ＲＴＬ－Ｔｏｋｉｏに基づくパイプライン化支援

3S-10

中村　宏・河野　真治・中井　正弥・*藤田　昌宏・田中　英彦
東京大学工学部・*富士通研究所

１．はじめに

我々は、従来より時相論理型言語Tokio、RTL-Tokioを中心とした論理設計支援を目指している[1]。ここでTokioは時相論理(Interval Temporal Logic)に基づくハードウェア記述言語であり、RTL-TokioはTokioに制限を加えたレジスタトランスファレベル動作記述言語である。Tokio及びRTL-Tokioは時相論理に基づくため、並列性・順序性といった時間に関する記述を厳密に行うことができる。さらにRTL-Tokioは動作解析ツールによって構造記述との無矛盾性のチェックができる[2]。これらの特徴を用いた、RTL-Tokioに基づくパイプライン化支援を現在考えており、ここではその手法について述べる。

２．RTL-Tokioと動作記述

２．１．TokioとRTL-Tokio

Tokio[3]はInterval Temporal Logic[4]に基礎をおく離散時間上に定義された論理型言語である。Tokio及びRTL-Tokio[5]の詳細は紙面の都合で省略するが、直観的にはPrologに時相演算子を加えたものと考えられる。

・（時相演算子を含まない）：これは現在のインターバルで実行されることを表す。
・chop(&&)：これは任意のインターバルを前半と後半の２つのインターバルの区切る演算子である。例えば”pred1 && pred2”は、現在のインターバルを２つに分け、前半でpred1後半でpred2を実行することを表す。
・always(#)：これは現在のインターバルと終了する時刻が同じである全てのサブインターバルで実行することを表す。　RTL-TokioはTokioに対し
　・バックトラックをしない
　・変数がレジスタ、メモリ等のデータに対応する
　・離散時間上の最小時間をレジスタ転送に対応させる

という制限を加えた、レジスタトランスファレベル動作記述言語である。

２．２．RTL-Tokioによる動作記述

RTL-TokioあるいはTokioでパイプライン化された動作記述を行う場合、大きく分けて、always(#)演算子を用いてそれぞれのパイプが毎時刻起動されることを記述する方法とchop(&&)を用いて、パイプライン化された動作記述を一つのオートマトンとして記述する方法がある。

簡単な例として、長さ１のステージが３つある動作記述、及びその動作を毎時刻起動するというパイプライン動作記述を下に示す。

① sequentialな記述

```
main :- stage1 && stage2 && stage3.
```

② #(always)を用いたパイプライン記述

```
main :- #(stage1 && stage2 && stage3 && true).
```

③ &&(chop)を用いたパイプライン記述

```
main :- stage1 &&
        ( stage2 && stage3 && true ), main.
```

ここで②と③の記述のtrueは長さ０以上で必ず成功する述語であり、インターバルの長さを合わせるために用いられている。#(always)を用いる記述では、各ステージの長さが一定でない場合（例えば条件分岐等）各ステージ毎にフラグを設ける必要があり、一方、&&(chop)を用いる記述では、条件分岐が存在しても必ずしもフラグを設ける必要はないがその際状態数が増加する事になる。

３．パイプライン化手法

３．１．パイプライン化

パイプラインには狭義のパイプラインと広義のパイプラインがある[6]。狭義のパイプラインは例えばアレイコンピュータにおけるパイプラインであり、スループットがパイプの長さあるいはアレイの長さに依存しないが、条件分岐等があると実現は難しくなる。一方、広義のパイプラインは、前の入力に対する出力が得られる前に次の入力が入るという意味のパイプラインで、スループットはパイプの長さに依存する。狭義のパイプラインは広義のパイプラインの特別な場合と考えられる。

sequentialな動作記述をパイプライン化する場合、costとperformance という、相反する二つの制約条件下でのtrade-off pointを求める必要がある。狭義のパイプライン化を対象にした[7]などの研究では、条件分岐の無いデータフローグラフを入力とし、評価関数を用意してcost/performanceの最適解あるいは最適解に近い解を見つけるアルゴリズムが述べられている。狭義のパイプラインでは、例えばＮ倍のデータパスを用意すればＮ倍の性能が得られる場合もあり、costとperformance のtrade-off pointをなんらかの評価関数をもとに決定するのが重要な処理になる。それに対し広義のパイプラインを考えるときは、データ間の依存関係を見つけ、パイプライン化できるステージを見つけることが主な処理になる。

３．２．パイプライン化の処理の流れ

現在、広義のパイプライン化を目指したシステムを構

Assistance of pipelining based on RTL-Tokio
Hiroshi NAKAMURA, Shinji KONO, Masaya NAKAI,
*Masahiro FUJITA, Hidehiko TANAKA
University of Tokyo, *FUJITSU LABORATORIES LTD.

築中であり、図１はその処理の流れを表す。

入力はRTL-Tokioによるレジスタトランスファレベルの動作記述及びデータパスである。パイプライン化ツールはデータ依存関係より並列に実行できるデータ転送を見つけ、パイプライン化された動作記述（chopを用いたオートマトンとしての記述）を出力する。構造記述に対する変更はここでは行わず、動作解析ツール[2]が動作記述と構造記述の間の無矛盾性を保証する。costとperformanceのtrade-offについては、Tokioのシミュレータを用いながら設計者がinteractiveに関与し、より良い解を求めるという方針である。レジスタトランスファレベルのパイプライン化支援としては、costをそれほど正確に評価できない等の問題もあり、この方針は妥当であると考える。

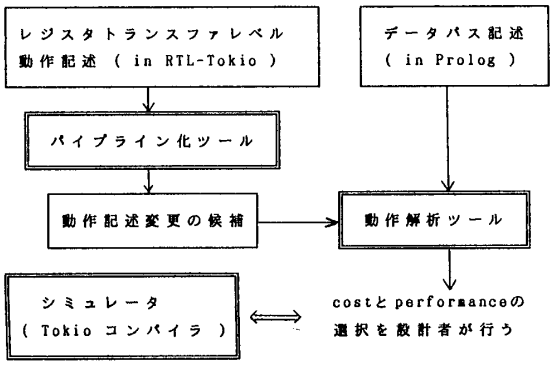
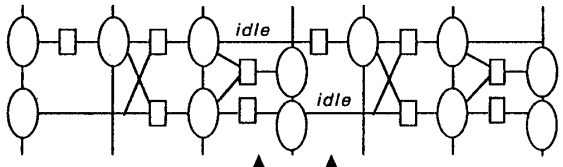


図１．パイプライン化の処理の流れ

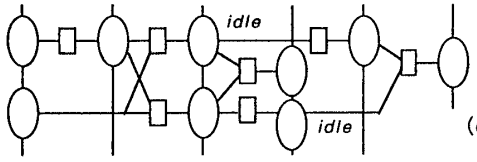
```
main :-  /* named as main-first */
   [localCond],  !,  *reg1 <= (*a >> 1)
     & &
   *a <= *reg1 + *b, *reg2 <= *reg1 - *b
     & &
   *b <= (*reg2 >> 1), *out <= *a + *reg2
   &&
     main.
main :-    /* named as main-second */
  !,  *reg1 <= (*a >> 1)
    & &
  *out <= *reg1 + *b.
```

(a)　もとの動作記述

(in case main-first follows main-first)



These stages can be executed in parallel.



(c)　(b)より作られるデータフロー

(in case main-second follows main-first)

３．３．パイプライン化ツール

パイプライン化ツールでの処理の概略を述べる。まずRTL-Tokioの記述をインターバル毎のデータ転送表に変換する。この表をもとにして、まず、各クローズ内でのidleなデータフローを見つける。あるノードに対する全ての入力についてidleなフローがあればそのノードを前の詰めることができ、さらに、二つ以上のクローズにまたがるidleなフローを見つける。条件分岐がある場合、全ての分岐に対してidleなフローがあれば、パイプライン化できる（図２）。

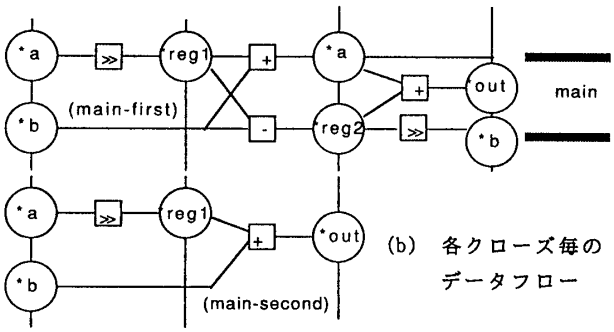
狭義のパイプラインについてもこのパイプライン化ツールで支援できるが、その場合costとperformanceのtrade-off問題に対する支援を強化する必要がある。

４．おわりに

RTL-Tokioに基づくパイプライン化支援について述べた。RTL-Tokioは、順序性・並列性といった時間に関する記述が厳密かつ容易にでき、しかも構造記述との対応が取れることを用いている。今後は、本稿では述べなかった、パイプラインにおけるクロックの最適化、制御系の回路のコスト等について考察を加えていきたい。

参考文献

[1]情報処理学会研究会報告,87-DA-40-18
[2]中井他：第38回情処全大発表予定
[3]M.Fujita et al.,IEE Proc.Pt.E.,pp283-294,1986
[4]B.Moszkowski:IFIP 6th CHDL,1983
[5]中村他：第37回情処全大，1U-3
[6]H.V.Jagadish:IEEE Trans on Comp,C-35,No5,1986
[7]N.Park et al:23rd DA Conference,pp454-460,1986



(b)　各クローズ毎のデータフロー

```
main :-  /* named as main-first */
   [localCond],  !,  *reg1 <= (*a >> 1)
     & &
   *a <= *reg1 + *b, *reg2 <= *reg1 - *b
     & &
   (*b <= (*reg2 >> 1), *out <= *a + *reg2 && true), main.
main :-    /* named as main-second */
  !,  *reg1 <= (*a >> 1)
    & &
  *out <= *reg1 + *b.
```

(d)　得られる動作記述

図２．パイプライン化の様子