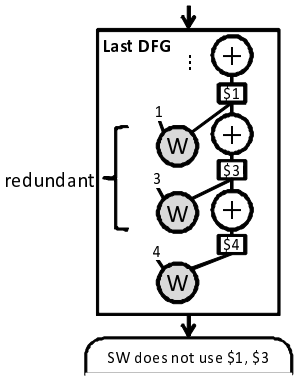


図 6　アクセラレータの制御フロー

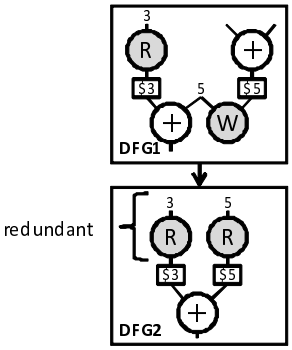
3.6　課　　題

本章の手法によるアクセラレータ合成には課題が 2 点ある.

1 点は, 実際には不要なレジスタファイルアクセス演算が挿入され, 実行サイクル数が増加することである. 例えば, 図 7 (a) には, $1, $3, $4 の値が定義されているが, これらの値が, 以降で使われるとは限らない. しかし, アクセラレータ化区間の情報だけでは, そのデータが使われるかどうかを判定できないため, DFG で値を定義している全てのレジスタに対してレジスタファイル書き込み演算を挿入する必要がある. また, レジスタファイルアクセス演算の挿入を DFG 単位で行うため, 図 7 (b) のように, DFG1 で読み出した $3 を次の DFG2 で再び読み出したり, DFG1 で定義した $5 レジスタを次の DFG2 で読み直したり等の冗長な処理が発生する.



(a) 不要データの書き込み



(b) データの不要な読み出し

図 7　レジスタファイルアクセス演算によるオーバーヘッドの増大

もう 1 点は, スケジューリング時の必要以上の待ち合わせの発生である. 例えば, 図 4 (a) の I3 がメモリアクセス命令ではない場合, ロード演算は 2 サイクル目に実行しても支障はない. しかし, アクセラレータ化する区間の解析だけではそれを判定できないため, ロード演算は必ず 3 サイクル目にスケジューリングせざるを得ない.

4.　機械語プログラムの全域解析による最適化

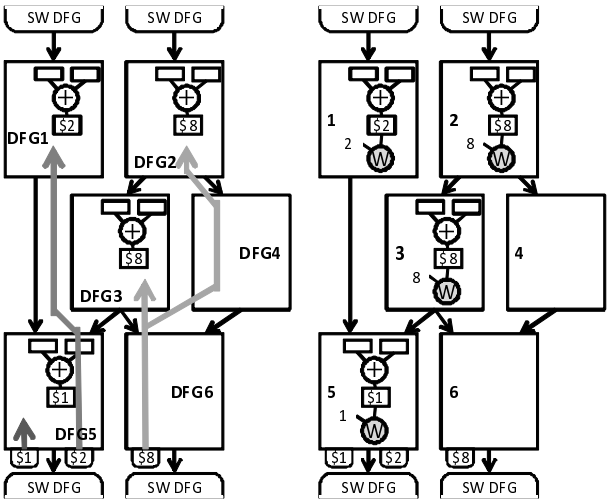
本稿では, 3.6 節の課題を解決するため, アクセラレータ化しない部分を含む機械語プログラム全域の解析に基づく最適化手法を提案する. 本手法では不要なレジスタファイルアクセスの抑制と, ソフトウェアとの依存関係を考慮したスケジューリングを行い, オーバーヘッドの削減を図る.

4.1　不要なレジスタファイルアクセス演算の抑制

例えば, 3.6 節の図 7 (a) において, アクセラレータの終了後に $4 レジスタのみ参照する可能性があることが分かれば, $1, $3 のレジスタファイル書き込み演算は挿入しなくて済む.

本手法では, アクセラレータ実行終了後に参照する可能性があるレジスタの集合を, プログラム全体のデータフロー解析により求め, これに基づいて不要なレジスタファイル書き込み演算の挿入を抑制する. 例えば, 図 8 (a) において, DFG5 の後に $1, $2 レジスタ, DFG6 の後に $8 レジスタのみが参照されることが分かったとする. すると, DFG5 と DFG6 より逆方向に DFG をたどることにより, $1, $2, $8 を最後に定義している箇所を同定できるため, この箇所にのみレジスタファイル書き込み演算を挿入する. 図 8 (b) がこの結果得られる CDFG である.

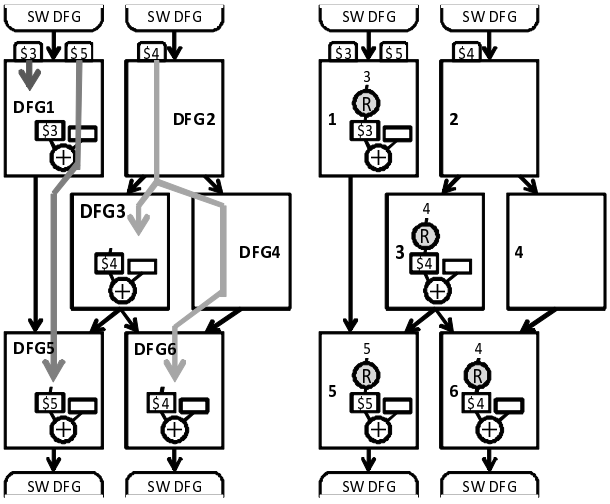
冗長なレジスタファイル読み出し演算の挿入も同様にして抑制することができる. データフロー解析により, アクセラレータで参照されるレジスタが図 9 (a) のように $3, $5, $4 と分かったとする. DFG1 と DFG2 より DFG をたどることにより, これらのレジスタが最初に参照される箇所が同定できるため, この箇所にのみレジスタファイル読み出し演算を挿入する



(a) 挿入前　　(b) 挿入後

図 8　レジスタファイル書き込み演算の挿入

(図 9 (b)). ただし, レジスタファイルアクセス演算を挿入する際, 前後の DFG との整合性を確保する必要がある. 例えば, 図 10 のように, DFG3 にソフトウェアとアクセラレータの両方の遷移がある場合を考える. ソフトウェアで定義される $4 の値を読み出すために, 図 10 (a) のように DFG3 にレジスタファイル読み出し演算を挿入するが, これによって, DFG1 で定義される $4 の値が失われてしまう. これを防ぐため, DFG3 に $4 レジスタファイル読み出し演算を挿入する際に, その遷移元をたどり, $4 レジスタを定義する DFG1 に $4 のレジスタファイル書き込み演算を挿入する (図 10 (b)).



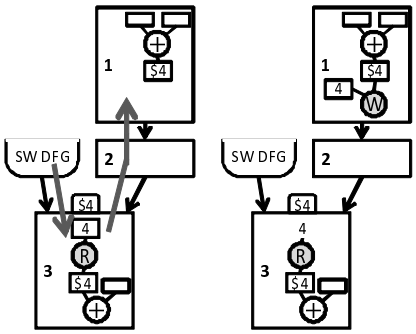
(a) 挿入前　　(b) 挿入後

図 9　レジスタファイル読み出し演算の挿入

4.2　ソフトウェアとの依存関係を考慮したスケジューリング

3.6 節の 2 つ目の課題は, ソフトウェア部分で実行される命令を把握することにより解決できる. これは, プログラム全体を CDFG に変換する際に, アクセラレータ起動の直前のソフトウェアで実行される数命令, およびアクセラレータ終了の直後のソフトウェアで実行される数命令のリストを取得することにより実現できる. 必要な命令数は CPU のパイプライン構成に依存して決まる.

図 11 に例を示す. プログラム全体を CDFG に変換する際, SW DFG1 のように, 元となったプログラムがいつメモリやレジスタにアクセスするかの情報を記憶する. その後, アクセラレータ化 DFG (Acc DFG) をスケジューリングする直前に, その遷移前の SW DFG1, SW DFG2 のアクセス情報を “Access Top” に集約し, その遷移後の SW DFG5 のアクセス情報を “Access Bottom” に集約する. 5 段パイプラインの場合, 遷移前 DFG の最後の 3 命令分, 遷移後 DFG の先頭から 3 命令分のアクセス情報を集約すればよい. この例では, SW DFG2 には 2 命令しかないため, さらに前の SW DFG3, SW DFG4 も調べ, それぞれ最後の 1 命令分のアクセス情報を集約する. そして, この情報を基に Acc DFG のスケジューリングを行う.



(a) 整合性の確認　　(b) 整合性確保後

図 10　レジスタファイルアクセス演算の整合性確保

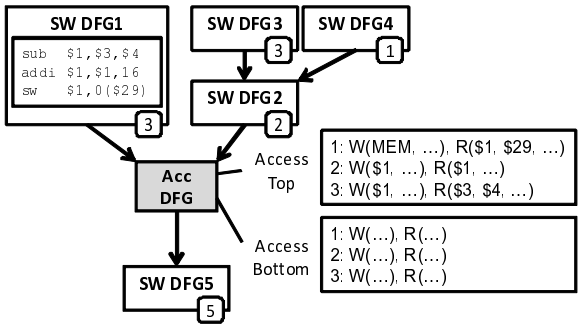


図 11　メモリおよびレジスタのアクセス情報の記憶と集約

ロード/ストア演算およびレジスタファイルアクセス演算のスケジューリングは, 図 11 の “Access Top” の情報を参照して行う. スケジューリング制約が緩和される条件を表 3 に示す. プログラムカウンタ更新演算のスケジューリングは, 図 11 の “Access Bottom” の情報を参照して行う. アクセス情報によるスケジューリング制約の緩和を表 4 に示す. ‘制約’ 列は, ‘条件’ 列の条件を満たす場合に, プログラムカウンタ更新演算が最後の ‘演算’ 列の演算の何サイクル前以降に実行できるかを表す.

表 3　遷移前 DFG のアクセス情報によるスケジューリング制約の緩和

演算	条件	制約
ロード演算 ストア演算	直前の 1 命令が非メモリアクセス	2 サイクル目以降
	直前の 2 命令が非メモリアクセス	1 サイクル目以降
R 演算	直前の 1 命令がメモリロードかつ R 演算が直前の 1 命令の結果を未使用	1 サイクル目以降
W 演算	直前の 1 命令が非 RF 書き込み	3 サイクル目以降
	直前の 2 命令が非 RF 書き込み	2 サイクル目以降
	直前の 3 命令が非 RF 書き込み	1 サイクル目以降

5.　実装と実験

本稿で提案するアクセラレータ合成手法および最適化手法を

表 4　遷移先 DFG のアクセス情報によるスケジューリング制約の緩和

演算	条件	PW 演算の制約
ロード演算	直後の 1 命令が非メモリアクセス	4 サイクル前以降
ストア演算	直後の 2 命令が非メモリアクセス	5 サイクル前以降
	直後の 3 命令が非メモリアクセス	6 サイクル前以降
R 演算	直後の 1 命令が非 RF 読み出し	1 サイクル前以降
	直後の 2 命令が非 RF 読み出し	2 サイクル前以降
	直後の 3 命令が非 RF 読み出し	3 サイクル前以降
W 演算	直後の 1 命令が書き込み結果を未使用	3 サイクル前以降
	直後の 2 命令が書き込み結果を未使用	4 サイクル前以降
	直後の 1 命令が非 RF 書き込み	5 サイクル前以降

ACAP [10] に実装した．いくつかのプログラムの一部をアクセラレータに合成した結果を表 5 に示す．プログラムは AES と, CHStone [11] の SHA, Blowfish の 3 つを用い, それぞれ呼出し回数の多い 1 関数全体をアクセラレータ化した．CPU は MIPS R3000 互換プロセッサを使用した．アクセラレータの合成では, ALU の資源制約は 9 個に設定し, バインディングには部分共有手法 [12] を適用した．ALU の遅延は 7 ns で, 1 サイクルあたり 25 ns になるようチェイニングを適用した．「SW 単体」は CPU のみ, 「SW+HW (最適化なし)」は CPU と最適化手法を適用せずに生成したアクセラレータ, 「SW+HW (最適化あり)」は 最適化手法を適用した上で生成したアクセラレータでの結果を表す．Slice は CPU とアクセラレータを合わせた回路のスライス数である．cycle はプログラム全体の実行に掛かったサイクル数, delay は遅延時間 (単位は ns) を表す．

結果として, 最適化手法を適用したアクセラレータの場合, CPU の約 0.5 倍から約 1.4 倍のアクセラレータの追加により, プログラム全体の実行速度を約 1.5 倍から 3 倍に高速化することができた．また, 表 5 の sha のように, 最適化なしのアクセラレータではソフトウェア単体より実行サイクル数が増大していた場合でも, 最適化手法により, 実行サイクル数を大きく減らせた．

しかし, 遅延はチェイニングの設定に反し, 25 ns を大きく超える結果となった．この遅延は, 実際には使用されない経路 (フォルスパス) によるものであり, 実際の遅延は 25 ns 程度であると考えられる．

6. むすび

本稿では, 機械語プログラムの指定区間を CPU 密結合型アクセラレータに合成する手法を提案した．本手法を高位合成システムに実装し, 生成したアクセラレータで実験した結果, CPU の最大約 1.4 倍の面積のアクセラレータの追加により, プログラム全体の実行速度を最大 3 倍に高速化できた．

今後の課題としては, バインディングアルゴリズムの改良によるフォルスパスの解消等が挙げられる．

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に感謝いたします．また, 本研究に関してご協力, ご討議頂いた山下真司氏, 伊藤直也氏をはじめ, 関西学院大学石浦研究室の諸氏に感謝いたします．

文　献

[1] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, and Steve Y-L Lin: *High-Level Synthesis: Introduction to Chip and System Design,* Kluwer Academic Publishers (1992).

[2] Seng Lin Shee, Sri Parameswaran, and Newton Cheung: "Novel architecture for loop acceleration: A case study," in *Proc. Workshop on Hardware/Software Codesign+International Symposium on System Synthesis '05*, pp. 297–302 (Sept. 2005).

[3] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Anderson, Stephen Brown, and Tomasz Czajkowsk: "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proc 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 33–36 (Feb. 2011).

[4] Greg Stitt and Frank Vahid: "Binary synthesis," *ACM Trans. on Design Automation of Electronic Systems*, vol. 12, no. 3, article 34 (Aug. 2007).

[5] 瀬戸謙修, 藤田昌宏: "高位合成技術を利用したカスタム命令自動生成手法," 情報処理学会 DA シンポジウム 2006, pp. 49–54 (July 2002).

[6] Nagaraju Pothineni, Philip Brisk, Paolo Ienne, Anshul Kumar, and Kolin Paul: "A High-level synthesis flow for custom instruction set extensions for application-specific processors," in *Proc. Asia and South Pacific Design Automation Conference 2010*, pp. 707–712 (Jan. 2010).

[7] 戸田勇希, 石浦菜岐佐, 神原弘之, 冨山宏之: "CPU と密に結合したコプロセッサに基づくハードウェア/ソフトウェア協調設計," 情報処理学会研究報告, 2010-ARC-187-16/2010-EMB-15-16 (Jan. 2010).

[8] 佐竹俊亮, 石浦菜岐佐, 田村真平, 神原弘之, 冨山宏之: "機械語の複数部分を高速化する CPU 密結合型ハードウェアアクセラレータ," 電子情報通信学会技術研究報告, VLD2012–119 (Jan. 2013).

[9] 伊藤直也, 石浦菜岐佐, 冨山宏之, 神原弘之: "CPU とハードウェアアクセラレータの実行切替えの高速化," 電子情報通信学会ソサイエティ大会, A-3-8 (Sept. 2013).

[10] 池上達也, 石浦菜岐佐: "MIPS アセンブリを中間表現とする高位合成," 情報処理学会関西支部大会 2008, A-03 (Oct. 2008).

[11] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii: "CHStone: A benchmark program suite for practical C-based high-level synthesis," in *IEEE International Symposium on Circuits and Systems, 2008*, pp. 1192–1195 (May 2008).

[12] 田村真平, 丸谷亮祐, 石浦菜岐佐: "高位合成のバインディングにおける演算器の部分共有," 電子情報通信学会ソサイエティ大会, A-3-6 (Sept. 2011).

表 5　アクセラレータの性能評価

Prog	SW 単体			SW+HW (最適化なし)			SW+HW (最適化あり)		
	Slice	cycle	delay	Slice	cycle	delay	Slice	cycle	delay
AES	3221 (1.00)	47953 (1.00)	25.957	4922 (1.53)	44453 (0.93)	37.612	4643 (1.44)	32353 (0.67)	28.627
SHA	3221 (1.00)	746649 (1.00)	25.957	7351 (2.28)	826062 (1.11)	49.274	7642 (2.37)	250125 (0.33)	42.909
Blowfish	3221 (1.00)	761878 (1.00)	25.957	7198 (2.23)	385987 (0.51)	37.785	7310 (2.27)	375448 (0.49)	62.253