Vol.2013-ARC-207 No.33
Vol.2013-HPC-142 No.33
2013/12/17

ストリーム解析処理のベンチマーク構築に向けた タスクグラフ自動生成ツール

秋岡明香[†1]

大規模データを多角的かつ横断的に解析して新しい知見を得るビッグデータ解析が注目を集めている．ビッグデータ解析の中でも，時系列に沿ってデータが到着するデータストリームを解析する大規模ストリーム解析処理は，そのデータアクセスパターンや条件分岐の多さ，タスク並列度，コスト見積り方法などに特徴がある．これにより，従来ハイパフォーマンスコンピューティング分野で注目してきたデータインテンシブアプリケーションと同様の手法では，効果的な最適化を見込むことが難しい．こうした背景を踏まえ，筆者らは大規模ストリーム解析処理に特化したベンチマークセットの構築を進めているが，効率良く既存の大規模ストリーム解析処理の挙動解析を行なうためのツールが必要となった．そこで，解析対象となる大規模ストリーム解析処理のソースコードを入力とし，適切な粒度のノードにより構成される制御依存グラフ，および各ノードの計算コスト見積りを得るツールを生成した．本稿では，このツールを紹介し，大規模ストリーム解析処理における代表的な手法に適用した例を通して，こうしたツールの有用性と今後の方向性について検討する．

A Task Graph Generation Tool for Benchmarking with a focus on Stream Mining Applications

SAYAKA AKIOKA[†1]

Big data analysis, which is supposed to analyze gigantic data across possible combinations of parameters from a broad set of perspectives, is expected to derive new findings, and extend peoples' knowledge. A large-scale stream mining application is one kind of big data applications, and very characteristic for its data access patterns, multitude of conditional branches, level of parallelism, and indeterminate calculation cost. These distinctions of a large-scale stream mining applications differentiate the strategy of speed-ups, and scale-outs from the conventional strategies built for data intensive applications in high performance computing community. This situation leads a requirement for a benchmarking set with an explicit focus on stream mining applications. Toward an establishment of such a benchmark set, every implementation, or algorithm of stream mining should be analyzed, and characterized in the most efficient way. This paper, therefore, introduce a task graph generation tool, which accepts source codes of stream mining applications, generates control flow graphs with proper granularity of nodes, and estimates computational cost of each node. We also discuss usability, and possible extensions of the tool with an example.

1. はじめに

ビッグデータ解析，特に時系列に沿って高速で到着する大量データの解析処理（大規模データストリーム解析）への需要が高まっている．現状では，解析処理能力や計算規模の問題から，統計的な解析や限られたパラメータセットについてのデータマイニングが限界である．ビッグデータ解析が目指す，世の中の全てのデータに対するリアルタイムな網羅的かつ横断的解析は，現状では非現実的である．

大量のデータを処理するアプリケーションの並列分散化手法や高速化手法は，並列分散処理の分野で長く研究されてきた．しかし，大規模データストリーム解析処理は，従来の並列分散処理アプリケーションとは挙動が全く異なることが知られており[1]，大規模データストリーム解析処理のモデルは明示されていないため，高速化やスケールアウトの指針は自明でない．

数値計算など，並列分散処理分野で飛躍的な速度向上を継続しているアプリケーションは，LINPACK ベンチマーク[2]などの世界標準ベンチマークがその特徴を的確にとらえている．したがって，これらのベンチマークを高速実行する計算環境は，数値計算などのアプリケーションを高速実行可能であると見なすことができ，計算環境の研究開発の指標が具体的かつ明確である．結果として，コミュニティ全体でこれらベンチマークの高速化に向けた研究を進めることができた．

一方で，大規模データストリーム解析処理では，その挙動の特異さが指摘されているにも関わらず，その特徴の明示的なモデルや，特性を反映したベンチマークセットなどは存在していない．また，FPGA の研究においては，データフローモデルに基づく FPGA 設計の提案が数多くあるが[3-10]，いずれも個別のアプリケーションの高速化に留まっており，大規模データストリーム解析処理という大きな括りでの最適化を目指した研究はない．こうした状況では大規模ストリーム解析処理を高速化・スケールアウトするための方針は明確ではなく，大規模データストリーム解析処理に特化したベンチマークセットの構築が必要である．このようなベンチマークセットを構築するためには，既存の大規模データストリーム解析手法を効率良く解析するた

†1 明治大学
Meiji University

めのツールが不可欠である．

そこで本稿では，大規模データストリーム解析の特徴を踏まえ，パイプラインによる並列実行[11]を前提とし，ソースコードを入力として，適切な粒度のノードにより構成される制御依存グラフ，および各ノードの計算コスト見積りを得るツールを紹介する．また，このツールを大規模ストリーム解析処理における代表的な手法に適用した例を通して，こうしたツールの有用性と今後の方向性について検討する．なお，本稿の構成は以下の通りである．第 2 章では，本稿が対象とする大規模データストリーム解析で頻繁に用いられるストリームマイニングアルゴリズム一般に共通する特徴についてまとめる．第 3 章では，本稿で紹介するツールの概要を述べる．第 4 章では，このツールの利用例を紹介し，ツール利用の利点と今後の課題を議論し，第 5 章でまとめる．

2. ストリームマイニングアルゴリズム

ストリームマイニングアルゴリズムとは，入力データを取りこぼすことなく，データストリームをリアルタイムに解析することを目的とした解析手法を指す．多様なストリームマイニングアルゴリズムの提案があるが，ストリーム解析全般に共通な依存関係や構造がある．この共通の特徴を図 1 にモデルとして示す[12].

ストリームマイニングアルゴリズムは，ストリーム処理部(Stream Processing Part)とクエリ処理部(Query Processing Part)に分けることができる．ストリーム処理部は，ストリーム処理モジュール(Stream Processing Module)，スケッチ(Sketch)，解析モジュール(Analysis Module)で構成する．ここでスケッチとは，キャッシュの役割を果たすメモリ領域である．クエリ処理部は，ユーザからのクエリを処理するモジュール(Query Processing Module)からなる．データ到着からの一連の処理の流れは，次の 4 ステップである．

1) 処理対象の単位データをストリーム処理部で読み込み，形態素解析や語句カウント等の簡易的な前処理を行なう．
2) 前処理の結果でスケッチの内容を更新する．
3) 解析モジュールがスケッチの内容を読み込み，ストリーム処理モジュールが行なった解析の続き（名詞句のみの抽出など）を行なう．
4) 解析モジュールの結果をクエリ処理モジュールが受け取り，より詳細な解析を行なう．

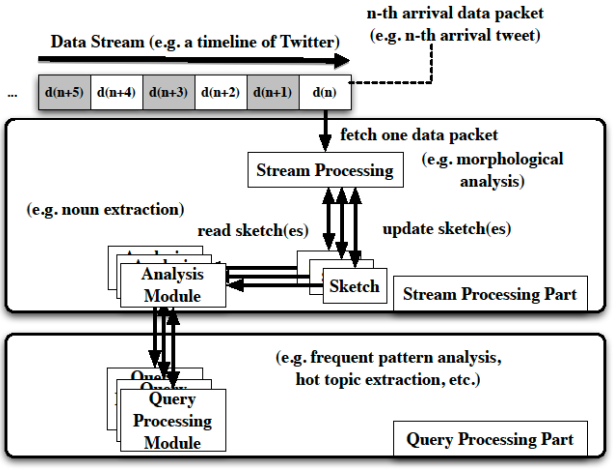

図 1 ストリームマイニングアルゴリズムのモデル
Figure 1 A model of stream mining algorithms.

ここで，スケッチは高速で到着する入力データの取りこぼしを防ぐためにストリーム処理部の負荷を軽くする役割を果たしている．つまり，スケッチの内容は頻繁かつインクリメンタルに更新される．一方で，クエリ処理部が担当する処理は，比較的実行に時間がかかり，かつ処理対象となるデータも時間軸上に広く分布する傾向がある．つまり，ストリーム解析処理全体のうちで，データ入力の取りこぼしを防ぎ，実行時の時間制約条件が厳しいのはストリーム解析部であり，クエリ解析部のリアルタイム実行の必然性は低い．したがって，大規模データ解析処理にストリームマイニングアルゴリズムを用いる場合，高速化およびスケールアウトの鍵となるのは，ストリーム解析部である．

さらに，図 1 から，ストリーム解析処理のデータアクセスパターンの特徴が明確になり，従来の HPC 分野で研究されてきたデータインテンシブアプリケーションとは，データアクセスパターンが大きく異なることがわかる．つまり，従来のデータインテンシブアプリケーションでは，同じデ

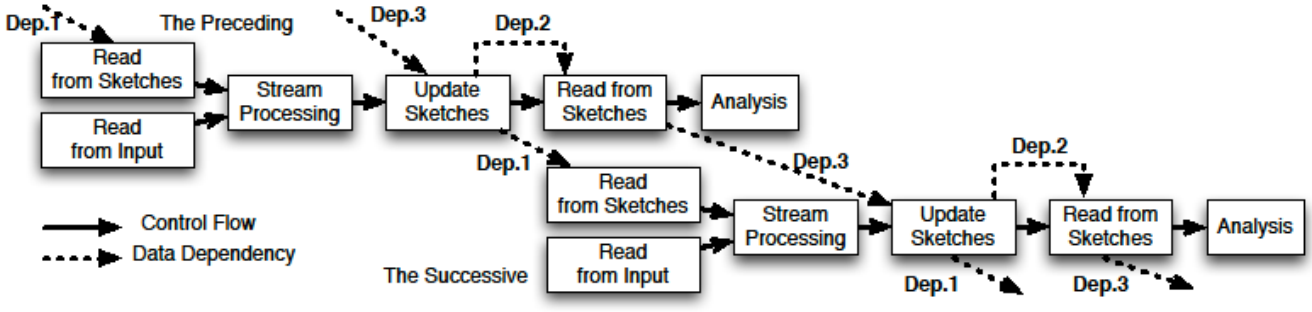

図 2 連続する 2 つのストリーム処理部プロセス間におけるデータ依存と制御依存
Figure 2 Data dependencies and control dependencies of the two processes of the stream processing parts.

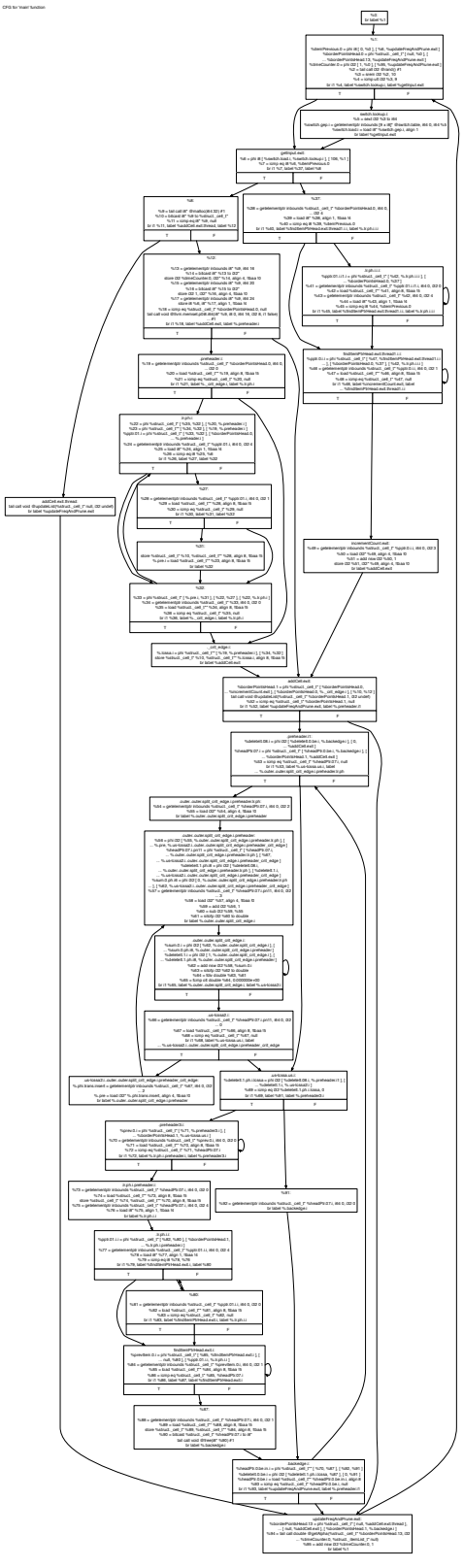

図 3 LLVM コンパイラによる Top-K アルゴリズム (Min-Summary) の制御依存グラフ
Figure 3 A control flow graph for Top-K algorithm (Min-Summary) by LLVM compiler.

ータを何度も使用する write-once-read-many 型のデータアクセスパターンであった[1]．したがってアプリケーション

for all input data items **do**
　(1) fetch one input data v
　for all distinct items appeared **do**
　　(2) create or update a border point for v
　　(3) update summary
　　(4) update frequency
　　(5) delete obsolete border points
　end for
　(6) update pruning threshold
end for

図 4 Min summary アルゴリズムの疑似コード
Figure 4 A pseudo-code of min summary algorithm.

高速化のためには，計算ノードにできるだけ近いところに処理対象のデータを配置し，データアクセススピードを向上させることが重要であった．一方で，ストリーム解析処理では, 一度処理を行なったデータは 2 度と読み込まない，read-once-write-once 型のデータアクセスである．したがって，ストリームマイニングアルゴリズムを対象とした場合には，従来のスケジューリング手法や高速化手法を単純に適用しても，充分な高速化を実現することは難しい．図 2 に, 前述の 1)から 3)の処理を 1 プロセスと見なした場合に，連続する 2 つのプロセスに注目した場合の，プロセス間およびプロセス内のデータ依存関係と処理依存関係を示す．

図 2 において，上段のフローが先行プロセスを，下段のフローが後続プロセスを示す．各プロセスは，入力データの読み込み (Read from Input)，スケッチの読み込み (Read from Sketches)，ストリーム処理 (Stream Processing)，スケッチの更新 (Update Sketches)，スケッチの読み込み (Read from Sketches)，解析 (Analysis)，の 6 ステージで構成される．矢印は処理依存関係を示し，破線矢印はデータ依存関係を示す．

3. タスクグラフ自動生成ツール

3.1 通常のコンパイラを利用した場合の問題点

タスクグラフを生成する上で，ソースコード中に存在するデータ依存や処理依存を洗い出すことは不可欠である．また，既存のコンパイラには制御依存グラフを出力するものもある．例として，LLVM コンパイラ[13]で Top-K アルゴリズム（Min-Summary アルゴリズム）を実装したソースコードを解析した結果得られる制御依存グラフを図 3 に示す．また，対応する疑似コードを図 4 に示す．

しかし，コンパイラが出力する制御依存グラフは図 3 に示すような細粒度なノードにより構成するグラフであり，本稿が目指すベンチマークセット構築という目的においては不適当である．さらには，ストリームマイニングアルゴリズムの特徴として，データ入力のコストに対して計算コ

ストが比較的小さいという特徴がある．したがって，コンパイラが生成する細粒度な制御依存グラフに加え，実行時コスト見積り等の情報を組み合わせて，適切な粒度のノードで構成する制御依存グラフを再構築する必要がある．

3.2 実行時コストの見積り

ストリームマイニングアルゴリズムにおいて，実行時のコスト見積りは非常に難しい．たとえば，Top-K アルゴリズムのひとつである Min-Summary アルゴリズムでは，実行時にユーザが指定する解析結果の精度や再現率，入力データに含まれるアイテムの種類数などによって，計算途中で保持するデータ数や履歴数が異なる．計算途中ではこれらのデータを走査する処理が何度も現れるが，こうした理由から，各処理の計算時間を静的に見積もることは難しい．また，計算時間の振れ幅も大きいため，多数の投機的実行を行なった結果の平均値などで前もって概算しておくことも，好ましくない．

そこで本ツールでは，LLVM コンパイラのフロントエンドを拡張し，ソースコード中のすべての関数について，関数単位で実行時間を測定するためのタイマーコードを挿入する．具体的には，FunctionPass を継承した独自 Pass を用意し，runOnFunction()で対象関数の先頭と末尾にタイマー関数を挿入すると同時に，関数の末尾に計算コストを算出する命令を追加する．FunctionPass を継承した独自 Pass を使用することで，LLVM でコンパイルしたコードは，すべて関数ごとに実行時間計測が可能になるのみならず，制御依存グラフを変更することもない．また，フロントエンドの拡張であるため，時間計測に関連するコードの挿入は，LLVM の中間言語コードについて行なう．その結果，入力ソースコードは LLVM でコンパイル可能な全ての言語で書かれたソースコードとすることができ，高い汎用性を得られるという利点もある．このようにしてタイマーコードを含んだコードを複数回実行し，各関数の平均実行時間を求め，次節で述べるノードの再構成を行なう際に利用する．

なお，関数単位で実行時間を求めてノード再構成の対象とすることは，大きく 2 つの利点がある．ひとつは，コンパイラが提供する細粒度の制御依存グラフを再構成する上で，関数単位の計算コストは重要な目安となることである．もうひとつは，関数に引数として渡されるデータはその関数中で利用される確率が高い．本稿では，対象コードをパイプラインで実行することを前提としており，依存関係があるデータはパイプライン上で受け渡される．この時，ひとつの関数を制御依存グラフ上の複数のノードに分割することは，関数に引数として渡されるデータや内部変数など，より多くの依存関係があるデータを複数ノードで使用する結果となる可能性が高く，パイプライン上を流れるデータ量が増える可能性が高くなる．これは，本稿で対象とするような，データ送受信コストと比較して計算コストが比較的小さい計算モデルには不向きである．

3.3 ノードの再構成

前節で述べた関数ごとの実行時間見積りと LLVM による制御依存グラフにより，関数単位での実行時間見積りを含んだコールグラフを得ることができる．ここで，複数の関数から呼ばれる関数や，ひとつの関数から複数回呼ばれる関数については，手法の基幹部分に関わる関数ではなくツール的に使われる関数である可能性が高い．したがって，ある関数（ツール呼び出し元関数）がこうしたツール的な関数を呼び出している場合，再構成後のノードの単位は，ツール呼び出し元関数以上とする．

以上の方針により再構成後のノード候補となる関数群を求め，それぞれの関数について実行時間見積りを得る．その後，ツール利用者が定めるパイプライン段数nに応じて，さらなる関数群の集約を行なう．つまり，ノード候補となる関数群がm個あり，$m > n$である場合には，以下の手順によりノードの集約を行なう（$m \leq n$の場合にはノードの集約は行なわない）．

1) 関数f_i（$1 \leq i \leq m$）の実行時間見積りをE_iとする．連続して呼び出される 2 つの関数の実行時間見積りの和を$E_{i,i+1}$とする．
$$E_{i,i+1} = E_i + E_{i+1}\ (1 \leq i \leq m-1)$$
2) $\min_{1 \leq i \leq m-1} E_{i,i+1}$を満たす$i$を求め，$f_i$と$f_{i+1}$を集約して新たなノード候補とする．
3) 以上の手順で得られた$m-1$個のノード候補群を$f_1^1, \ldots, f_{m-1}^1$とし，これらのノード候補群に上記 1)および 2)の手順を適用する．
4) $m = n$となるまで繰り返す．

上記 1)～4)の手順により，隣接する関数でかつ実行時間見積りの合計が小さい関数の組が優先的に集約されることが期待できると同時に，この手順を繰り返すことで，集約後のノード群の実行時間見積りの分散を小さくすることが期待できる．したがって，再構成した後の制御依存グラフは，パイプライン実行した場合に各段の実行時間の分散が小さくなることが期待でき，効果的に実行できる．

4. 利用例とツールの検討

図 4 に疑似コードを示した Top-K アルゴリズム（Min-Summary アルゴリズム）について，第 3 章で述べたツールを適用した結果を図 5 に示す．各ノードの番号は図 4 の疑似コードに振られた番号と対応し，図中の「1+2」と記されたノードは，疑似コードの(1)および(2)に対応するノード 1 およびノード 2 が集約された新たなノードであることを示す．

なお，同じソースコードについて main 関数からの呼び出される関数単位でノードを構成し，タイマー関数を用いた実行時間見積りを行なった上で人手により書いた制御依存グラフが図 6 となる．このグラフはノードの集約等の再構

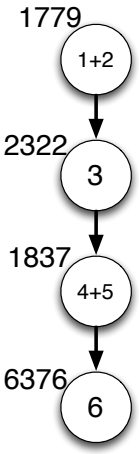

図 5　Top-K アルゴリズムの制御依存グラフ
（ツールによる描画，パイプライン 4 段を指定.）
Figure 5 A control flow graph of Top-K algorithm
by the tool (case of 4 stages).

成は行なっておらず，各ノードの番号は図 4 の疑似コードに振られた番号と対応する．また，出現アイテム数などで並列化可能な部分については，並列実行が可能であることがわかるようにノードを同じレベルで複数並べて表現している．

図 5 および図 6 を比較することで，次のことが分かる．

1) 元々の制御依存グラフには，他のノードと比較して実行時間見積りが極端に小さいノード 1 やノード 6 が独立して存在しているが，ツールでのノードの再構成により，これらのノードは近隣のノードに集約され，極端に実行時間見積りが小さいノードがなくなった．
2) 手動による制御依存グラフでは，並列実行可能な箇所が明確になっているが，ツールによる制御依存グラフでは，こうした並列性が表現されていない．これは，LLVM でのコンパイル時にシリアルコードとしてコンパイルしており，並列性の抽出を行なっていないことが主な原因である．
3) ノード 6 は実行時間見積りが極端に大きく，可能であれば実行性能を損なわない範囲内で分割して複数ノードに再構成することが望ましい．しかし，関数単位での実行時間見積りを行なっていること，第 3 章で述べたように，本稿での手法はノードを集約する方向にあることなどから，今回はノードの分割は一切行なっていない．依存関係を増やしたり，パイプライン上で受け渡されるデータを極端に増やしたりすることなく，このような大きなノードを分割するための手法について，検討が必要である．
4) 実行時間見積りについては，入力データの特性やユーザによる手法の精度指定によって流動的であることは変わらない. 実運用と同じ特性の入力データを用いて，精度指定などのパラメータ指定も実運用と同様に行なうことができれば，現在の実行時間見積り方法でも問題ないが，現実的ではない．今後，抜本的な改善が必要である．

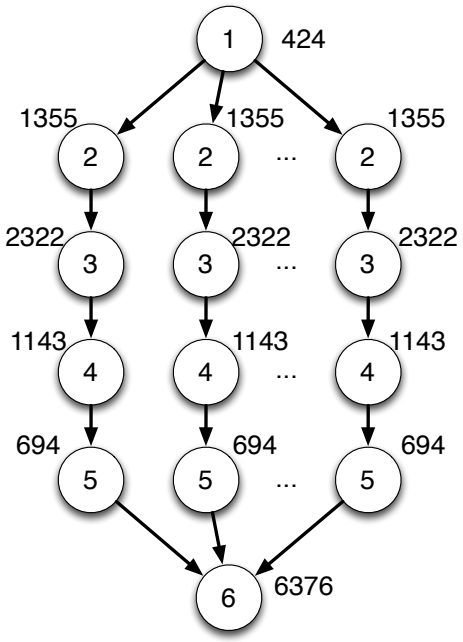

図 6　Top-K アルゴリズムの制御依存グラフ（手動）
Figure 6 A control flow graph of Top-K algorithm
by hand.

5. まとめ

本稿では, 大規模データストリーム解析の特徴を踏まえ，パイプラインによる並列実行を前提とし，ソースコードを入力として，適切な粒度のノードにより構成される制御依存グラフ，および各ノードの計算コスト見積りを得るツールを紹介した．ツールは主に既存のコンパイラである LLVM を拡張することで実現しており，ソースコードから実行時間見積り付きの制御依存グラフ生成までを半自動的に行なうことができることから，多くのデータストリーム解析実装を効率的に解析する基盤として利用できる．ただし，並列性抽出や実行時間見積りの精度などについて改良の余地が多くあることから，今後はこれらの点を中心に改良を重ね，より実用的なツールへ発展させる予定である．

参考文献

1) I. Raicu, I. T. Foster, Y. Zhao, P. Little, C. M. Moretti, A. Chaudhary, and D. Thain, “The quest for scalable support of data intensive workloads in distributed systems,” Proc. 18th ACM International Symposium on High Performance Distributed Computing (HPDC’09), pp. 207-216, Munich, Germany, June 2009.
2) J. Dongarra, “Performance of Various Computers Using Standard Linear Equations Software,” University of Tennessee Computer Science Technical Report CS-89-85, 2013.

3) J. Fowers, and G. Sitt, “Dynafuse: Dynamic Dependence Analysis for FPGA Pipeline Fusion and Locality Optimizations,” Proc. 21st ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA’13), pp.201-210, Monterey, USA, February 2013.

4) W. Sun, M. J. Wirthlin, and S. Neuendorffer, “Combining Module Selection and Resource Sharing for Efficient FPGA Pipeline Synthesis,” Proc. 14th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA’06), pp. 179-188, Monterey, USA, Februrary 2006.

5) A. Tumeo, M. Branca, L. Camerini, C. Pilato, P. L. Lanzi, F. Ferrandi, and D. Sciuto, “Mapping Pipelined Applications onto Heterogeneous Embedde Systems: a Bayesian Optimization Algorithm Based Approach,” Proc. Fifth IEEE/ACM/IFIP International Conference on Hardware-Software Codesign and SystemSynthesis (CODES+ISSS’09), pp.443-452, Grenoble, France, October, 2009.

6) W. Wang, B. Duan, W. Tang, C. Zhang, G. Tan, P. Zhang, and N. Sun, “A Coarse-grained Stream Architecture for Cryo-electron Microscopy Images 3D Reconstruction,” Proc. 20th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA’12), pp.143-152, Monterey, USA, February 2012.

7) J. T. Zhai, H. Nikolov, and T. Stefanov, “Modeling Adaptive Streaming Applications with Parameterized Polyhedral Process Networks,” Proc. 48th ACM/EDAC/IEEE Design Automation Conference (DAC’11), pp.116-121, San Diego, USA, June 2011.

8) J. Zhu, I. Sander, and A. Jantsch, “Pareto Efficient Design for Reconfigurable Streaming Applications on CPU/FPGAs,” Proc. EDAA/ACM Design, Automation, and Test in Europe (DATE10), pp.1035-1040, Dresden, Germany, March 2010.

9) M. Hashemi, M. H. Foroozannejad, and C. Etzel, “FORMLESS: Scalable Utilization of Embedded Manycores in Streaming Applications,” Proc. 13th ACM SIGPLAN/SIGBED International Conferences on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES2012), pp.71-78, Beijing, China, June 2012.

10) W. Najjar, and J. Villarreal, “FPGA Code Accelerators – The Compiler Perspective,” Proc. 50th ACM/EDAC/IEEE Design Automation Conference (DAC’13), pp.141-146, Austin, USA, May, 2013.

11) S. Akioka, “Task Graphs of Stream Mining Algorithms,” Proc. First International Workshop on Big Dynamic Distributed Data (BD3 2013), pp. 55-60, Trento, Italy, August 2013.

12) S. Akioka, Y. Muraoka, and H. Yamana, “Data Access Pattern Analysis on Stream Mining Algorithms for Cloud Computation”, Proc. the 2011 International Conference on Parallel and Distributed Processing (PDPTA2011), pp.36-42 (2011).

13) The LLVM Compiler Infrastructure Project, http://llvm.org/.

14) P. Domingos, and G. Hulten, “Mining High-Speed Data Streams,” Proc. The 6th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD’00), pp.71-80 (2000).