特別研究報告書

Radial Layout に基づく区間再利用の分析

指導教員　　富田 眞治 教授

京都大学工学部情報学科

杉浦 信吾

平成 17 年 2 月 10 日

Radial Layout に基づく区間再利用の分析

杉浦 信吾

内容梗概

主要な商用マイクロプロセッサは、プログラム資産を継承しつつ高速化を図るスーパスカラ方式を採用し、動作周波数を極限まで追求する高速化競争を繰り広げている。しかし、キャッシュや主記憶など、記憶装置ごとの通信速度の差は拡大しており、IPC 向上率は鈍化している。一方、次世代ハイエンドプロセッサにおいて、消費電力を抑えつつ並列度を向上する中核的技術として、ハードウェア実装によるチップマルチプロセッサ構成、プログラム実行モデルによる SpMT（投機的マルチスレッディング）という 2 種類の高速化手法が注目されている。しかし、コンパイラによる緻密な並列化や専用命令の埋め込みを前提とした高速化では、一般に普及できるプロセッサとはなりえない。普及させるためには、一般的なコンパイラが生成する平凡なロードモジュールを高速実行できるマルチスレッドモデルが必要である。

本論文で用いる区間再利用機構、および並列事前実行機構では、SPARC ABI (Application Binary Interface) の規定に基づいたプログラムを対象とし、専用命令を追加することなく高速化を実現させる。プログラムは、多入力多出力の複合命令となる関数やループといった命令区間によって構成されている。区間再利用機構とは、プログラム実行中に命令区間を動的に検出し、その実行結果を記憶させ、out-of-order に再利用することにより、実行する命令を大幅に削減させる高速化手法である。並列事前実行機構とは、区間再利用と SpMT を 1 つの実行モデルに統合した高速化手法である。並列事前実行では、区間再利用を行うプロセッサ (MSP) とは別に、MSP の実行履歴より入力値を予測し、同一命令区間に対し投機的に事前実行を行うプロセッサ (SSP) を複数個設ける。これにより、入力値が単調に変化し続ける命令区間など、1 プロセッサでは再利用が行われない命令区間においても、再利用による高速化が可能となる。

以上の通り、多数の要素により区間再利用機構は実現されている。しかし、従来の研究では、各命令区間において、最終的な統計値のみによる評価を行っており、入力列がどのように格納され、利用されているかを把握することは非常に困難であった。本論文では、再利用表を可視化し、再利用の効果をリアルタイムに表現する手法を提案する。本論文における可視化機構は、次のように実

行される。初めに、Radial Layout[1] と呼ばれる描画アルゴリズムを用いて、再利用表の木構造を可視化する。次に、枝の描画色を変化させることにより、再利用が行われた入力列の抽出を行う。色分けによって、短縮されたステップ数など、高速化の効果を示す。同時に、MSP による入力列を実線、SSP による入力列を点線により表すことで、並列事前実行の有効性について判定する。このような機構を SPARC シミュレータ上で実装し、Stanford-integer ベンチマークによる評価を行った。

評価の結果、プログラムの性質により、再利用されやすい入力列のパターンや頻度に大きな違いがあることがわかった。今までの研究の中で、再利用の効果が高いとされていたプログラムも、実際に再利用される様子は大きく異なっていた。ある命令区間では、一度の再利用により 1000 ステップ以上の高速化がなされた。別の命令区間では、一つ一つの効果は小さいものの、区間のほぼ全てにおいて再利用が行われていた。再利用がなされない命令区間についても、注目すべき発見があった。最大 100 以上にも及ぶ大域変数を参照する命令区間では、入力列の長さに、2〜4 倍になる大幅な差が現れ、再利用の効果があまり期待できないことがわかった。これは、事前実行のための予測が困難であることを示している。また、入力列の最大長が 100 以上に及ぶ命令区間では、一度に登録される入力列が 20 種類以内と少ない上に、再利用できるはずの入力列が短期間で消去され、結果的に再利用の機会を逃していた。並列事前実行による効果にも長所と短所があった。長所としては、単調変化する入力値を持つなど、1 プロセッサでは再利用されない命令区間における再利用頻度の増加が挙げられる。しかし、前述の通り、入力列の長さに、10〜36 という大幅な開きが見られる命令区間の場合、予測した入力列が 100 以上の長さにおよび、無駄に記憶領域を消費する事態も生じていた。以上のように、本研究では、再利用状況をリアルタイムに可視化することにより、従来わからなかった現象を観察することができた。

Analysis of Region Reuse Baced on Radial Layout

Shingo Sugiura

Abstract

Major commercial microprocessors that introduce superscalar procedures for the sake of inheritance of program property and compete for speedup by pursuit of clock frequency. But, there is a considerable gap of correspondence speed among storage units, such as cache and main memory, and, the progress of IPC is slowing down. On the other hand, there are two notable techniques that save power resource and improve parallelism. These are multi-processor using the hardware construction and SpMT (speculative multi-threading) using the model of program translation. However, it is not expectable to become a popular the speedup technique that has a premise of minute parallelize and special proposal instructions. For popularization, it needs multi-threading model that rapidly translates common load module made by common compilerat.

In this paper, the region reuse system and the parallel precomputation are used to improve the speed of programs, running based on SPARC ABI (Application Binary Interface) without special proposal instructions. Programs consist of nested regions such as functions and loops. Each region is regarded as a super-instruction with multiple inputs and outputs. The region reuse system is one of speedup techniques translated by detecting nested regions dynamically in legacy load modules, accumulation of the sets of inputs and outputs in the associative buffer, and the reuse by main thread out-of-order. The parallel precomputation is one of speedup techniques that unites region reuse system and SpMT as one computation model. In the process of parallel precomputation, programs are translated with a combination of the simple reuse mechanism provided by MSP (Main Stream Processor) and the precomputation mechanism provided by input estimation of SSPs (Shadow Stream Processors). The parallel precomputation makes it possible to reuse for regions that is not reused by one processor because of linearly changed arguments.

The region reuse system consists of various elements. But, in the past researches that estimated the value of speedup only by final statistics of each region, it was very difficult to understand the way to memorize and to reuse

input lines (the sets of inputs). This paper proposes the way to visualize the input lines and to express real-time effect of reuse. The major points of the visualization system are followings. First, it visualizes tree-structure of input lines with Radial Layout[1]. Second, the visualization system raises the reused input lines by coloring. In addition, this coloring expresses the effect of the region reuse, such as saved steps. Moreover, this system provides an effect of the parallel precomputation by solid lines as input lines by MSP and dotted lines express as input lines by SSPs. I developed a visualization system in the SPARC simulator and evaluated the reuse mechanism with Stanford-integer.

I found various patterns and frequency of reuse. The programs, which has been highly reused in the past researches, was varied of the following. In first region, the region reuse saved cycles more than 1,000 steps per one reuse. In second region, each effect of reuse was little, but reuse took place in the most of all. On the other hand, there were the notable discovery in non-reused regions. In the regions which referred to more than 100 of global variables, I found that there was defference of 2 to 4 times among the segments of input lines. In that case, it is helpless for reuse. In addition, it is difficult to make estimation for the precomputation. Moreover, one region had input lines those length was more than 100. This region was only able to memorize less than 20 input lines. In this region, the input lines helpful in reuse was purged in short time. This case decreases the chances of reuse. The effect of the parallel precomputation had both plus and minus points. For plus points, it increased reuse frequency of the region that was not reused in one processor because of linearly changed arguments. But, in the region that had 10 to 36 segments of input lines by MSP, the estimated input lines had more than 100 segments. These long lines excessively waste memory entry. The above real-time visualization of region reuse proves phenomena that has not discovered in past research.

Radial Layout に基づく区間再利用の分析

目次

第 1 章　はじめに　1

1.1　背景 1

1.2　論文の概要 2

第 2 章　区間再利用機構　2

2.1　並列事前実行機構 3

2.2　汎用 CAM における再利用表の構造 7

2.3　再利用オーバーヘッド機構 8

第 3 章　再利用表の可視化とユーザーインターフェース　8

3.1　Radial Layout の実現 10

3.2　再利用表の拡張 12

3.3　再利用された入力列の抽出 13

3.4　主スレッドと投機スレッドの区別 13

3.5　再利用表のエイジング 14

3.6　関数・ループ区間ごとの可視化 14

3.7　入力列・出力列ごとの参照 15

3.8　リアルタイムな可視化 16

第 4 章　計測および考察　16

第 5 章　まとめ　24

謝辞　25

参考文献　25

付録　A-1

第1章　はじめに

1.1　背景

区間再利用(以下では、単に「再利用」と記す。) とは、出現した命令区間に対し、入力および結果をメモリに保存し、再度、同一の入力により、同一区間を実行する際に、命令区間を実行することなく、保存された結果を利用することにより、実行時間の短縮を図る高速化手法である。現在では、過去の入力および結果を参照するだけでなく、入力パラメータが単調に変化する場合でも再利用が実現できるよう、複数のプロセッサを投入した**SpMT**（投機的マルチスレッディング）を組合わせた。SpMT を用いることで、再利用が行われる命令区間が増え、より効率の高い高速化を行っている。記憶構造の視点からは、入力および結果を時系列に従って並べた表構造から、共通する値をまとめ、木構造の形に格納することにより、幅の狭い汎用 CAM を用いた区間再利用の実現が可能となっている。木構造を採用することにより、記憶領域の節約ができるという利点もある。従来の研究では、最終的に得られる統計値のみによる評価を行っていたものの、入力列がどのように格納され、利用されているかを把握することは非常に困難であった。具体的には、実行終了時の命令区間ごとの高速化の効果は判別できても、実行途中での判別は不可能であった。実行中の状態が把握できれば、どのような入力値が高い頻度で再利用されるか、また、その入力値が再利用されることで、どれだけ実行時間が短縮されるのかといった情報を把握することができると考えた。そこで、本論文では、この入力列の木構造を可視化し、入力列ごとの再利用状況および高速化の効果をリアルタイムに表現する手法を提案する。本論文の構成は以下の通りとなる。第 2 章では、可視化に際して重要となる、再利用の具体的な動作の説明を行う。第 3 章では、入力列および再利用状況を可視化する方法、可視化を実現するアルゴリズム、そして、計測のために付加したユーザーインタフェースの説明を行う。第 4 章では、Stanford ベンチマークにおいて、入力列の記憶および再利用の様子を可視化した。結果を示し、再利用が行われやすい命令区間について分析を行う。

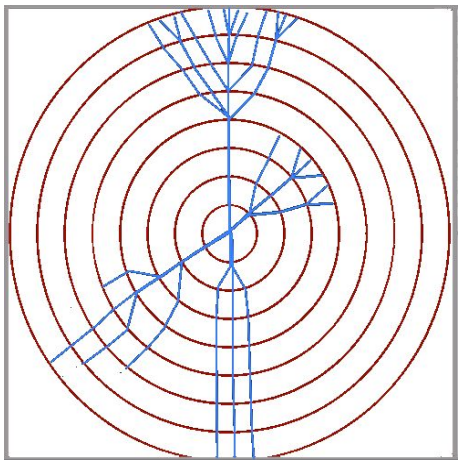

図 1: Radial Layout の例

1.2　論文の概要

計算機上の動作の可視化を行っている、最も有名な機関が、IEEE の一部である Infovis (Information Visualization)[1)] と呼ばれる研究団体である。この研究団体では、WEB やプログラム内のバグなど、膨大なデータベースを効率的に可視化する研究や、新たな可視化アルゴリズムの開発を行っている。この中で注目したアルゴリズムが、**Radial Layout**[1] と呼ばれるグラフの描画アルゴリズムである。これは、ある節点を中心として、同心円上に節点を配置する方法である。中心点に隣接する節点を最も小さい同心円上に配置し、それらの節点に隣接する節点を次に小さい同心円状に配置する。この動作を繰り返すことにより、図 1 のような木構造の描写が実現される。この描画方法では、枝ごとの描画領域を、含まれる節点の個数などにしたがって、動的に配置する。そのため、WEB のリンク構造 [2, 3] など、膨大かつ事前に構造の予測が不可能な木構造を省略することなく描写できる。全体的な把握を要する再利用表の描写に最も適したアルゴリズムと判断した。

第 2 章　区間再利用機構

プログラムの基本性能は、レジスタや主記憶アドレスを参照（入力）し、各命令区間に応じた処理のあと、結果をレジスタや主記憶アドレスに格納（出力）することである。命令自身が変更されない限り、入力が同じである間、実行結

[1)] http://www.infovis.org/

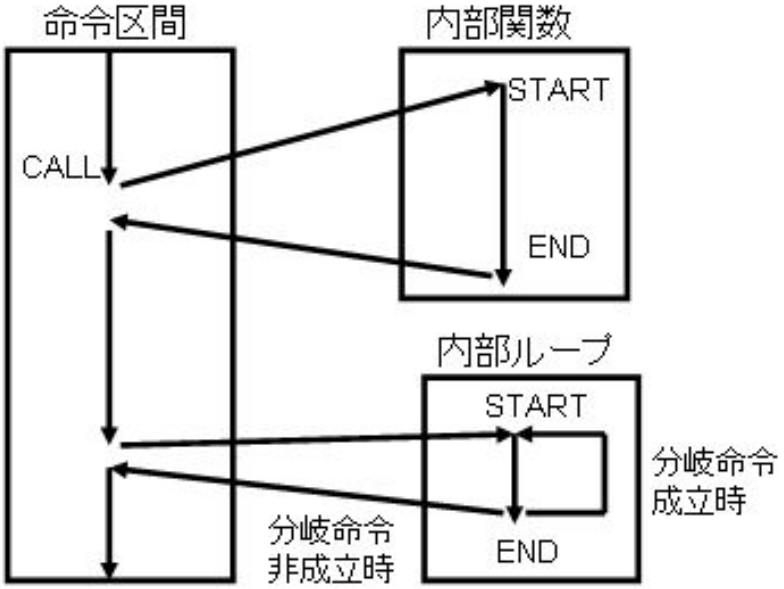

図 2: 内部命令との入出力関係

果も同一となる。区間再利用機構は、特定の命令区間における入出力情報を再利用表を用いて記録し、同一の命令区間が同一の入力値によって呼び出された場合に、命令実行を省略する。この手法には、コンパイラによる専用命令の埋め込みを前提とし、ハードウェアが識別するもの、ソフトウェアのみによるもの、双方を用いるものなど、様々なものが提案されている。本論文では、SPARC ABI(Application Binary Interface) の規定に基づき、命令区間を自動的に判別する機構を仮定する。この機構には、プログラムにおいて再利用のための専用命令が不要であり、既存のロードモジュールをそのまま利用した高速化が実現できるという利点がある。[4] 一般的なプログラムにおける命令区間としては、図 2 のような関数とループが挙げられる。本論文では、内部命令を飛び越した再利用を狙うため、始点と終点を容易に特定できる関数およびループを対象とする。命令区間内において、レジスタや主記憶アドレスより参照した値、すなわち、引数や大域変数が入力列となり、レジスタや主記憶アドレスへ格納された値が出力列となる。

2.1 並列事前実行機構

並列事前実行とは、再利用技術を利用した、非対称な SpMT である。並列事前実行機構では、再利用を行うプロセッサ (**Main Stream Processor**：以下、MSP

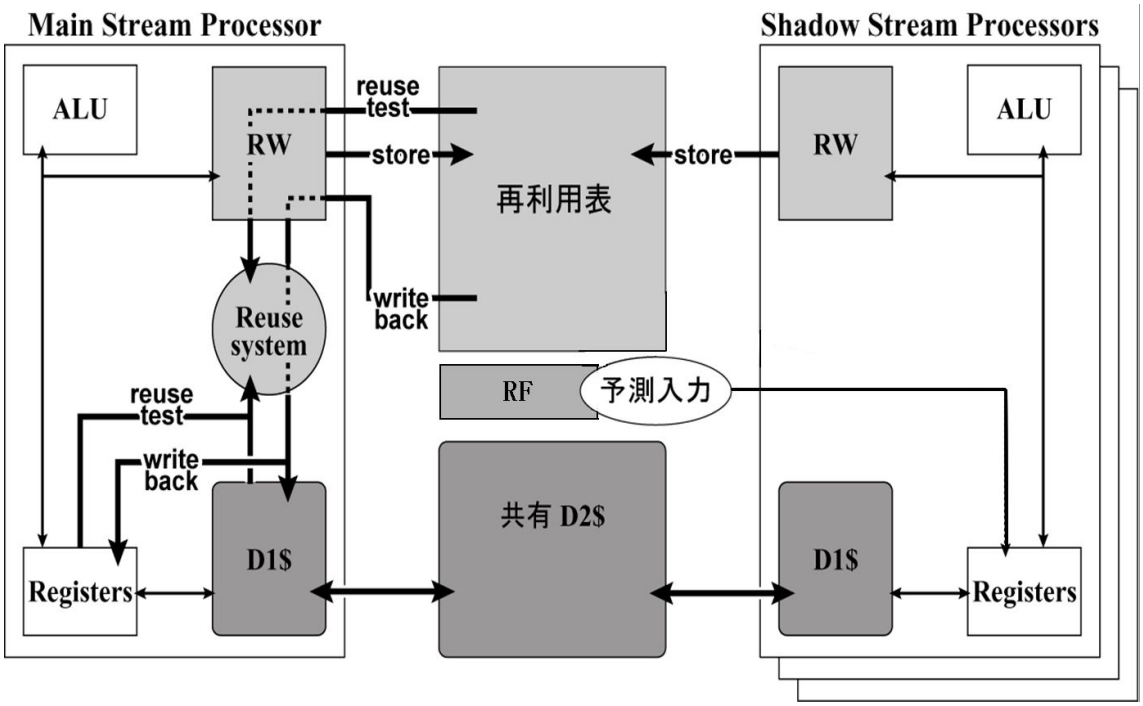

図 3: 並列事前実行機構

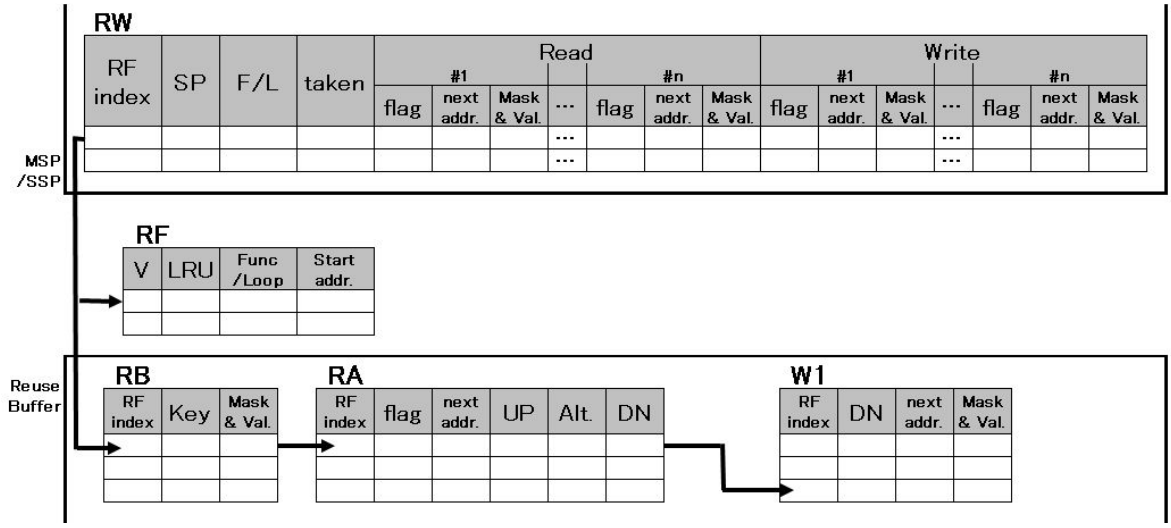

図 4: RW から再利用表への記録

と略す) に加え、投機的実行を行うプロセッサ (**Shadow Stream Processor**：以下、SSP と略す) を複数台設ける。一般的な SpMT とは異なり、MSP は常に通常実行のみを行い、SSP は常に投機的実行のみを行う。SSP は、予測された入力値を用いて、命令区間の実行を MSP に先駆けて行い、結果を再利用表に登録する。その後、MSP から同一の入力が来た場合、再利用が行われ、処理時間が短縮される。図 3 中央に示した再利用表が、MSP と SSP の間で、多対 1 のデータ引継ぎを可能としている。また、投機的実行された入出力も通常実行と同様の仕組みで再利用表に記録されるため、投機的実行時点で予測が失敗しても、それをキャンセルする必要が無い点が長所となる。図 3 の通り、各プロセッサには、レジスタ、キャッシュ、演算器の他に、各命令区間の実行時に一組の入出力パターンを記録する**再利用ウィンドウ (RW)** を設けている。RW は、現在実行している最内命令区間から最外命令区間までの各々について、一組の入出力を時系列に並べ、アドレスを連想検索させる表構造をとっている。最内命令区間をレベル 1 とし、外側の命令ほどレベル数は上がる。このとき、入出力はレベル 6 の命令区間まで登録できる。より内側に新たな命令区間を検出した場合、レベル 6 の記録を破棄し、登録中のレベル 1〜5 の記録をレベル 2〜6 に読み替え、空いた RW 列をレベル 1 として使用する。RW に登録された入出力は、命令区間実行完了時に、再利用表に送られる（図 4)。始点アドレス、終点アドレスなど、実行された命令区間の情報は、**関数管理表 (RF)** に記録される。関数の場合、始点、終点などの情報は、実行終了後、外側の命令に復帰した直後に RF に記録される。ループの場合は、後方分岐成立時に、分岐先を始点とし、後方分岐命令自身を終点として記録する。すなわち、ループ 1 回目実行中や、後方分岐不成立であった場合、ループの情報は RF に記録されない。RF では、予測値の生成も行っている。MSP から命令区間が実行あるいは再利用され、RW から再利用表へ蓄積すると同時に、入力列のみを入力履歴として RF に格納する。入力履歴は、RW と同様、入力値が時系列に並べられた表構造を 2 セットとった FIFO である。この入力履歴により、予測値を求め、SSP のレジスタに格納する。SSP はこの予測値を入力として参照することにより、投機的実行を行う。本論文では、表 1 のパラメータに従い、再利用機構を構成している。

D-Cache	64 Kbytes
Line Size	64 bytes
Ways	4
Cache Miss	20 cycles
Register Window	4 sets
Register Window Miss	20 cycles/set
Load Latency	2 cycles
Integer Mult.	8 cycles
Integer Div.	70 cicles
Floating Add/Mult.	4 cycles
Single Div.	16cycles
Double Div.	19cycles
RW Depth	6
RF Entry	256
Read Address	256
Write Address	256
RB Entry	4096
Time Stamp ID MAX	4
RB Purge Timer MAX	RB/TSID - RB/(TSID $*$ 4)
RB(Reg.) $\Leftrightarrow$ Register Compare	1 cycle
RB(Reg.) $\Rightarrow$ Cache Compare	4 bytes/cycle
RB(Write) $\Rightarrow$ Cache Write	4 bytes/cycle
RB(Reg.) $\Rightarrow$ Register Write	1 cycle
SSP Local Memory	64 Kbytes

表 1: 区間再利用機構のパラメータ

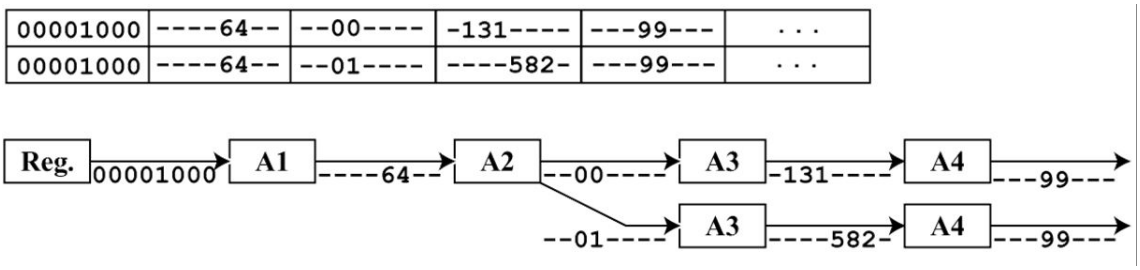

図 5: 再利用表が作る木構造

2.2 汎用 CAM における再利用表の構造

入出力を RW から再利用表に登録する際、同一アドレス内で入力値が異なる部分に flag を立てることにより、概念的に図 5 の木構造をとる。再利用表の構成は、以下の 3 点に分かれている。

入力記録表 (RB) 入力値そのものを"Mask/Val."に記憶する。同時に、親のインデクスを"Key"として格納する。RB は CAM を用いて実装され、Key と Mask/Val. の 2 要素より連想検索を行う。(木構造の枝)

アドレス記録表 (RA) 次に参照すべきレジスタおよび主記憶のアドレスを"next addr."に格納する。"flag"は、同一アドレスに複数回のストアが起こった、すなわち、この節点で枝分かれがおきていることを表すフラグである。さらに、flag が 0 の場合、次に flag が 1 (枝分かれ部分) もしくは E(終端) となっている RA のインデクスと主記憶アドレスをそれぞれ"DN", "Alt."として格納する。RB から RA への検索は、RB 自身のインデクスを用いた 1 対 1 対応のものである。よって、RA は RAM 上に実装される。(木構造の節点)

出力記録表 (W1) 出力値を格納する。同時に、次の出力値が格納されているインデクスを参照することにより、長大な出力列を汎用 CAM に記憶できる折りたたみを実現している。

一旦、再利用表に登録されれば、同一の命令区間を呼び出す前に再利用表の検索が行われる。その検索の手順は、以下の通りになる。(図 6 参照)

1. 木構造の根を表す初期 Key(FF) と先頭アドレス (00001000) により、RB を連想検索する。この場合は、インデクス 00 が該当する。
2. インデクス 00 の RA で、次に参照すべきアドレスは A1 であるが、flag が 0 であるため、値を読み出す必要は無い。

3. DN より、次に枝分かれがおきているアドレス A2 のキャッシュ値を参照。
4. 再び、DN の値 (01) と、アドレス A2 のキャッシュ値 texttt(–01—-) により、RB を検索すると、インデクス 05 が該当する。
5. インデクス 05 の RA では、flag が E となっているため、RW と再利用表は完全に一致するとみなされる。
6. RA に格納されている W1 のインデクス番号から、出力値を参照する。

命令区間の実行や再利用の後、他の命令区間の出力によって、参照するアドレスの値が変化する場合がある。例えば、図 7 のように、アドレス A4 に該当する主記憶の値が---89---に変化した場合、アドレス A4 を参照するインデクス 06 の RA において、flag は E から 1 に変化する。この時点で再利用表の検索を行うと、インデクス 05 の RA における DN の値 (06) と、アドレス A4 のキャッシュ値 (---89---) に一致する RB は存在しないため、再利用は実行されない。この場合、改めて命令区間を実行する。その後、RW から再利用表への登録の際に、Key を 06、Mask/Val. を---89---とする新たな RB, RA エントリを追加する。

2.3 再利用オーバーヘッド機構

再利用は、前節に示した仕組みをとって実行される。よって、再利用不可であった場合、処理時間は再利用機構を使用しない場合に比べて増大する。再利用表の検索や登録にかかったステップ数が検索コストとして加わるためである。本論文では、無駄になる検索コストの総計が、再利用によって削減できるサイクル数の合計よりも大きくなる場合、その再利用は効果が無いとみなす。RF において、区間ごとに過去規定回数分の呼び出しに対して再利用を行ったか否かを保持する。その規定回数分内での無駄な検索コストが、再利用によって削減されるサイクル数を上回るならば、この命令区間における再利用は効果がないとみなす。以後は、再利用の実行自体を止め、再利用表への登録や、SSP による事前実行も行わない。

第 3 章 再利用表の可視化とユーザーインターフェース

この章では、第 2 章で示した再利用機構の動作を Radial Layout を用いて可視化する方法（以下、**可視化機構**）について説明する。可視化機構は、図 8 に

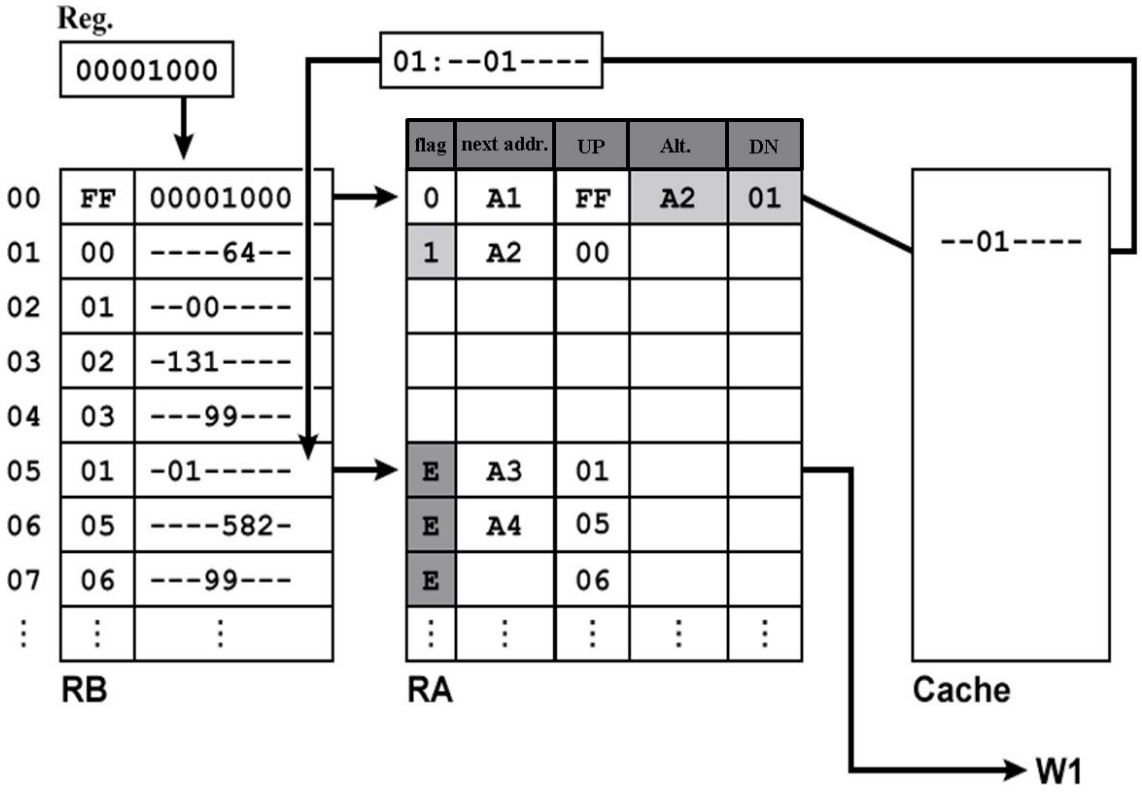

図 6: 再利用が実行される仕組み

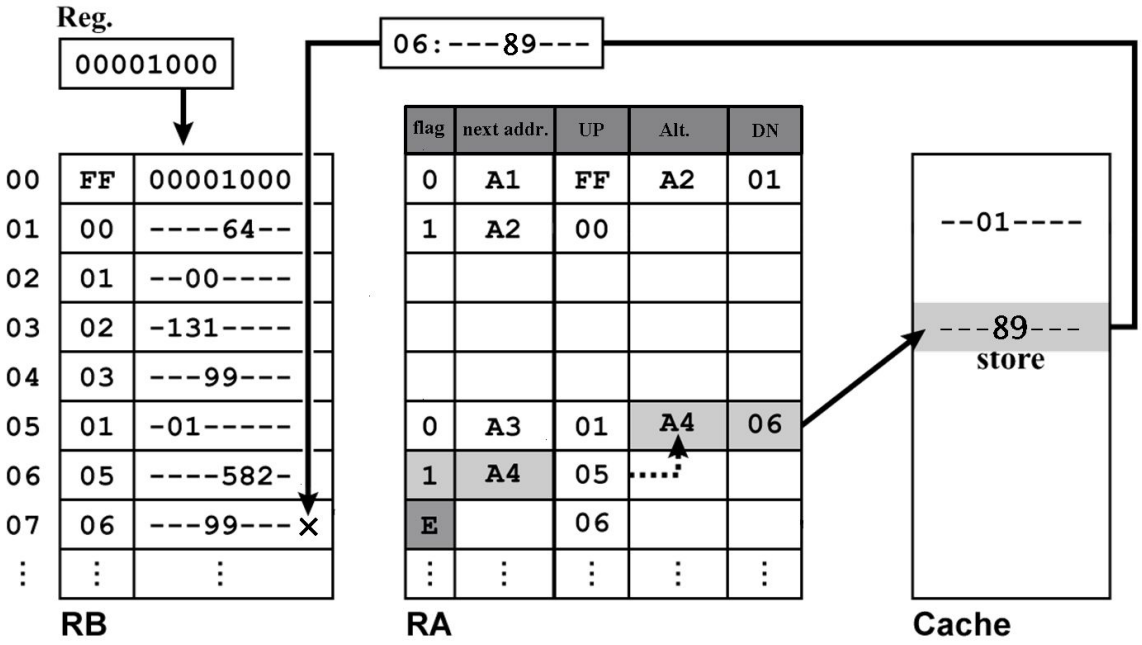

図 7: 再利用が実行されない場合

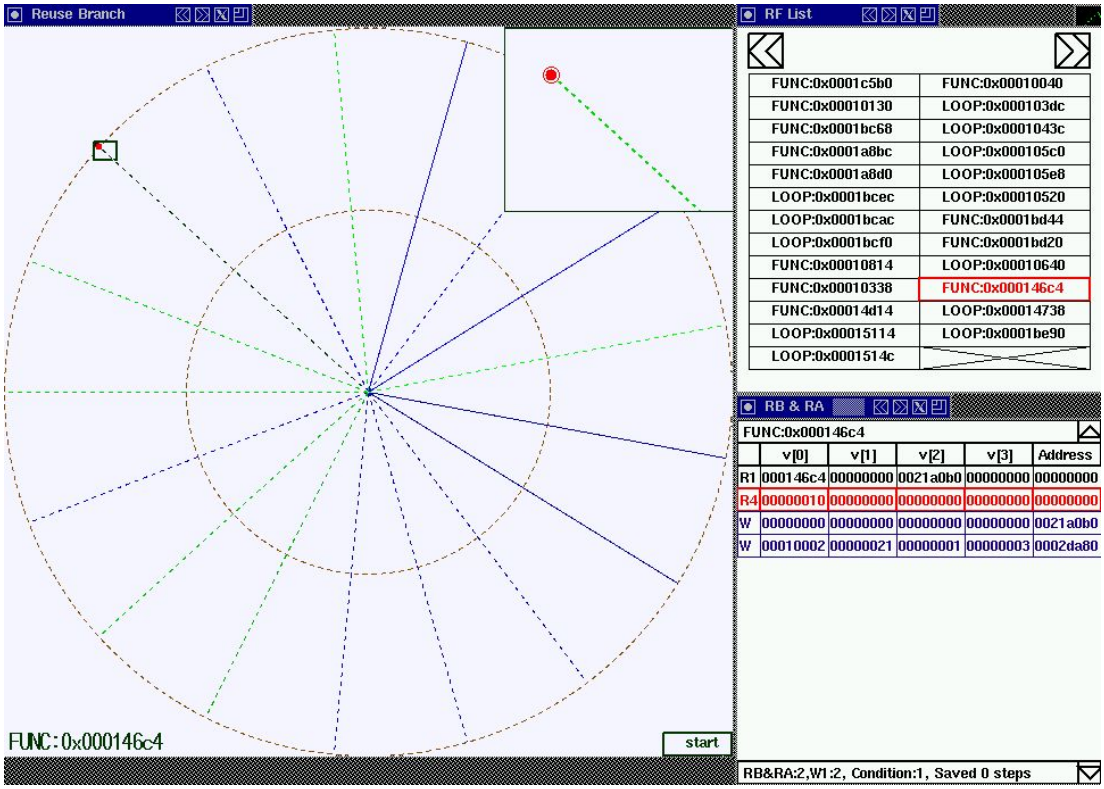

図 8: 可視化機構の凡例

示す通り、以下の 3 つのウィンドウに分かれている。

再利用表ウィンドウ（左） この機構で最も重要となるウィンドウである。再利用表に記録されている木構造を Radial Layout で可視化する。

RF ウィンドウ（右上） RF に蓄えられている命令区間の先頭アドレスおよび、関数・ループの区別を記述している。描写される順番は、命令区間が終了し、RF に登録された順番に従う。よって、内部に命令区間がある場合、レベル 1 の命令が先に、最外命令区間は最後に来ることになる。

入出力列ウィンドウ（右下） 検索している入出力列、およびその列の再利用の状態を記述する。入出力の検索については後述。リアルタイム可視化時には、最近に登録もしくは再利用された入出力列がここに表示される。

3.1 Radial Layout の実現

Radial Layout では、図 9 の手順に従って座標が決定される。

(1) Radial Layout は、根 O を原点にした xy 平面状に座標を置く。しかし、X window の仕様上、通常の xy 平面とは異なり、y 軸は下方向になり、節点

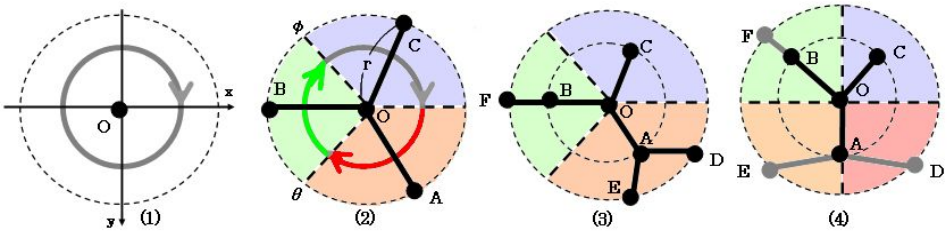

図 9: Radial Layout の座標決定

の位置ベクトルと x 軸とのなす角も時計回りに従って増大していく。各接点には**描画領域**が角度の形で与えられる。図 9(1) のように、原点の描画領域は $[0, 2\pi)$ となる。

(2) 子の描画領域はその下にある枝分かれの本数（以下、枝数）にしたがって決まる。根 O に 3 つの節点 A, B, C が加わった場合、根 O の枝数は 3、3 点 A, B, C の枝数はそれぞれ 1 となる。よって、描画領域は均等に割り付けられる。点 A の描画領域は $[0, \frac{2}{3}\pi)$, 点 B の描画領域は $[\frac{2}{3}\pi, \frac{4}{3}\pi)$, 点 C の描画領域は $[\frac{4}{3}\pi, 2\pi)$ となる。各接点の座標は、描画領域内の角を二分する線上に置かれる。最も外側にある同心円の半径が r（固定値）, その同心円に対応する最大の深さが d_{max} の場合、ある節点 K の深さが d_k, 描画領域が $[\theta, \phi)$ とすると、節点 K の座標 (x_k, y_k) は以下の計算式で定義できる。

$$\begin{cases} x_k = r\frac{d_k}{d_{max}}\cos(\theta + \frac{\phi-\theta}{2}) \\ y_k = r\frac{d_k}{d_{max}}\sin(\theta + \frac{\phi-\theta}{2}) \end{cases} \tag{1}$$

すなわち、図 9(2) の場合、節点 B の座標は $(r\cos\pi, r\sin\pi)$ となる。

(3),(4) 枝数に偏りが生じた場合、描画領域は枝数の多い節点に優先して割り付けられる。新たに、2 節点 D, E が A の子として、点 F が B の子として加わった場合、4 点 O, A, B, C の枝数はそれぞれ 4, 2, 1, 1 となる。このとき、点 A の描画領域は $[0, \pi)$, 点 B の描画領域は $[\pi, \frac{3}{2}\pi)$, 点 C の描画領域は $[\frac{3}{2}\pi, 2\pi)$ と変化する。また、2 点 D, E の描画領域は、A の描画領域を 2 等分し、それぞれ $[0, \frac{1}{2}\pi)$, $[\frac{1}{2}\pi, \pi)$ となる。点 F の描画領域は、B の描画領域をそのまま引き継ぐ。これらの描画領域から座標を算出すると、図 9(4) のようになる。

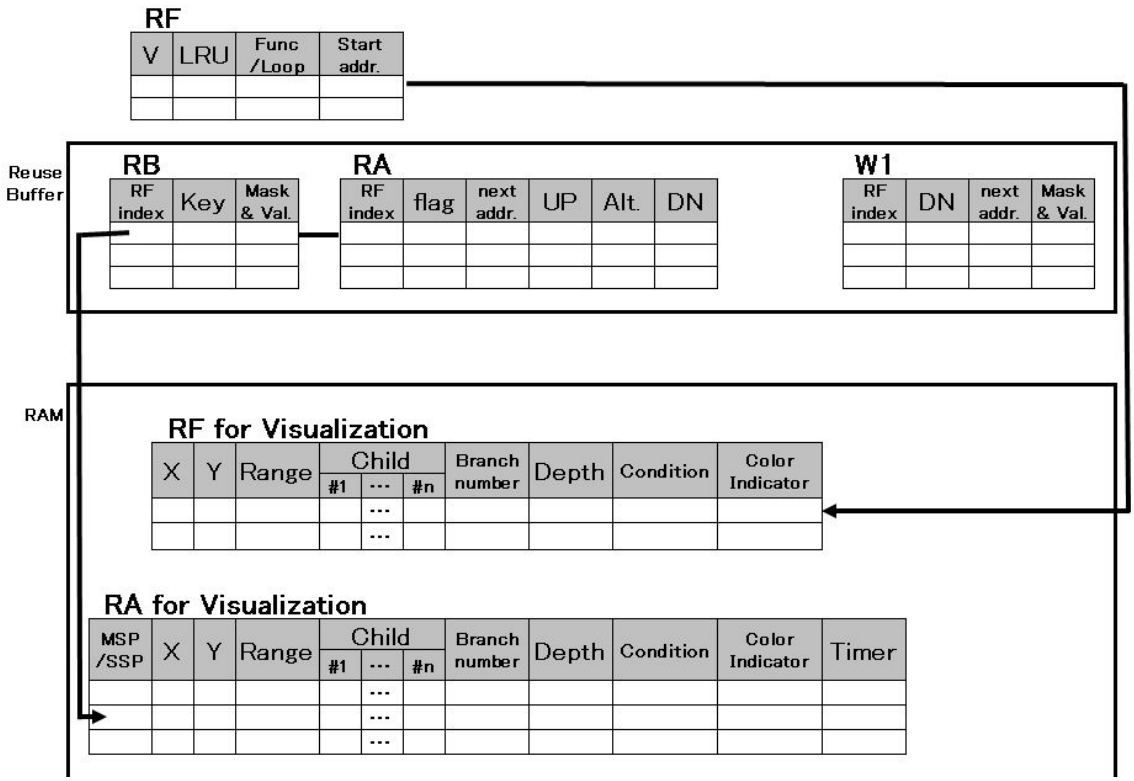

図 10: 再利用表の拡張の様子

3.2 再利用表の拡張

第 2 章で説明したとおり、再利用表の検索では、1 節点を比較し終えたら、次に枝分かれがおきている部分を参照し、もう一度 RB 全体から一致するエントリを探す仕組みをとっている。RB に使用される CAM は、一般的に使用される RAM とは異なり、連想検索が定数時間で実行可能である。一方、可視化のためには、再利用表の時系列を親子関係として、親のインデクスと `Key` の値とを比較しながら、再度 CAM を検索する必要がある。これを避けるため、本稿では、子のインデクスを登録する配列 `child` を新たに作成した（図 10 参照)。本論文では、再利用表における親子関係を次のように定義する。

1. 各 RA エントリの `UP` に格納されている値をを親のインデクスとみなす。UP が初期 `Key(FF)` であった場合、その RA エントリの親インデクスは、入力列を実行する命令区間に対応する RF のインデクスとなる。
2. RW から再利用表に登録するときに、新規に登録された RA のインデクスを、親 RA が持つ `child` の最後尾に加えることで、木構造の親子関係を実

現している。

この他にも、各 RA,RF ごとに、次のような要素を拡張している。

X, Y, Range　再利用表ウィンドウにおける座標や描画領域 (3.1 参照)

Branch number　描画領域の確保に必要な枝数

Depth　入力列内におけるこの節点の順番（深さ）

Condition　再利用されたことを表すフラグ

Color Indicator　節約したステップ数もしくは再利用された回数 (色分け要素)

これらの情報は、対応する RF, RA のインデクスより、1 対 1 で検索を行うため、RAM 上での実装が適している。

3.3　再利用された入力列の抽出

再利用表に組み込まれた入力列は、再利用されない場合、青の連続直線で可視化される。そして、再利用が実行された時点で枝の色が変化する。枝の色は、再利用によって短縮されたステップ数が大きくなるにしたがって、**緑** → **黄** → **赤**へと変化していく。短縮されたステップ数が 256 を超えると、枝の描画色は黄色から赤へと近づき、512 ステップを越える再利用の場合、枝の描画色は完全な赤となる。再利用された入力列の色分けには、もう一つ、再利用が行われた回数に従った色分けがある。これは、初めて再利用された場合は緑色になり、以降、同一の入力列が再利用されるごとに、その色を緑 → 黄 → 赤へと変化させる。これも同様に、再利用された回数が 256 を超えると、枝の描画色は黄色から赤へと近づき、512 回以上再利用された場合、枝の描画色は完全な赤となる。

3.4　主スレッドと投機スレッドの区別

MSP による主スレッドと SSP による投機スレッドは、2.1 で説明したように、その意味合いにおいて大きく異なる。よって、再利用が行われた入力列が、主スレッドによる過去の入力なのか、投機スレッドによる予測値なのかを判別する必要がある。可視化機構では、MSP による入力列を実線で、SSP による入力列を点線で描写する。図 8 の場合では、SSP による予測値は半分以上が再利用されているが、MSP による入力列は全く再利用されていないと判別できる。

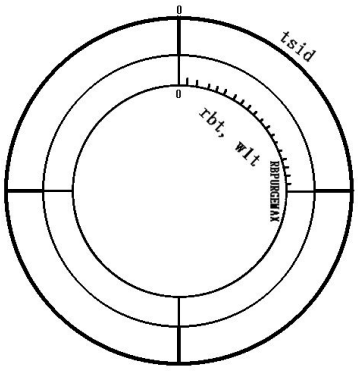

図 11: tsid と rbt, w1t との関係

3.5 再利用表のエイジング

メモリの容量は無限ではない。長大な入出力列を登録し続けていれば、いつかはその容量も尽きてしまう。そのため、区間再利用機構では、最も古い時間帯に登録された入出力列を削除する仕組みをとっている。再利用表における時系列の管理は、以下の 2 つの要素から成り立っている。

RB/W1 Purge Timer(以下、rbt, w1t) それぞれ、入力値と出力値が新たに登録されるたびに 1 ずつ増えていく。

Time Stamp ID(以下、tsid) rbt もしくは w1t が一定値に達するごとに、値を一つ進める。この時点で rbt と w1t は 0 に初期化される。

このとき、tsid は図 11 のように待ち行列の形式となっており、一定値に来たらもう一度 0 に初期化される。tsid を進めた際、同じ tsid を持つ RB, RA, W1 は、最も古い再利用表として消去される。可視化機構では、この 2 つの要素から各 RA エントリごとの時刻を算出する。$rbt \geq RBPURGEMAX$ のとき、tsid が進むとすると、時刻 $time$ は

$$time = tsid \times RBPURGEMAX + rbt$$

で求められる。可視化の際には、入力列が登録もしくは再利用された時刻と、現在の時刻の差を t として求める。各枝の RGB 値から、t に比例する数値を減算することにより、可視化される枝は薄く暗くなり、最後は消去される。

3.6 関数・ループ区間ごとの可視化

再利用表の可視化には、全体的な動作の可視化と、命令区間ごとの再利用表の可視化が必要となる。そのため、可視化機構では、根（図 5 の Reg. の部分）

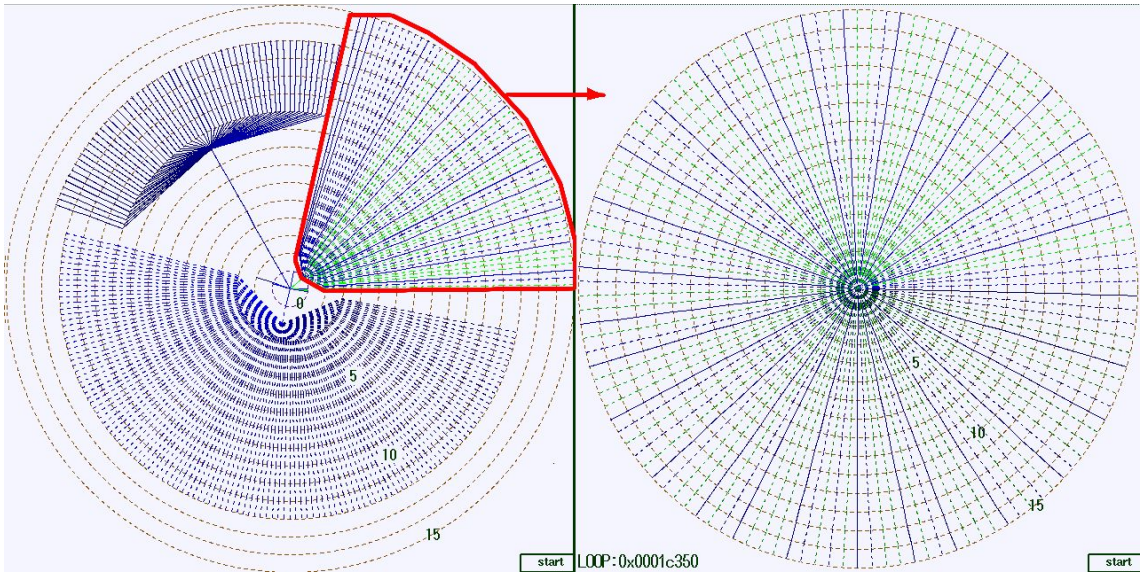

図 12: 赤枠内が抽出される

から描写し、RF 内の命令区間全てに蓄えられている再利用表を描写する方式（図 12 右側）と、一つの命令区間を抽出し、その中で使用された再利用表を描写する方式（図 12 左側）がある。この抽出は RF ウィンドウ内、関数アドレスが表示された部分をクリックすることにより行われる。もう一度全体の可視化に戻したい場合は、×字が表記されている (RF が NULL となっている) 部分をクリックする。

3.7　入力列・出力列ごとの参照

描写された入出力列を具体的に検索したい場合は、再利用表ウィンドウに描写されている枝をクリックすればよい。ただし、場所をわかりやすくするため、検索は以下の 2 段階で行う仕様になっている。

1. 枝の節点近くをクリックすると、その周辺の拡大図が上部に子ウィンドウとして表示される。
2. 子ウィンドウ内の節点をクリックすると、その節点が属する入出力列が入出力列ウィンドウに表示される。

図 8 の再利用表ウィンドウ中で、白の直線で表された部分が検索された入力列であり、右上に赤の二重丸で表された部分がクリックした節点である。この節点は、入出力列ウィンドウでは赤く表示される。

3.8　リアルタイムな可視化

可視化機構には、再利用実行後に最終状態を記述するモードと、実行中、一定のステップごとに状態を記述するモードがある。後者のモードでは、クリックによる再利用の計測に制限がある。命令区間の抽出はリアルタイム可視化中でも可能である。しかし、入出力列の検索は、リアルタイム描写中にはできない。検索を行うためには、一旦、プログラムの実行を停止させる必要がある。リアルタイム可視化中に再利用表ウインドウをクリックすると、プログラムの実行が一時停止され、入出力の検索が可能となる。実行を再開する場合は、再利用表ウィンドウの右下にある"start"をクリックすればよい。

第4章　計測および考察

可視化機構を用いて、Stanford-integer の動作を調査した。Stanford-integer は、10 種類のプログラムからなる。それぞれのプログラムについて、MSP のみを使用する場合、MSP および 3 台の SSP を使用する場合の 2 通りに分けて測定を行った。

Queens

8 つのクイーンをお互いが取れない位置に置く 8 クイーンパズルを解く。再利用が顕著に現れたのは、クイーンが置けるかどうかを判定する関数 Try（付録図 A.2）と、Try を以下のように再帰的に実行させるループ（付録図 A.3）であった。

```
while ( (! *q) && (j != 8) ){
  j = j + 1; *q = false;
  if ( b[j] && a[i+j] && c[i-j+7] ){
    x[i] = j; b[j] = false; a[i+j] = false; c[i-j+7] = false;
    if ( i < 8 ){
      Try(i+1,q,a,b,c,x);
      if ( ! *q ){
        b[j] = true; a[i+j] = true; c[i-j+7] = true;
      }
    }
    else *q = true;
```

```
    }
  }
```

(j は Try の内部変数、あとの変数は Try の引数である。)

関数 Try は、配列 a, b, c, x の全体が入力値となっているため、入力列の長さに 2〜3 倍もの差が見られる。付録図 A.3 のループは、i や q の入力値によって、値を参照、格納する回数が変化する。よって、入力列の長さも大きく変動し、その差は 3 倍以上にもなる。再利用の効果は双方とも非常に高く、1000 以上のステップ数を短縮できた入力列も存在していた。しかし、再利用された入力列は、長さが 10 前後の短いものに集中していた。付録図 A.2, 付録図 A.3 では、右側が MSP のみによる再利用表、左側が事前実行を使用した再利用表となる。双方を比較すると、事前実行による効果の違いはさほど見られない。付録図 A.2 にいたっては、投機スレッドによる再利用は全く見られなかった。むしろ、登録数を増やした分、再利用された入力列が消去されてしまう部分も見られた。付録図 A.1 のように、初期入力を行うループに事前実行による再利用が見られるが、その効果は、一つ一つの短縮されたステップ数から見ても、再利用された入力列の個数から見ても、付録図 A.2 や付録図 A.3 に比べてかなり小さい。このため、Queens においては、事前実行の効果は期待できないと考えられる。

Towers

ハノイの塔問題を解くプログラムである。付録図 A.4 を見ての通り、再利用の行われる部分は一部分に固まっている。再利用が行われる関数は、右上から順に、円盤を持ち上げる関数 Pop（付録図 A.5 左）, 持ち上げた円盤を別の塔に乗せる関数 Push（付録図 A.6）, および円盤を積載可能な塔を見つける関数 Getelement（付録図 A.5 右）の 3 つのみであった。Push と Pop を使って積み木を動かす関数 Move と、その Move を再帰的に実行する関数 tower は、実行回数・入力の長さ共に大きく、全体図の右側をかなり大きく陣取っている。しかし、この部分では再利用は全く実行されない。しかし、再利用が行われる関数 Push, Pop, Getelement は、一つ一つの効果自体は小さいものの、入力列の 9 割近くが再利用されている。また、Push においては、事前実行による入力列は全く再利用されず、最終的には MSP による実行のみが残る。付録図 A.5 左側の Pop においても、右下部に再利用されない投機スレッドが固まっていることから考えて、ある程度 MSP による入力との共通要素が無いと、再利用は行われないことがわかる。これは、入力すべき値の制限が厳しい Towers ならではの特徴

と言える。

Bubble

要素数5000の整数配列をバブルソートする。このプログラムにおいては、MSPからの入力による再利用は全く行われず、事前実行による再利用も、乱数を出力する関数 Rand のみで行われている。（付録図 A.7）この関数は、ソート対象となる整数配列を決める関数 bIntarr 内のループ区間で以下のように実行される。

```
for ( i = 1; i <= srtelements; i++ ) {
  temp = Rand();
  sortlist[i] = temp - (temp/100000)*100000 - 50000;
  if ( sortlist[i] > biggest ) biggest = sortlist[i];
  else if ( sortlist[i] < littlest ) littlest = sortlist[i];
};
```

このループ区間の入力列は付録図 A.7 左側の上部に示されている。このループは、実行回数は非常に多いものの、再利用は一回も行われていない。これは、乱数入力による temp の共通性の無さと、処理時間の増大によるオーバーヘッドの二点が理由と考えられる。その後のバブルソート実行時にも、再利用の実行こそ無かったものの、面白い特徴が見られた。この部分は、配列の交換を行う内部ループと、そのループを配列の終端から先頭まで繰り返し行う外部ループとに分かれている。外部ループにおける入力は、未ソートの配列全体となっており、時系列にしたがって、付録図 A.9 のように長さが短くなっていく。また、内部ループも、最初のころは値の共通部分が無く、枝が放射状に広がっているのに対し、終了間際では、先頭部分に共通性が見られ、ある程度枝がまとまっているのが見て取れる。

Quick

上の Bubble と同様の整数配列に対し、クイックソートを行う。Bubble と同じく、再利用は、関数 Rand における事前実行のみで発生する。クイックソート実行時には、再帰的にクイックソートを行う関数 Quicksort と、その内部で配列の入れ替えを行うループの中に再利用表が現れる。関数 Quicksort は、命令実行中に参照される配列の長さに大きな変動があるため、配列の状況によって、入力列の長さが大きく変動する。MSP のみの実行でも、入力列の長さは、10 前後〜36 と、3 倍以上の差が見られる（図 13 左側）。しかし、事前実行による予測値を加えると、その差はより大きくなる。大きい場合には、100 を超え

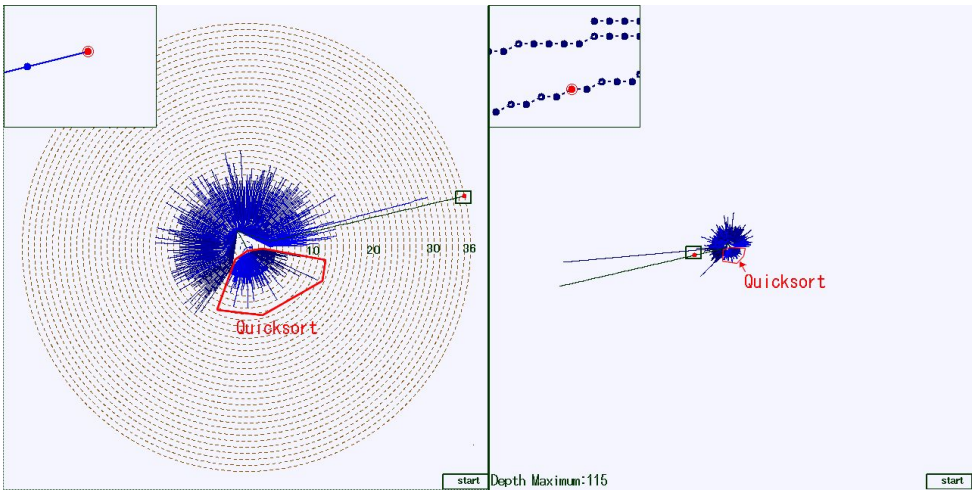

図 13: (Quick)Quicksort 実行時の再利用表

る長さの入力列が登録されることもあった（図 13 右側）。このことから、入力列の長さが不定期に変わるようなループに対しては、再利用や事前実行の効果は期待できないことがわかる。

Trees

上記の 2 つのソートと同様の整数配列に対し、ヒープソートを行う。これは、上の 2 つのソートとは異なり、新たな枝を作るたびに malloc 関数を使用して二分木を確保する。この malloc 関数自身やその内部の命令区間において、再利用が発生する。malloc 自身、多数の内部命令区間を持つ非常に処理時間の長い関数であるため、再利用が発生した場合に 200〜300 ステップの処理時間が短縮される場合もある（付録図 A.10）。malloc 内部の命令区間においても、一つ一つの効果は小さいものの、かなりの頻度で再利用が行われている命令区間が多い。命令区間によっては、付録図 A.11 左側のように登録中の入力列全てが再利用されたり、同図右側のように、一定期間ごとにまとまった再利用が起こる場合もある。しかし、この再利用も、ほとんどが SSP からの入力によるものである。事前実行の効果が最も顕著に現れる例と言える。

Intmm, Mm

それぞれ、整数要素と浮動小数点要素からなる 2 つの行列の積を計算する。計算を行う 41×41 の正方行列に入力される値は、上記 3 つのソートと同じく、関数 Rand によって決定される。この部分でも、事前実行による再利用が行われるが、実行回数が整数配列に比べてかなり大きい分、見た目の再利用の効果

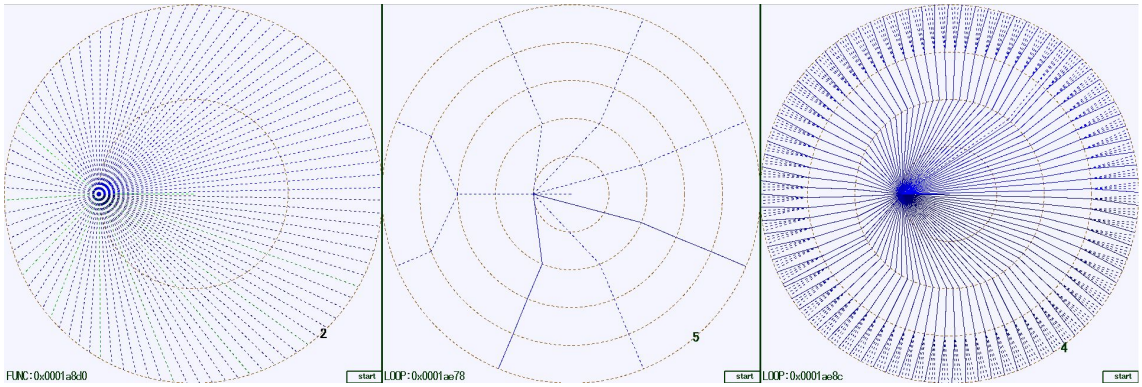

図 14: (Intmm) 左：関数 Rand, 中：行ごとの入力, 右：要素ごとの入力

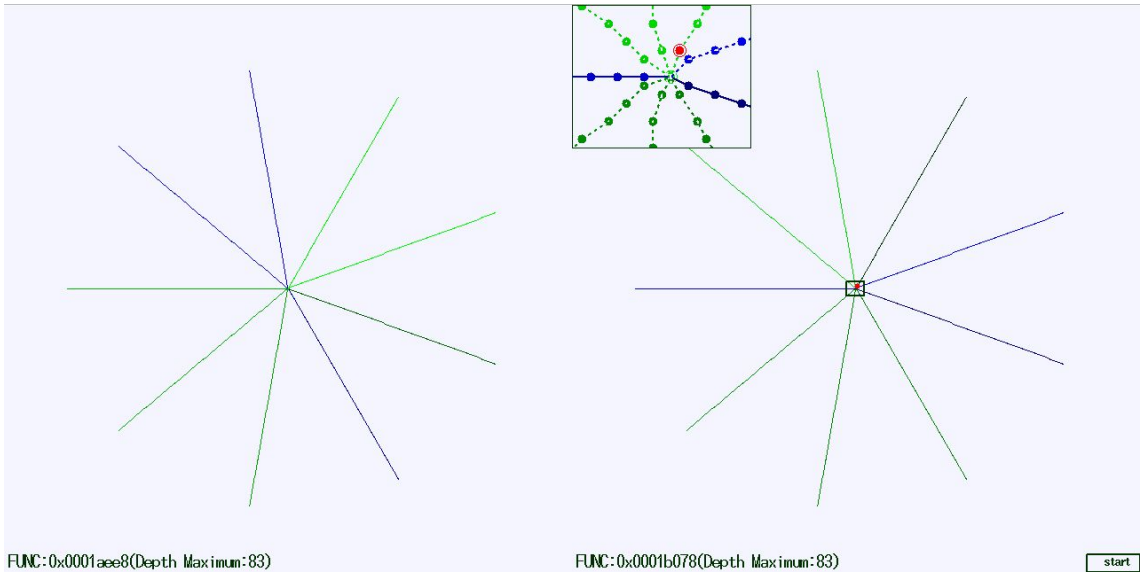

図 15: (Intmm)Innerproduct において再利用された列

は比較的小さい（図 14）。このプログラムにおいて、再利用が最も効果的に現れるのは、要素ごとに行列積の値を求める関数 Innerproduct である。この関数は 2 つの行列全ての値が入力されるため、入力列の長さは、83 にも及ぶ。この関数は、要素同士の積を行列の長さ分だけ繰り返す関数となっており、かなり処理時間が長い。よって、再利用によってかなりの処理時間が短縮できると考えられる。しかし、長さ 83 にもなる長大な入力列を検索するため、検索時間の増大や登録できる入力列の少なさが原因となり、一つ一つの再利用で節約できるステップ数は、1 桁程度に留まる。また、この入力列の参照先は、単純変動するループによって決まるため、MSP のみの実行では、再利用は発生しない（図 15）。

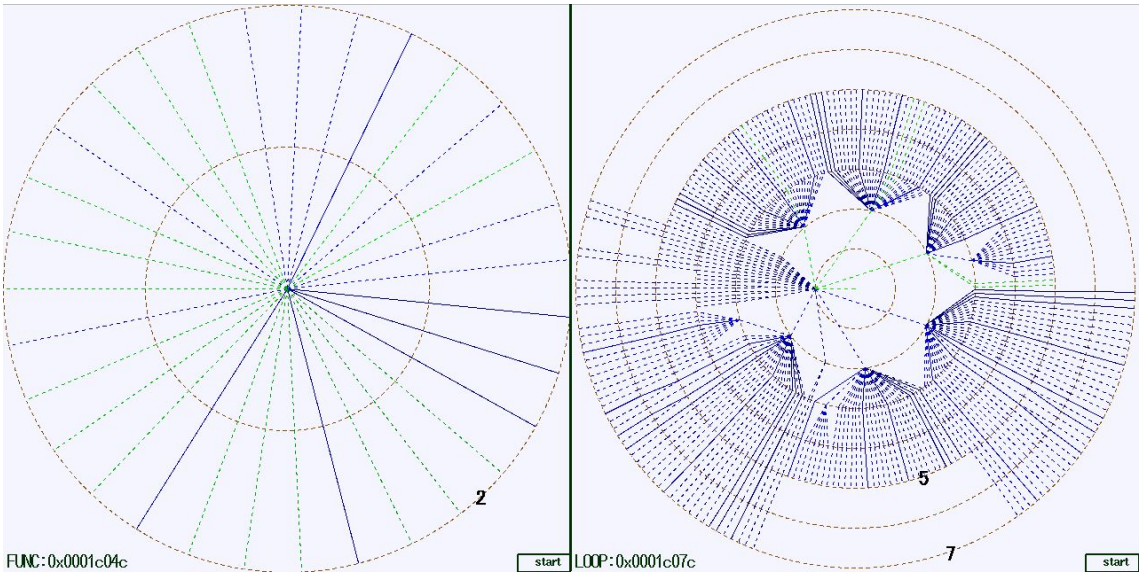

図 16: (FFT) 関数 Cos(左) とその内部のループ (右)

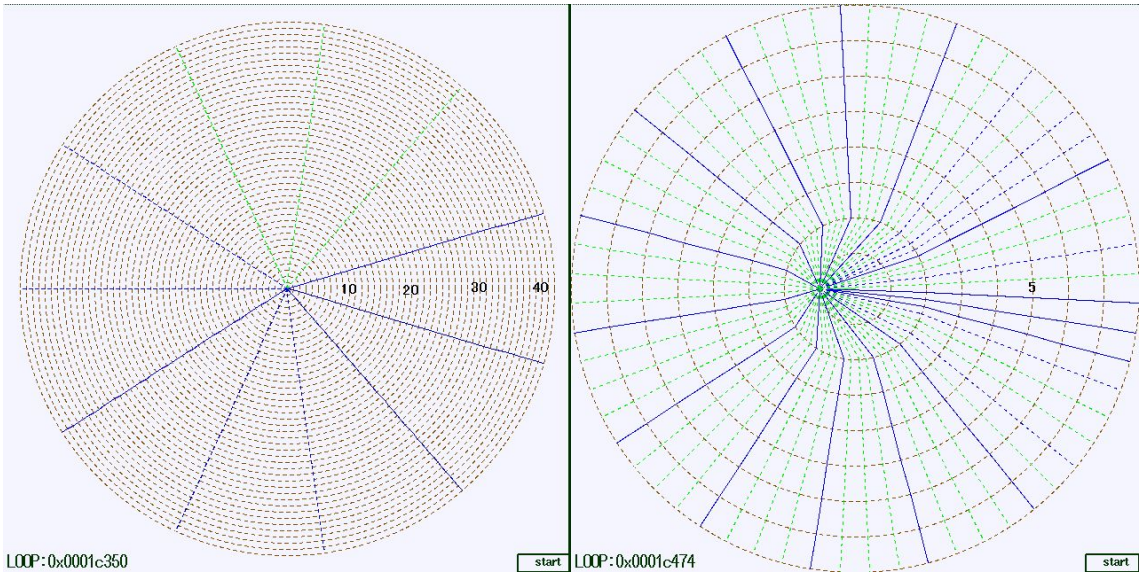

図 17: (FFT)Fft における再利用表

FFT

高速フーリエ変換を行う。このプログラムでは、MSP のみでは再利用は起こらないが、事前実行による再利用が顕著に発生する。再利用が発生する部分は、初期入力の際に余弦の値を求める関数 Cos と、実際にフーリエ変換を行う関数 Fft 内のループである。まず、Cos は、呼び出す関数 Exptab において

```
for ( i=1; i <= 25; i++ ) {
```

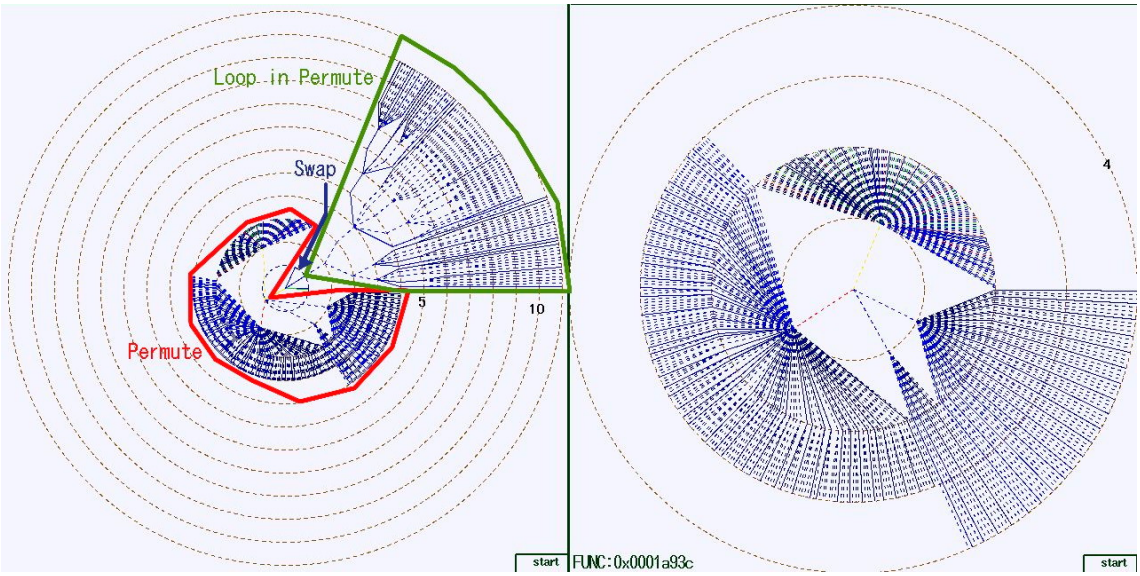

図 18: (Perm) 左：Permute 実行時、全体の再利用表　右：Permute の再利用表

```
        h[i] = 1/(2*Cos( theta/divisor ));
        divisor = divisor + divisor;
    };
```

のように、入力値が単調増加するため、事前実行による再利用が多く行われると考えられる。内部ループでの再利用率が少ないのは、Cos 自身の再利用率が高いためであると考えられる（図 16）。FFT における事前実行の効果は、実際のフーリエ変換時に最も高くなる。この部分は、最大 3 階層にもなる複雑なループとなっているが、変動自体はごく単純で、入力列の長さもほぼ一定のため、事前実行による再利用が多く発生する。図 17 左側のような長大な列が再利用される場合や、図 17 右側のように、事前実行による予測値全てが再利用される場合も観察することができた。

Perm

再帰的に配列上の値の入れ替えを行うプログラムである。このプログラム内では、配列の入れ替えを行う関数 Swap が何度も呼び出される。対象となる配列は長さが短く、入れ替え対象となる 2 値の場所と値は高々252 パターンしか存在しない。しかし、今回のシミュレーションでは、RB エントリを関数ごとに確保せず、プログラム全体で共有する方式を取っているため、Permute と、その内部で Swap を実行するループの 2 つの命令区間で再利用表は使用され、Swap における入力列は観測できなかった。（図 18 左側）このプログラムは、ループ内

での再帰的な実行が多く、MSP のみでの再利用は観測されなかった。しかし、Swap を再帰的に実行させる関数 Permute において、事前実行による再利用が見られた。ここでの再利用は、図 18 右側のように、頻度はまばらで、一極に集中しているものの、200〜800 ステップを短縮する高い効果を得ている入力列も多い。

Puzzle

13 個のピースを組み合わせ、$5 \times 5 \times 5$ の立方体を作るプログラムである。この部分での再利用は、事前実行によるもののみで、主に 2 つの場面に分かれて発生した。

1. 最初に、１２個のピースと、それらを組み合わせる空間を定義する。この部分は、以下のように、単純変化するループで実行されるため、事前実行による再利用が起こりやすいと考えられる。付録図 A.12 のように、事前実行による入力列のほとんどが再利用された例もある。

   ```
   for ( m = 0; m <= size; m++ ) puzzl[m] = true;
   ```

   しかし、各ループの処理は、1 命令程度の非常に小さいものであるため、一つ一つの再利用で短縮されたステップ数は 0 となる。
2. 12 個のピースを作り終えたら、関数 Trial 内において、ピースを詰める関数 Place, 外す関数 Remove, そして、ピースを詰めることが可能であるか否かを判定する関数 Fit を立方体が完成するまで繰り返し実行する。関数 Fit においては、以下のように、空間の状態を大域変数として参照するため、入力列の長さに 4〜5 倍にもなる大きな差ができる。

   ```
   int Fit (i, j) int i, j;
       {
       int k;
       for ( k = 0; k <= piecemax[i]; k++ )
           if ( p[i][k] ) if ( puzzl[j+k] ) return (false);
       return (true);
       };
   ```

   また、空間全体の状態を表す大域変数 p は、要素数が 6656 にもなる大きな配列である。よって、それを利用する関数 Fit や Trial にも、最大長が 40〜100 になる入力列が登録される。しかし、基本的にトライアンドエラーの繰り返しであるため、再利用の頻度は高いと考えられる。実際、関数 Fit

（付録図 A.13 右側）や、同様のループを持つ Place（付録図 A.14）、また、それらを再帰的に呼び出す関数 Trial 内のループ（付録図 A.13 左側）において、事前実行による入力列が再利用された。一つ一つの再利用も、最大 1000 ステップを越える処理時間が短縮される、効果の高いものであった。しかし、予想に反し、MSP のみによる実行では、関数 Fit, Place, Trial においても再利用は観測されなかった。この理由は、登録される再利用表の少なさにあると考えられる。一つ一つの入力列が、最大 40〜100 になる長さを持つため、登録される入力列の個数は、多くとも 20 種類程度とかなり少なく、また、一つ一つの入力列が保持される期間も短いため、再利用できる機会を逃す場合もあると考えられる。また、関数 Remove では、MSP のみの場合でも、事前実行による場合でも、再利用は観測されなかった。この関数も、Fit や Place と同様、大域変数 p を参照するループを持つ。しかし、Remove の実行回数は、Fit や Place に比べて少ない。そのため、次に Remove が実行される前に、蓄えられていた再利用表が消去されてしまい、再利用は発生しなかったと考えられる。

第 5 章　まとめ

本論文では、区間再利用機構における入力列の構造を可視化し、入力列単位で再利用状況を観察する手法を提案した。それを基に、Stanford-integer の再利用状況を観察した結果、プログラムの性質により、再利用されやすい入力列のパターンや、その頻度に大きな違いがあることがわかった。今までの研究の中で、再利用の効果が高いとされていたプログラムも、実際に再利用される様子はそれぞれ大きく異なっていた。大まかに、次の 3 通りに分けられる。一回の再利用により 1000 ステップ以上の高速化がなされた命令区間、一つ一つの効果は小さいもののほぼ全ての入力列が再利用されていた命令区間、多数の内部命令を持ち、再利用された際には、内部命令全ての実行が省略できる命令区間の 3 種類である。再利用がなされない命令区間についても、注目すべき部分がある。ソートや行列積、Puzzle など、最大 100 以上に及ぶ長さの大域変数を参照する命令区間は、RB エントリを大きく消費する。そのため、登録される入力列の個数も少なく、一つ一つの入力列が記憶される期間も自然と短いものとなる。再利用が起こる前に消去されてしまう場合も考えられる。Perm においても、実

行数の多い関数 Swap に再利用表が割り当てられず、再利用の機会を失う場合もある。また、Queens, Quick にも、長さ自体は 40 以下と小さいものの、入力列ごとに、2〜4 倍近い極端な変化が現れていた。このような命令区間では、入力値の予測は難しいと考えられる。

謝辞

本研究の機械を与えてくださった、富田眞治教授に深く感謝の意を表します。

また、本研究に関して、研究の方向性について適切なご指導、そして、数々のご助言を賜った中島康彦助教授、森眞一郎助教授、五島正裕助手に深く感謝いたします。

さらに、日ごろ暖かく御鞭撻下さった京都大学工学部情報学科富田研究室の諸兄に心より感謝いたします。

参考文献

[1] Tollis, I., Battista, G. D. and Battista, G. D.: *Graph Drawing: Algorithms for Geometric Representations of Graphs*, Prentice Hall, U.S. (1998).

[2] Yee, K.-P., Dhamija, D. F. R. and Hearst, M.: Animated Exploration of Dynamic Graphs with Radial Layout, *2001 INFORMATION VISUALIZATON*, pp. 43–50 (2001).

[3] Jankun-Kelly, T. and Ma, K.-L.: MoireGraphs: Radial Focus+Context Visualization and Interaction for Graphs with Visual Nodes, *InfoVis. 2003*, pp. 59–66 (2003).

[4] 中島康彦, 緒方勝也, 正西申吾, 五島正裕, 森眞一郎, 北村俊明, 富田眞治: 関数値再利用および並列事前実行による高速化技術, 情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム, HPS5, pp. 1–12 (2002).

[5] van Ham, F., van de Wetering, H. and van Wijk, J. J.: Visualization of State Transition Graphs, *2001 INFORMATION VISUALIZATON*, pp. 59–66 (2001).

[6] Plaisant, C., Grosjean, J. and Bederson, B. B.: SpaceTree: Supporting Exploration in Large Node Link Tree, Design Evolution and Expirical Evaluaton, *InfoVis. 2002*, pp. 57–64 (2002).

[7] Nguyen, Q. V. and Huang, M. L.: A Space Optimized Tree Visualization, *InfoVis. 2002*, pp. 85–92 (2002).

[8] 中島康彦, 津邑公暁, 五島正裕, 森眞一郎, 富田眞治: 動的命令解析に基づく多重再利用および並列事前実行, 情報処理学会論文誌：コンピューティングシステム, Vol. 44–No. SIG 10(ACS 2), pp. 1–16 (2003).

[9] 津邑公暁, 中島康彦, 五島正裕, 森眞一郎, 富田眞治: 並列事前実行機構における主記憶値テストの高速化, 情報処理学会論文誌：コンピューティングシステム, Vol. 45–No. SIG 1(ACS 4), pp. 1–16 (2004).

[10] 津邑公暁, 笠原寛壽, 清水雄歩, 中島康彦, 五島正裕, 森眞一郎, 富田眞治: 大容量 3 値 CAM を用いた並列事前実行機構の効率的実現, 先進的計算基盤システムシンポジウム SACSIS2004, pp. 251–259 (2004).

[11] 清水雄歩, 笠原寛壽, 中島康彦, 五島正裕, 森眞一郎, 富田眞治: 汎用 CAM を用いた区間再利用プロセッサシミュレータの高速化, 平成 16 年度情報処理学会関西支部 支部大会 講演論文集, pp. 175–178 (2004).

[12] 笠原寛壽, 清水雄歩, 津邑公暁, 中島康彦, 五島正裕, 森眞一郎, 富田眞治: 2 次キャッシュを用いた再利用および並列事前実行機構における高速化手法, 情報処理学会研究報告　2004-ARC-149(HOKKE-2004) (2004).

[13] 柴山守: X11 による画像処理基礎プログラミング―ビットマップからビデオ動画像まで―, 技術評論社, Japan (1994).

付録

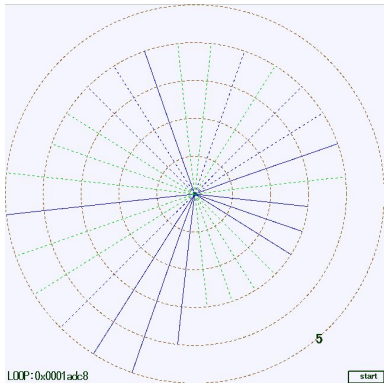

図 A.1: (Queens) 初期入力を行うループの事前実行

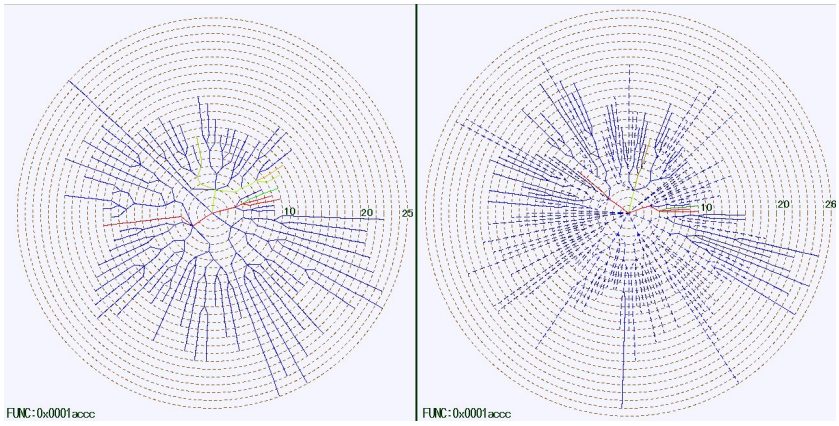

図 A.2: (Queens) 関数 Try における再利用表

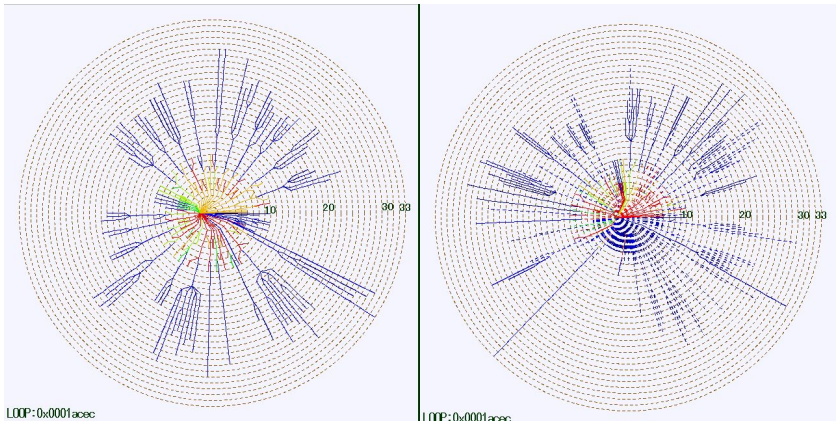

図 A.3: (Queens) 関数 Try を再帰的に実行させるループ

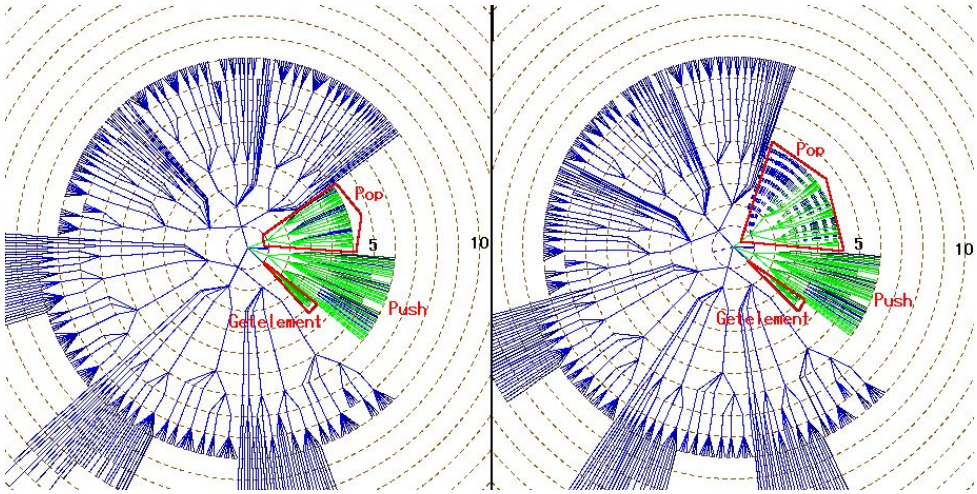

図 A.4: (Towers) 全体の再利用表（左側：MSP のみ, 右側：事前実行を使用）

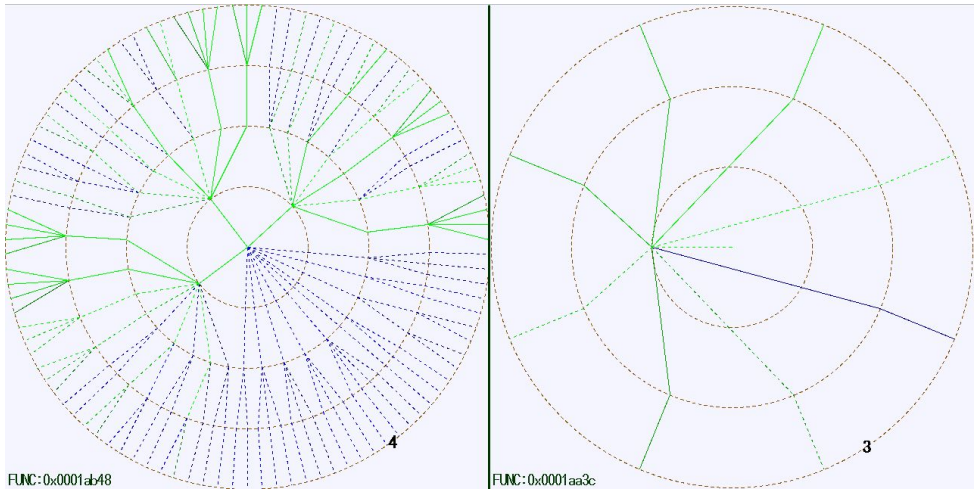

図 A.5: （Towers) 再利用の行われた関数（左側：Pop, 右側：Getelement）

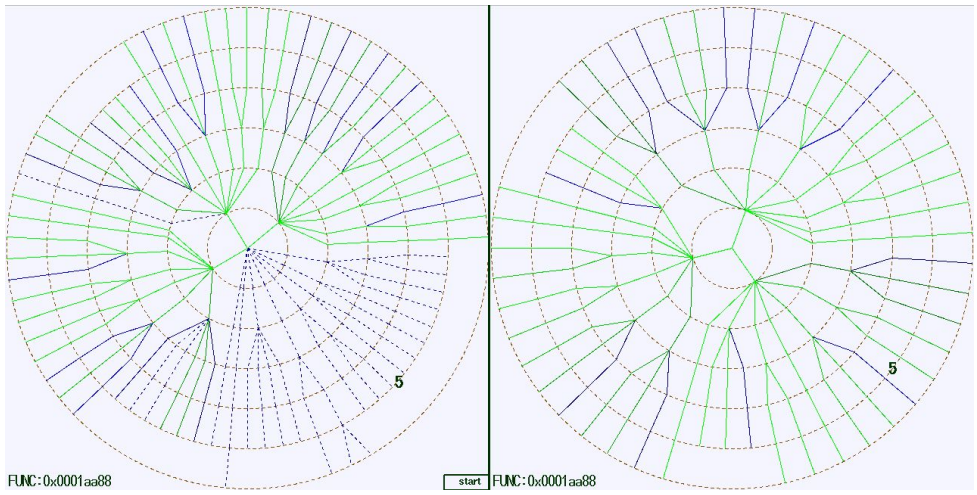

図 A.6: (Towers) 関数 Push の変化（事前実行を使用した場合）

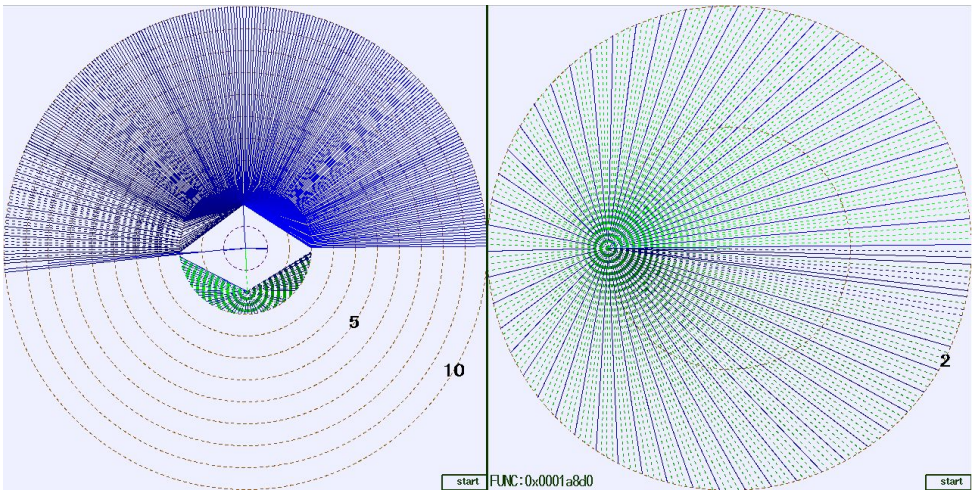

図 A.7: (Bubble) 左図上部：bIntarr 内のループ、左図下部と右図：関数 Rand

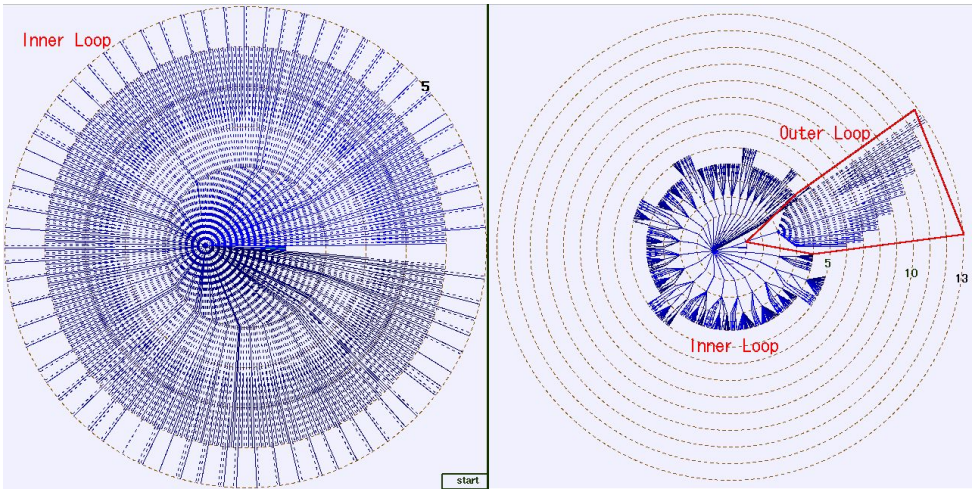

図 A.8: (Bubble) ソート時、全体の変化

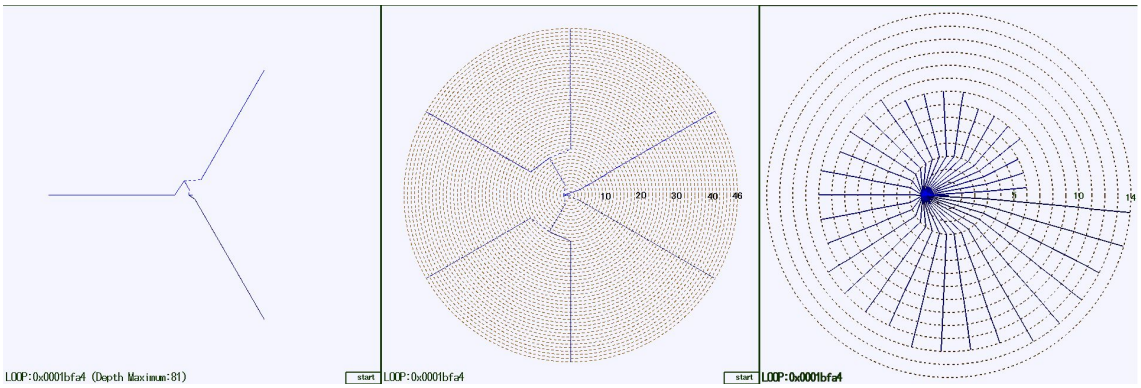

図 A.9: (Bubble) ソート時、外側のループの変化

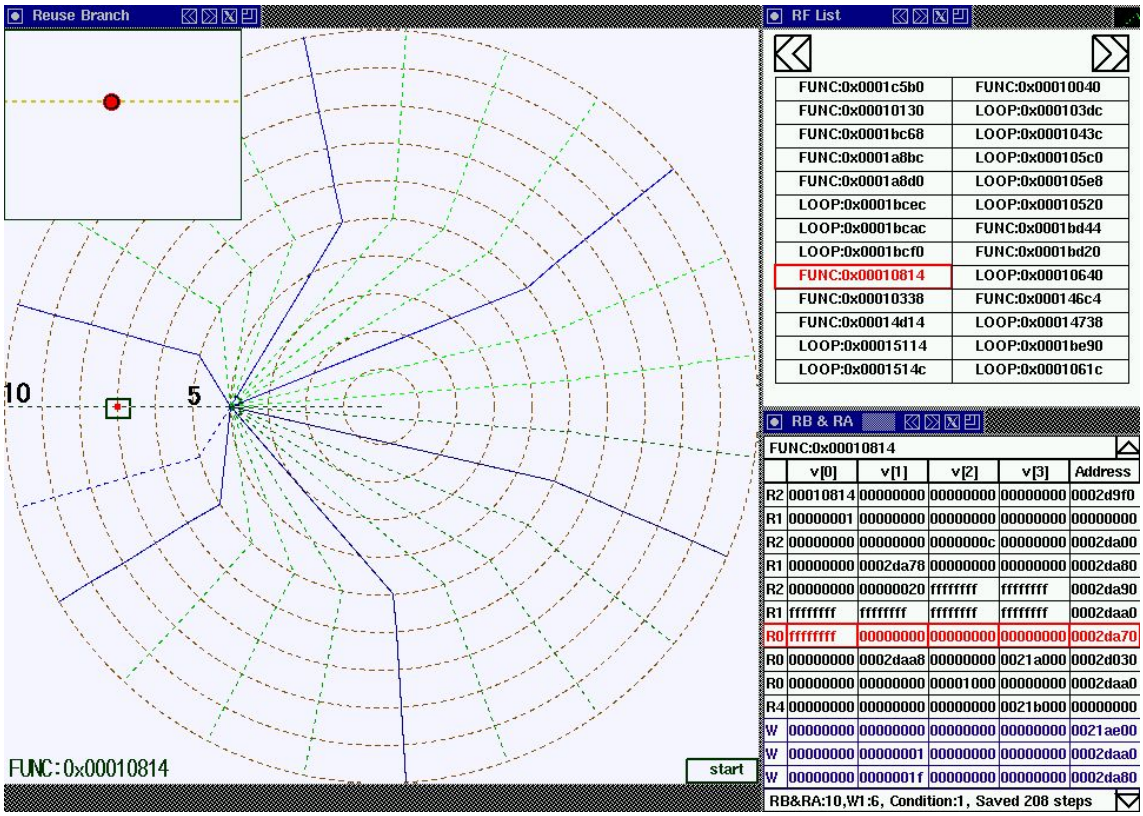

図 A.10: (Trees) 関数 malloc の再利用

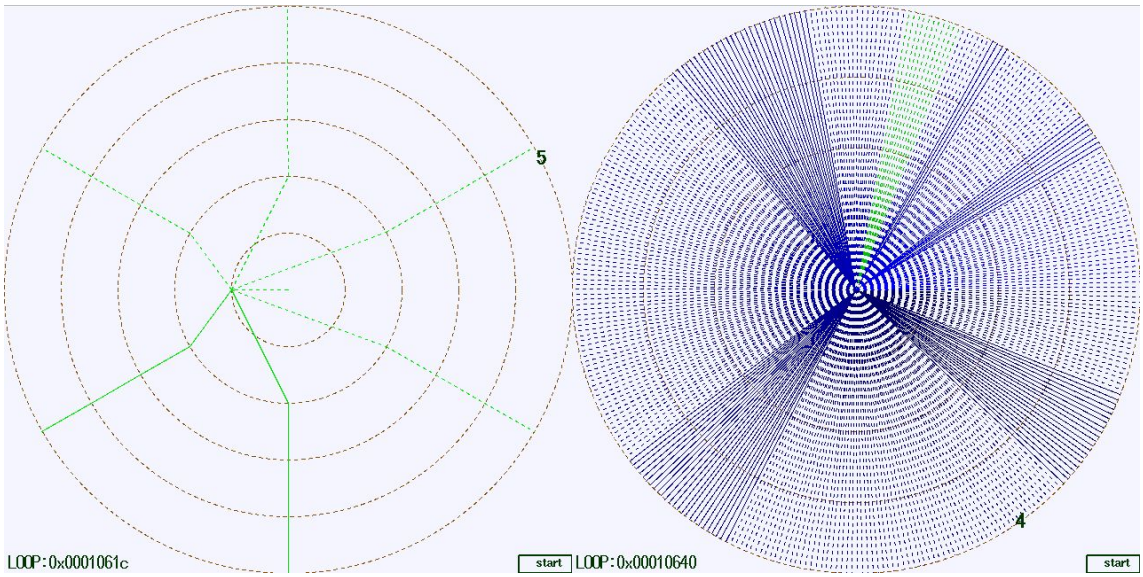

図 A.11: (Trees)malloc 内部で顕著に再利用が現れる命令区間

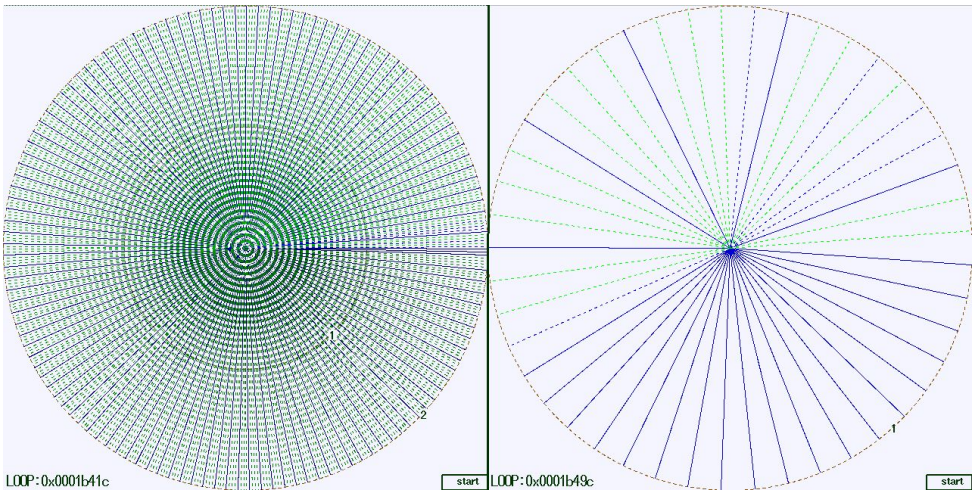

図 A.12: (Puzzle) 初期入力における再利用

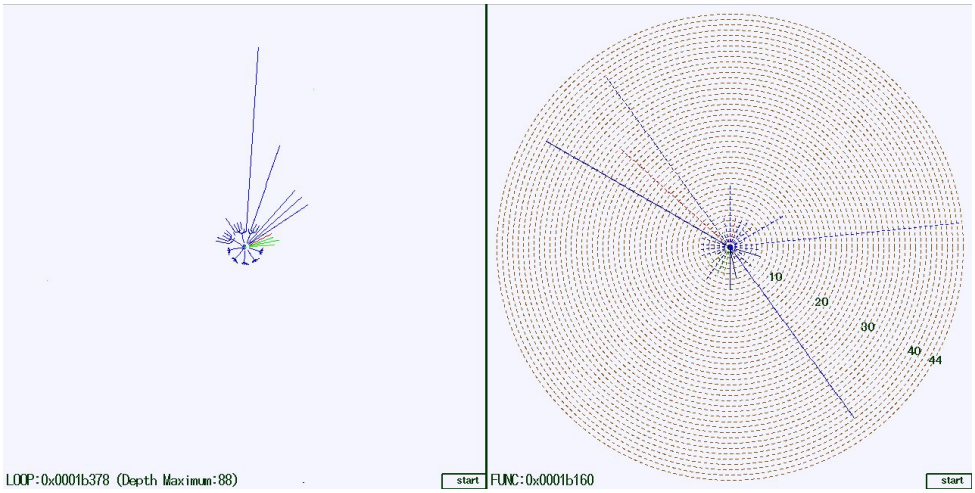

図 A.13: (Puzzle) 左：Trial における再利用, 右：Fit における再利用

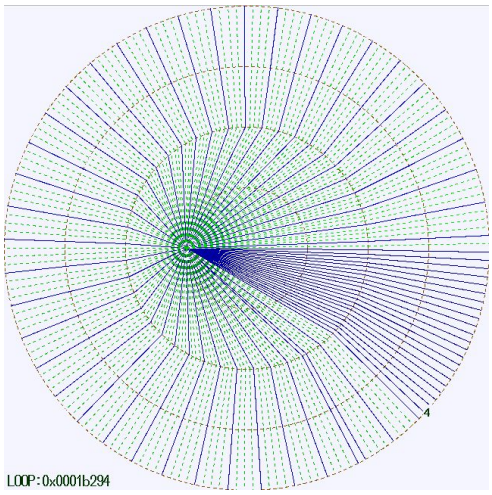

図 A.14: (Puzzle)Place 内における再利用