言語処理系におけるごみ集めと
コード最適化に関する研究

鈴木　貢

電気通信大学大学院電気通信学研究科
博士（工学）の学位申請論文

2007年3月

言語処理系におけるごみ集めと
コード最適化に関する研究

博士論文審査委員会

主査	教　授	野下　浩平
委員	教　授	尾内　理紀夫
委員	教　授	岩田　茂樹
委員	教　授	岩崎　英哉
委員	助教授	小林　聡
委員	助教授	中山　泰一
委員	名誉教授	渡邊　坦

Studies on Garbage Collection and Code Optimization in Language Processors

Mitsugu Suzuki

Abstract

Two research issues related to programming-language implementations are discussed in this thesis. The first is the Garbage Collector (GC). A GC manages memory locations dynamically as a run-time system, and is indispensable to implementing symbol-processing or object-oriented languages. However, the GC is a performance bottleneck in language processors. The second issue is compiler optimization for Single Instruction Multiple Data-stream (SIMD) instruction sets. We treat SIMD instruction sets that are based on partitioned-register vector operation, and are designed as extended sets for conventional instruction in this thesis. The SIMD instruction sets have been developed to provide low-cost solutions to the high-performance processing of media data, and have become popular. Conventional optimization techniques can certainly help us to partially exploit them, but these are insufficient and new ones need to be developed. Compiler optimization for SIMD instruction sets are currently in the study stage, and programmers must use assembly language or intrinsic routines to exploit them.

Chapter 1 describes the backgrounds to the two issues.

The GC is at first formalized in Chapter 2. Second, three strategies for GC, i.e., reference counting, copying, and mark-and-sweep (M&S) are introduced. Third, two problems that have not yet been solved in M&S GCs are presented. Finally, previous work is introduced and analyzed.

Chapter 3 presents ways of reducing the processing time for M&S GCs. Conventional M&S GC consumes time that is proportional to S, where S is the size of a heap. A sliding compactor means an M&S GC that compacts the memory area occupied by in-use objects by sliding them, and it preserves correspondences between the generated order and location order of the objects. We will call this property "Generated Orders Preserved (GOP)" after this. A sliding method of compaction is presented, which consumes time that is theoretically proportional to $AlogA$ or actually A , where A is the number of words of the in-use objects. When A is less than one-tenth of S, the GC is sufficiently faster than a conventional sliding compactor.

Chapter 4 presents a method of adopting a GC strategy called "Generation Scavenging (GS)" into the sliding compactor. The GS has been conventionally combined with copying GCs. The method we presented exploits GOP for generation management. It needs a shorter run-time and has smaller space overhead than the conventional GS. The GC processing time is also reduced.

Chapter 5 first introduces the kinds and functions of SIMD instructions. Second, problems related to automatically generating the SIMD instructions are presented. Third, previous work is introduced and analyzed. Finally, a compiler infrastructure, on which the implementation discussed

in this thesis is based, is described.

Chapter 6 discusses a method of matching and extracting operation sequences that are equivalent to each SIMD instruction from program codes written with normal language specifications. We extended peephole optimization and adopted it in the method.

Chapter 7 discusses the compatibility between integral promotion rules and minimizing the data size for processing. Integral promotion rules mean that integer data that are smaller in size than integral data are zero-extended or sign-extended before they are used in a sequence of operations. The analysis of program code is presented, which enables the compiler to select smaller data than integral data for processing with the same results if the integral promotion rules can strictly be observed.

Chapter 8 claims that existing benchmark or vector-processing programs are not suitable for testing, evaluating, or analyzing the SIMD optimization by a compiler. Examples extracted from these programs were analyzed and improved so that they could efficiently be processed with SIMD instructions. We found coding styles suitable for SIMD instructions with this methodology, and developed a set of benchmark programs that could analyze the progress of SIMD optimization achieved by compilers. They were sets of programs with several versions that were systematically modified to simplify the application of SIMD instructions.

Chapter 9 concludes the results of this study. First, we developed time and space efficient GC. Even with limited storage, GCs, which are indispensable in modern programming languages, can efficiently be implemented in a computer. Second, we implemented and evaluated two compiler optimization techniques for SIMD instructions with a dedicated method of evaluation, and these contributed to speeding-up execution. With this, computers with different instruction sets can share the same media-processing programs.

言語処理系におけるごみ集めと コード最適化に関する研究

鈴木 貢

論文要旨

本研究では，言語処理系の実装に関連する二つの事項を取り扱う．本論文の第 1 部（第 2 章〜第 4 章）では，実行時系における記憶場所の動的な管理である「ごみ集め」に関して議論する．ごみ集めは，記号処理言語やオブジェクト指向型言語の実装には不可欠なしくみで，その性能が全体の性能を左右する．そして，第 2 部（第 5 章〜第 8 章）では，SIMD（Single Instruction Multiple Data-stream）命令の一種のレジスタ分割型ショートベクタ命令セット（以下 SIMD 命令と略す）に注目し，翻訳系による SIMD 命令向け最適化に関して議論する．この命令セットは，メディアデータ処理の高速化の安価な手法として考案され定着してきた．これには，従来のコード最適化技法を応用可能な事項もあるが，新しく開発する必要がある事項も多い．現状では，コンパイラによる SIMD 命令の自動生成は研究段階であるので，アセンブリ言語等を用いないと SIMD 命令の満足がいく活用ができない．

第 1 章では，二つの事項を研究するに至る背景について議論する．

第 2 章では，先ず，第 1 部で取り扱うごみ集め問題を定式化した．次に，参照計数法，複写法，印掃法というごみ集めの三つの戦略を示した．そして，従来の印掃法のごみ集めでは解決されていなかった二つの問題を指摘し，関連する既存の成果を紹介し分析した．

第 3 章で議論している第一の問題は，印掃法ごみ集めの実行時間を小さくすることである．従来の印掃法ごみ集めは，記憶領域のサイズを S として，S に比例する時間が必要であった．これに対し，使用中の記憶の量を A として，理論上の計算量は $AlogA$，実用上は A に比例する実行時間で済む印掃法のごみ集めで，使用領域を圧縮するもの（以下印掃式圧縮型と略す）を提案した．これにより，A が S の 1 割以下という一般的な状況の下では，ごみ集めが高速化された．

第 4 章で議論している第二の問題は,「世代別ごみ集め」と呼ばれるごみ集めの手間を省く戦略を，印掃法の圧縮型ごみ集めに適用することである．世代別ごみ集めは，通常は，複写法のごみ集めとの組み合わせで実現される．ここでは，印掃法の圧縮型ごみ集めの生成順序保存という特徴を生かした方法を提案し，小さなオーバヘッドで世代別ごみ集めを実装する方法を示した．これにより，さらにごみ集めが高速化された．

第 5 章では，先ず，第 2 部で取り扱う SIMD 命令の動作や種類を説明する．次に，コンパイラによる SIMD 命令の自動生成に関する問題点を指摘し，関連する既存の成果を紹介し分析する．そして，本研究での実装に関連する成果を紹介する．

第 6 章で議論している第一の問題は，SIMD 命令向けの言語仕様の拡張を使用していない（つまり通常の言語規格の範囲内で記述された）プログラムから，SIMD 命令の振る舞いに相当するオペレーションを抽出する方法である．このために，覗き穴最適化を応用した方法による最適化技法を提案した．

第 7 章で議論している第二の問題は，高級言語の汎整数拡張が，効果的な SIMD 命令の適用を阻害

していることである．汎整数拡張とは，汎整数型（通常は計算機が効率よく処理できる整数型）より小さいサイズのデータは，演算に使用される前に汎整数型への符号拡張やゼロ拡張が施されるという仕様である．これに対して，なるべく小さいデータサイズによる演算で，汎整数拡張を行う場合と同じ結果を得るためのプログラム解析法を提案した．

第 8 章で議論している第三の問題は，実在する多くのベクタ処理向きのプログラムやベンチマークプログラムが，コンパイラによる SIMD 命令活用のベンチマークに適していないことである．そこで，これらの中の例題を分析し，SIMD 命令を効果的に適用できるように変形し，SIMD 命令向きのコーディングを示した．さらに，コンパイラの SIMD 命令の活用度を調べるためのベンチマークとして，各々の例題に対して，SIMD 命令の適用を容易にする種々の変形を組織的に施したプログラム集で，複数の変形版から成るものを用意した．

最後に第 9 章では，本研究で得られた成果を総括する．成果の一つは，記憶領域の使用効率が高く，実行時間も小さいごみ集めを提案したことである．それにより，記憶領域が乏しい計算機でも，近代的な言語が必須とするごみ集めを効率よく実現できる．もう一つは，SIMD 命令向けの最適化技術を提案し，実装し，評価し，実行性能の向上を示したことである．これにより，命令セットが異なる計算機間でも，同じメディアデータ処理プログラムを共有できるようになる．

目 次

図 目 次

表 目 次

第1章　序論

本論文では，プログラミング言語処理系における，実行時系（runtime system）の自動記憶管理系の構成方法と，コンパイラ（compiler，翻訳系）の最適化（optimization）の方法について議論する．これ以降,「計算機システム」という語をソフトウエアとハードウエアから成る計算を行うシステムの意味で用い,「計算機」という語を計算機システムのハードウエアのみに注目する場合に用いることとする．また，プログラミング言語処理系のことを単に言語処理系と呼ぶことにする．実行時系とは，言語処理系の中でもプログラムの実行時に必要とされる部分のことを意味し，例えば標準ライブラリルーチンや記憶管理系を例として挙げることができる．自動記憶管理系とは，第 2 章でその定式化や方式の詳細説明を行うが，要約すれば記憶領域を動的に確保できるシステムにおいてその明示的な解放を必要としないようにするための実行時系のことであり,「ごみ集め」（Garbage Collection, GC）とも呼ばれる．本論文では,「印掃法」に基づくごみ集めで，使用中領域の圧縮を行うものに注目し，その改良法を 2 つ提案する．この方式は第 2 章で議論するように，他の方式に比べて記憶領域の使用効率が良い．

コンパイラの最適化には，その対象命令セットアーキテクチャへの依存の有無に依らず，数多の技法が公表されている [58]．本論文では，レジスタ分割式ショートベクタ処理 SIMD（Single Instruction Multiple Data-stream）命令セット（以下 SIMD 命令セット と略す）に注目し，SIMD 命令セット向けの最適化に寄与する 3 つの成果を示す．

1.1　ソフトウエアの生産性向上とごみ集め

オブジェクト指向は，構造化プログラミング [18] と並んでソフトウエア工学が獲得した有益な手法であるといえる．ソフトウエアを大勢のプログラマによって開発したり，後の拡張性を考えて設計する場合において，オブジェクト指向を無視することは，あり得なくなっている．一方で，Common Lisp や Scheme に代表される記号処理言語は，記号・知識処理や大規模ソフトウエアシステムの補助言語として，多くのフィールドで使われている．例えば高機能エディタ Emacs の舞台裏では，Emacs Lisp と呼ばれる Lisp の方言が働いて，高度な機能やエディタのカスタマイズを実現している．

このようにオブジェクト指向型言語や記号処理言語は，ソフトウエアの生産性向上に大いに貢献しているが，記憶管理の側から見れば，これらの言語が必ず備えている「自動記憶管理」，あるいは「ごみ集め」が生産性向上の大きな鍵を握っていると考える．

C++はオブジェクト指向型言語であると言われているが，その元になっている C 言語よりはよいとしても，それほどソフトウエアの生産性は高くない．その理由は，C++に標準ではごみ集めは付いておらず，解体子（destructor）を明示的に呼び出さねばならないことである．アプリケーションの不具合に関して，しばしば「メモリリーク」という言葉を聞くが，これは解体子の呼び出し損ないに他ならない．そのために最近では保守的ごみ集め（conservative GC）[11] を実行時系に組み込ん

で，ソフトウエアの開発を行うのが定石になっている．また，あるソフトウエアベンダーは，それまで C++を開発言語として推奨していたのを，ごみ集めが付加されている C#[31] に切り替えた．

組み込み計算機クラスのものでもプロセッサの能力が向上したため，生産性の高さから Java[7] 等のオブジェクト指向言語が，そのソフトウエア開発に使われることが多くなっている．また，パーソナルコンピュータで稼動しているアプリケーションを，組み込み用の計算機で稼動させたいという要求もあり，オブジェクト指向はごみ集めと共にソフトウエアの生産性向上の道具として，多くの計算機システムに浸透しつつある．

この例を身近に見つけるならば，最近の携帯電話がその典型として挙げられる．ディジタル通信機能を応用して，地上局に接続されたサーバからプログラムをダウンロードし実行して，ゲーム等を楽しめるようになっているが，多くはオブジェクト指向言語 Java のアプレットになっており，その稼動には，やはりごみ集めが必要である．

第 2 章で詳述するが，ごみ集めは，ユーザが使い捨てする記憶領域を再利用するための仕組みである．使った記憶領域を捨てている，つまりプログラムの実行環境から到達できないようにしている限りは，ユーザにとっては無限に記憶領域が与えられているように見える．記憶領域は，管理のための領域的なオーバヘッドを無視できるならば，必要な時に必要なだけを動的に確保し，不要になったら返却する，もしくは，捨てるようにすることが，記憶領域の利用効率を最も高めると考えられ，このスタイルは特に主記憶が少ない計算機，例えば組み込み計算機にとって，相応しいと考えられる．(FORTRAN の COMMON 変数のように，プログラマが記憶領域の多重利用を明示的に指定するような場合を除く.）しかし，そういったところにまでごみ集めのメリットがもたらされるには，使用中の記憶領域の総量との比較で，主記憶量が少ない場合でも満足の行く性能で動作し，多い場合にはそれがごみ集めの手間の削減につながるようなごみ集め手法が望ましい．

本論文の第 1 部では，記憶領域の利用効率が高いごみ集め方式について議論する．本研究では，印掃法（mark and sweep method）に基づく，滑り圧縮（sliding compaction）型のごみ集めの問題点の改良を行った．

印掃式滑り圧縮のごみ集めは，記憶領域の全てを使用中のデータ（換言すればごみ以外のデータ）が占めることを許しているので，記憶の利用効率が高いことが利点であるが，ごみ集めを行うのに，記憶領域全体の走査を必要としているので，その手間は記憶領域のサイズに比例する．これでは使用中のデータの占める割合が低くなったときに問題となるので，本研究では（実用的には）使用中のデータの量に比例する手間で済ませるための改良を提案し，さらのその手間を世代別管理と呼ばれる管理戦略を用いて少なくする手法を提案する．

1.2　メディアデータ処理

パーソナルコンピュータやワークステーションは，事務処理，科学技術計算，プログラム開発を含む記号処理といった用途以外に，信号処理やメディアデータ処理といった用途にも広がりを見せている．これ以降，メディアデータとは，音声や静止画像，動画像といったデータのことを指すものとする．メディアデータ処理とは，そのようなデータの圧縮・伸張や識別，認識といった計算を意味する．特にパーソナルコンピュータでは，音楽情報や動画情報のディジタル化と共に，その用途の主軸がメディアデータ処理に移りつつある．この潮流は，パーソナルコンピュータに限らず，組み込み計算機システムの類にまで浸透しつつある．

従来はその類の用途に対しては，専用の DSP（Digital Signal Processor）や画像データ圧縮・伸張用 LSI を搭載した，一般にアクセラレータと呼ばれる類の拡張ボードを計算機に搭載して，計算能力を高めるという手法で対応していた．

最近では，携帯電話のようなものでも動画データや圧縮された音楽データを取り扱うようになっており，新しい端末が発売される度に，ますます高機能になっている．このように，メディアデータ処理でも安価で高性能な方法が希求されている．

1.2.1 バーストアクセス性能の向上

メディアデータ処理について考える前に，計算機の主記憶のアクセス性能の推移について考えてみたい．メディアデータは，例えば DVD 品質の圧縮していない動画像では，毎秒約 66 メガバイトのデータ流量になる．このようなデータの処理を行うには，処理方法によっても異なるが，プロセッサと主記憶の間でその流量の数倍のデータのやり取りを行う必要がある．大抵の場合は，メディアデータ処理で扱われるデータの量は，キャッシュメモリの有効範囲を越えており，その点では次に説明するスーパーコンピュータが扱うデータと同じ傾向であるといえる．

古典的な計算機の教科書では，主記憶のアクセス性能は連続アクセスであろうとランダムアクセスであろうと，同じであるとし，キャッシュの当たり外れによるアクセス時間が議論の中心であった．しかし近年の計算機では，以下に述べる経緯でスーパーコンピュータの技術がデスクトップ計算機にまで技術移転してきたため，主記憶のアクセス性能を，ランダムアクセス性能とバーストアクセス（連続アクセス）性能に区別して考えなくてはならなくなった．

スーパーコンピュータに代表される科学技術計算用機の特徴は，倍精度浮動小数点演算器が高性能であることと，主記憶/プロセッサ間のバーストアクセス性能が桁違いに高いことにあると言っても過言ではない．スーパーコンピュータの多くがその技術的な基盤としているベクタ処理では，ベクタレジスタと呼ばれる 64 から 4096 程度の長さの倍精度浮動小数点や整数レジスタの配列に，なるべく短い時間で主記憶からデータを出し入れできる性能が要求された．スーパーコンピュータが文字通り「スーパー」であった頃は，高速 SRAM（Static RAM）や高速 DRAM（Dynamic RAM）を，さらにインタリーブやマルチバンク化といった手法を用いて束ねたり，メモリシステムもパイプラインの一部に取り込むことによって，バーストアクセス性能を向上させていた．計算機の論理素子に高価な ECL（Emitter Coupled Logic）を用いていたこともあるが，このような物量的な手法でバーストアクセス性能を向上させる回路もスーパーコンピュータのコストアップの一因となっていた．

電子デバイス技術の発展に牽引されて，デスクトップクラスの計算機でもマイクロプロセッサの内部動作クロックが向上し，スーパーコンピュータで用いられているような高性能な浮動小数点演算器を搭載できるようになった．すると，主記憶へのアクセス性能が計算機の性能の足かせになってきた．そこで，1〜2 クロックでアクセス可能なキャッシュメモリをプロセッサと同じ LSI 上に集積して，平均アクセス時間を短縮するようになった．これは，スーパーコンピュータのベクタレジスタのアクセス性能に匹敵する．ここでキャッシュメモリは，通常は「語」単位ではなく，その 2 のべき乗倍のライン単位（通常は 32 から 256 バイト）で管理され，キャッシュメモリのラインへの出し入れがバーストアクセスになるようにしているということに注意されたい．

スーパスカラ技術 [37] がデスクトップクラスの計算機にも波及して命令レベル並列性（Instruction Level Parallelism，ILP）が，1 クロックあたり 2〜3 命令程度であることが当たり前になり，プロセッサ内部の PLL（Phase Locked Loop）を用いて，外部から与えるクロックを逓倍（基本クロック周波

数の整数倍のクロックを得る技術）して，プロセッサのクロック信号とするようになると，プロセッサ外部のメモリ素子（つまり主記憶）のアクセス速度と，プロセッサ内のキャッシュメモリを含めたプロセッサの性能の間の乖離がますます大きくなった．しかし，スーパーコンピュータで用いられてきたような物量的な手法を採ることは，コストあるいは空調等の運用面での理由で許されない．このような状況のブレークスルーとなったのは，非同期式 DRAM からクロック同期式 DRAM への転換である．

FPM-DRAM（First Page Mode D-RAM）や EDO-DRAM（Extended Data Out DRAM）のような従来の非同期式 D-RAM は，RAS（Row Address Strobe）等の各々の制御信号のエッジに同期したり，パルスに対応して素子が動作するようになっている．それらのタイミングをコントロールする信号は，プロセッサの外で作られるので，ある水準以上の高速化が困難であった.（この種の DRAM であっても，ファーストページモード等によって，バーストアクセス性能をランダムアクセス性能に比べて高めることができる.）

一方で，クロック同期式 DRAM である RDRAM（Rambus DRAM）や SDRAM（Synchronus DRAM）は，クロックのエッジに同期して素子が動作するようになっており，バーストアクセス性能がクロックレートに比例して高まる．そして．内部の動作がパイプライン化しているのでクロックレートを高めやすい.（市販の SDRAM 等の CL2，CL3 等の表示は，パイプラインの段数に対応している.）しかも，クロックの立ち上がりと立下りの両方でデータ転送を行う DDR-SDRAM（Double Data Rate SDRAM）や，DDR2-SDRAM 等の改良が行われており，数年で 2 倍の割合でバーストアクセス性能向上を達成している．

しかしながら，クロックレートの向上にも拘らず，ランダムアクセス性能はほとんど向上していない．その理由は，ランダムアクセス性能はメモリのレスポンス時間，つまりメモリにコマンドを発行してから結果が返って来るまでの時間に対応しているが，クロックレートの向上と共にパイプラインの段数も増加しているからである．これはプロセッサの「クロックレート増加 → パイプライン段数の増加 → 分岐ペナルティーの増加」という図式とよく似ている．

1.2.2　メディアデータ処理のための省資源向きの解—SIMD 命令

キャッシュメモリも含めたメモリ性能については，クロック同期式 DRAM の登場により，スーパーコンピュータ並の性能になった．また LSI の集積度の向上によって高速な浮動小数点演算回路が汎用プロセッサに内蔵されるようになった．そのために，科学技術計算用の計算機としては，デスクトップクラスでもスーパーコンピュータ並かそれ以上の性能を享受できるようになった．しかし，大部分のパーソナルコンピュータのユーザは，科学技術計算のために計算機を購入しているのではないので，このような性能向上のメリットをあまり享受できない．

メモリのバーストアクセス性能の向上を振り向けることができる計算対象で，一般のユーザが恩恵を受けられるものとして，この節の冒頭で述べた，DSP や専用ハードウエアで行われてきた信号処理や音声や画像といったメディアデータの処理が挙げられる．その発端はアナログモデムの DSP を，メインプロセッサの余剰計算能力で置き換えて，モデムのコストを下げようとしたことに始まる．この種のデータは，比較的小さな主記憶の DSP や専用ハードウエアで処理できる程度のメモリ参照連続性と処理規模を持つ処理対象であり，さらにデータのアクセスを連続した番地にするアルゴリズム上の工夫が可能な場合が多いので，主記憶のバーストアクセス性能が活かせると考えられる．

しかし，先に述べたようなバーストアクセス性能は，倍精度浮動小数点データのデータ幅（64 ビッ

ト）に合わせて設定されているということを考慮しなければならない．一方で，各種信号処理や音声・画像データの処理では，大部分が 8〜16 ビットの整数演算で十分である．つまり，演算の複雑さにも依るが，科学技術計算に要求されたバーストアクセス性能を使い切ることができない．

多くの場合，信号処理やメディアデータの処理には実時間性が要求される．近年はパーソナルコンピュータの OS でさえもプリエンプティブなマルチタスキングで動作するようになったので，規定量の処理を完了したら別のアプリケーションに CPU 時間を譲ることが可能であることから，信号処理等に対する性能が高いことを，計算機全体の性能向上として享受できる．

このような処理内容に対する処理能力の向上を，なるべく低いコスト，かつ，既存の命令セットアーキテクチャを変更せずに，追加だけで達成するための手段として提案され一般化したのが，レジスタ分割式ショートベクタ処理 SIMD（Single Instruction Multiple Data-stream）命令セットで，既存の命令セットアーキテクチャに対する拡張の形で実装されるようになった．以下，この種の命令を単に「SIMD 命令」あるいは「SIMD 命令セット」ということにする．このような SIMD 命令セットとして，1996 年ごろに，Intel 社の IA-32 への MMX 拡張命令セットや，Hewlett-Packard 社の PA-RISC 2.0 への MAX 拡張命令といったものが相次いで発表された．

この種の命令セットは，浮動小数点演算器を構成しているユニットを流用して，その制御論理回路に改良を加えることで実現している．このような改良は，DSP 機能や専用のハードウエアを搭載するのに比べて，比較的安価に実現可能である．そのために SIMD 命令は，並列化による処理の高速化を，低コストかつ省資源に達成できる．

初期の SIMD 命令セットは整数データしか扱えなかったが，新しいデータ圧縮方式等が提案され実用化したり，多くはゲームソフトウエアであるがユーザが使うアプリケーションの広がりと共に，例えば IA-32 では MMX から SSE や SSE2 や SSE3 といったように浮動小数点演算の並列実行ができるように改良・拡張され，例えば IIR（Infinite Impulse Response）フィルタ等を高速に実現できるようになった．さらにデータアクセスに関するプログラミング上の工夫や，記憶素子の改良に伴うバーストアクセス性能の向上も相俟って，最近のパーソナルコンピュータでは，音声の圧縮を実時間の 1/20 程度の時間で遂行可能であったり，高圧縮な動画像の再生をコマ落ちなしでリアルタイムに行えるといった，10 年前には想像もつかなかったことが可能になった．

また，信号処理やメディア処理に限って言えば，スーパスカラ化することで命令レベル並列性を向上したり，高クロック化といった性能向上方法に比べて，コストや消費電力の点でレジスタ分割式の SIMD 命令セットの実装の方が有利である．そこで，ARM のような組み込み計算機向けプロセッサでも SIMD 命令セットが追加されるようになっている．これは，この方式がメディアデータの処理に対して，上述のように SIMD 命令が低コストかつ省資源な解であるということの裏づけになっていると考えられる．

1.2.3 SIMD 命令セット活用の現状

レジスタ分割による並列処理というアイディアは，同時にコンパイラによる支援が貧弱であるというプログラミング上の問題を提起した．レジスタ分割式というアイディアが製品として登場してから 10 年近く経つが，通常の言語規格で記述されたコードから適切な SIMD 命令を生成できるようになったのは，5.2.4 節で触れるように，一部の商用コンパイラでこの数年のことで，GCC（GNU Compiler Collection）では最新の ver 4.01 からである．

実際の（ソースが公開されている）プログラムにおいて，現在のコンパイラの SIMD 命令の自動的な活用による高速化が可能な部分は必ずしも多いとはいえない．しかし，例えば動画圧縮プログラムにおいては，その実行時間の 3 割程度を占める画像間の比較演算において，SIMD 命令をうまく適用すると，その部位だけで 5 倍以上の高速化が可能であり，全体として 2 割以上の高速化が可能になる．最適化技術の高度化によって，その効果が飽和に近づいている現状では，通常命令関連の最適化だけでは，このような高速化の達成は非常に難しいため，SIMD 命令の活用による高速化の可能性を探ることは，十分に価値がある．また，プロファイリングには現れない程度の実行時間を占める部位についても，コンパイラが SIMD 命令を自動的かつ適切に適用すれば，効果の積み重ねによる高速化を達成できる．

しかし，実際のプログラムを検討していくと，そのままでは SIMD 命令の適用範囲は狭いので，適用範囲を広げるには SIMD 命令向きの工夫を必要とすることがわかった．また，そのように SIMD 命令向きにアルゴリズムやコードを改良しても，現状のコンパイラでは，それを思ったように反映できない．現状では，コンパイラが用意しているイントリンシック ルーチン（intrinsic routines 高級言語から使える，機械語命令を使った組み込みルーチンのこと．イントリンシックと略して呼ばれる場合が多い.）や，アセンブラを使って記述しない限り，SIMD 命令の満足のいく活用ができない．これらは機種依存な手段であるので，ソフトウエア工学的に好ましくない．また，コンパイラによる SIMD 命令を活用するコードの自動生成は，ベクトル化等の既知のコンパイラ最適化技術だけでは達成不可能で，さらに新しい最適化技術の開発が必要である．

本研究で焦点を当てているのは，スーパーコンピュータのベクタ化の派生技術に相当するソースレベルで可能な SIMD 命令向け最適化ではなく，そのような最適化が済んでいるプログラムから如何にして適切な SIMD 命令を生成するかということである．その成果として，適用範囲が広いとはいえないが，SIMD 命令向けに記述されたプログラムから，想定したとおりの SIMD 命令を生成できるような最適化系を実装した．しかし，これは単なるパターンマッチングだけで済む問題ではないことが，研究の途上でわかった．

そのひとつは，第 7 章で詳述する，高級言語の整数演算に関する仕様である「汎整数拡張」（integral promotion rule）を遵守した場合と同じ計算結果をもたらしながら，処理データサイズをなるべく小さくすることである．汎整数拡張とは，多少不正確であるが簡潔に言えば，int 型よりも小さいサイズのデータを参照するときには，データの型に応じて符号拡張なりゼロ拡張を施して int 型に変換し，int 型より小さいサイズのデータに書くときは，対応する下位ビットをそのまま書き出すという規則である．SIMD 命令では一定のサイズのレジスタを，8 ビット，16 ビットといった語長に分割して，その語長での演算を並列処理しているので，処理データサイズを小さくできると，サイズ変換のオーバヘッドを削減でき，しかも並列度が向上して，SIMD 命令のメリットが活かされるようになる．そこで，汎整数拡張との互換性を保ちながら，SIMD 命令による処理向けに最適なデータサイズをプログラムから推論するという問題が浮上する．

また，既存のメディア処理等 SIMD 命令による高速化が期待されるアプリケーションの多くに対して，現在のコンパイラ最適化技術では，そのままでは実行効率が良い SIMD 命令を適用することが不可能か困難である．SIMD 命令の活用による高速化を期待するには，先に述べた記憶装置の特性を生かすためのメモリアクセスの連続化のような既知の改良のほかに，演算データサイズがなるべく小さくなるように定数や演算の順序を適切に選ぶといった，SIMD 命令の特性に依存した工夫を行うべきである．スーパーコンピュータのベクタ命令といった従来型の特殊命令セットに対しては，ベンチマークプログラムにはそれ向きのコーディングの跡がうかがえ，マニュアルはベクタ命令有効活用

のための親切丁寧なドキュメントになっているが，SIMD 命令に関しては，商用のコンパイラにおける，それぞれのコンパイラに依存した工夫が示されているに過ぎない．それらの多くの場合は，イントリンシックの解説である．

この問題に対しては，SIMD 命令向けにアルゴリズムやコーディングが改良されたコードから成るベンチマークプログラム集であって，通常の言語規格の範囲内で記述された（つまり，コンパイラの特別仕様や命令セットアーキテクチャに依存しない）ものを用意することが，解のひとつであると考える．その用途は，そのベンチマークのコードを基準として，コンパイラの設計者は，処理系の改良や調整を行ない，コンパイラのユーザは，作成するコードのお手本としたり，使うコンパイラの SIMD 命令生成に関する特性を知ることにある．

1.3 序論のまとめと本論文の構成

以上で，本論文で議論している事項の背景と具体的な問題を概観してきた．最初に，ソフトウエアの生産性に注目し，効率よく動作する自動記憶管理の必要性を指摘した．さらに，近年の計算機システムの用途におけるメディアデータ処理の重要性に注目し，その効率的な処理という問題を指摘した．

そして，それぞれの事項に対して，次の問題を提起した．

1. 主記憶の大小に依らず効率が良い自動記憶管理方式の開発
2. コンパイラによる SIMD 命令の自動生成法と，レジスタ分割式という SIMD 命令の性質に対応した処理データサイズの推論法，それに SIMD 命令の有効活用を支援するベンチマークの開発

本論文では，構成を第 1 部と第 2 部に分け，第 1 部（第 2 章〜第 4 章）では第 1 項の問題について論じ，第 2 部（第 5 章〜第 8 章）では第 2 項の問題について論じる．

本論文の章立ての構成は，以下の通りである．

第 2 章 ごみ集め問題を定式化し，関連研究を調査し，3 章と 4 章で解決する問題を明示する．

第 3 章 使用中オブジェクトの量に比例する処理時間で完了する滑り圧縮型ごみ集め方式と，実験結果を示す．

第 4 章 滑り圧縮型ごみ集めで世代別管理を行う方式と，実験結果を示す．

第 5 章 本論で取り扱う SIMD 命令セットを明確にし，それに対してどのようなコードを生成すべきかを議論する．さらに関連研究を調査し，最後に 6 章と 7 章と 8 章で解決する問題を明示する．

第 6 章 SIMD 並列化方式の設計と実装と評価を示す．

第 7 章 レジスタ分割式という SIMD 命令の特性に対応して，最適な処理データサイズを推論する方式とその効果を示す．

第 8 章 SIMD 命令セット向けベンチマークの設計と実装と実験結果を示す．

第 9 章 本論文をまとめ，結論を述べる．

第2章　ごみ集めの定式化と関連研究

この章では，先ず，ごみ集め問題の背景を述べ，定式化を行う．そして，提案する方式に関連する研究を紹介する．

1960 年代，FORTRAN の誕生と同じ時期に Lisp[51] が生まれた．Lisp の画期的な点は，リスト構造をはじめとする動的なデータ構造の取り扱いを積極的に支援し，記憶の管理を処理系が自動的に行なった点にある．プログラマは，データの型や，動的なデータ構造の取り扱いには不可避である動的記憶管理を意識しないでプログラムを記述できる．Lisp の意味論では，記憶場所が計算を完了するのに充分なだけ，極端に言えば無限に存在するという仮定がある．これは，つまり記憶場所を使っては捨てるという操作を繰り返すということであり，同時に，変数の動的な束縛（binding）という意味論を含んでいるので，このような記憶場所の確保と解放は，Pascal や C のような手続き型言語のスタックでは実現することができない．プログラムがコンパイルされることを前提にした Common Lisp[68] では，動的な束縛に起因する多くの問題を解消すべく，従前の Lisp の仕様とは逆に，特に指定が無い変数は静的な束縛として，スタック上に確保可能にしている．

そして，本論文の議論の対象のひとつである，「ごみ集め」という問題は，この「無限の記憶場所」の実現に根ざしてしている．このごみ集め問題は，その後に開発された Simula[16]，CLU[50]，SNOBOL4[29]，Smalltalk-80[26]，Icon[28] 等の言語，最近では Java[7] や C#[31] のようなオブジェクト指向型言語の実現において，必須の機能となっている．

2.1　ごみ集め問題の定式化

2.1.1　プログラミング言語と記憶管理

プログラマがプログラム中で認識し得る「記憶」は，通常のプログラミング言語では，変数やデータオブジェクト，あるいは単にオブジェクトと呼ばれ，何らかの名前で参照可能である．以下で，「オブジェクト」という語にはいろいろな意味づけがなされるが，ここでは次の点に注目する．

- ビット列に対する解釈（あるいは型）を有する．
- 何らかの構造を有する．
- 記憶領域に置かれている．
- コンパイル時あるいは実行時に，そのサイズを知ることができる．
- コンパイル時あるいは実行時に，それが置かれている番地を知ることができる．

いかなるプログラミング言語を使ってプログラミングする場合でも，プログラマはオブジェクトがどのように管理されるかということを意識しないわけにはいかない．それでは，オブジェクトの管理とは何か．オブジェクトは次の属性を持つ．

可視範囲（scope） そのオブジェクトを参照することができる範囲.

生存期間（extent） そのオブジェクトが有効である期間.

例えば FORTRAN では，COMMON 宣言された変数の可視範囲は，プログラムテキスト全体であり，全ての変数の生存期間は，プログラムの実行開始から終了までの期間であるといえる．また別の例としては，Pascal の手続き内で宣言された変数（ローカル変数）の可視範囲は手続きの内部であり，生存範囲は手続きが呼び出されてから呼び先に戻るまでの期間であるといえる．上の 2 つの例では，可視範囲と生存期間を翻訳時に決定することが可能である．このような場合を「可視範囲が静的である」とか「生存期間が静的である」という．逆に，可視範囲や生存期間を翻訳時に決定することができない場合がある．このような場合を「可視範囲が動的である」とか,「生存期間が動的である」という．Lisp での動的な束縛では，可視範囲も生存期間も動的である．

オブジェクトに関する 2 つの操作がある．

確保（allocation） オブジェクトをメモリ上のどの番地に置くかを決めること．

解放（release） オブジェクトが実行中のプログラムから参照され得なくすること．

またオブジェクトの確保が，翻訳時に行なわれるならば「静的な確保」，プログラムの実行時に行なわれるならば「動的な確保」という．例えば FORTRAN では，全ての変数は静的に確保され，Pascal のローカル変数は動的に確保される．そして，一般的なプログラミング言語では，静的に確保されたオブジェクトの生存期間はプログラムの実行開始から終了までの間という静的な期間であることに注意されたい．

プログラマが意識する記憶の管理とは,（プログラマが意識し得る）オブジェクトの可視範囲と生存期間，それに確保と解放のタイミングのことである．

以下，オブジェクトの確保が動的に実施される場合に限って話を進める．

2.1.2　動的な記憶管理と自動記憶管理系

まず，どういうときにオブジェクトの動的な確保が可能であると有利か，あるいは必要かということについて考える．その例の一つとして，データ構造のうち，オブジェクトの部品レベルの構造は静的に決まっているが，オブジェクト間の接続で構成する全体的な構造が動的に変化するような場合，例えば翻訳系の構文木を扱う場合が挙げられる．もう一つの例として，データ構造のサイズを動的に決められると，メモリの使用効率を高めることができる場合，例えば記憶域に隙間なく詰め合わせて置かれている文字列の操作が挙げられる．これらの例のように，オブジェクトの動的な確保は，柔軟なプログラミングを支援する．

さて，プログラマはオブジェクトを動的に確保したりや解放する場面でそれを意識するかもしれない．例えば Lisp での，

```
(CONS 'A 'B)
```

のような記述は，処理系の仕組みに習熟したプログラマ以外には，オブジェクトの確保を意識させないであろう．しかし，C の場合の

```
char *cp;
.....
cp = (char *)malloc(anysize);
```

のような記述は，プログラマに（動的な）オブジェクトの確保を意識させずにはおかない．一方で，次のような C の記述に

```
.....
free(cp);
```

対して，プログラマは確保したオブジェクトの「明示的」な解放を意識するに違いない．呼び出された free は，解放を宣言されたオブジェクトを然るべき手順でシステムに返却し，次の再利用に備える．しかし，Lisp での，

```
(PROGN
  (APPEND '(0 1 2 3) '(4 5 6 7))
  (CONS 'A 'B) )
```

のような記述を見て，プログラマは果たしてオブジェクトの解放を意識するだろうか．APPEND で確保された CONS セルは，次の CONS の呼び出しで，プログラムの実行環境（後述）からたどり着けなくなる．このように，プログラマが意識しないうちに実施されるオブジェクトの解放を「暗黙的」な解放といい，解放されたオブジェクトを「ごみ」という．それでは，ごみを，いったい誰が，いつシステムに返却するのだろうか．この作業の実施を「ごみ集め」（Garbage Collection），実施者を「ごみ回収子」（Garbage Collector，GC），あるいは「ごみ集め」という．そして，Lisp での例のような，プログラマにオブジェクトの解放を意識させない記憶管理系を「自動記憶管理系」という．

ここで注意されたいのは，C や Pascal の手続き呼び出しの実現に使用されるスタックも，オブジェクトが動的に確保され暗黙的に解放されるので，自動記憶管理系の一種であると考えられることである．しかし，これはごみ集めではない．それでは両者の違いはいったい何であろうか．それは管理の対象とするオブジェクトの生存期間が，スタックを使用する記憶管理系では静的，つまりコンパイル時に知り得るものであるのに対し，ごみ集めを使用する必要である記憶管理系では動的，つまりコンパイル時には知りえないものである点である．そもそも，生存期間が静的であるならば，プログラムの翻訳時にオブジェクトが占めていたメモリをシステムに返却するコードを実行コードの中に埋め込むことが可能であり，ごみ集めは不要である．

以後の議論では，スタックによる記憶の管理を動的な記憶の管理の中に含めないことにする．また，以後，スタック以外の動的に確保されるオブジェクトを置く領域のことを,「ヒープ」あるいは,「動的記憶領域」ということにする．

ごみ集めの利点

先ず，ごみ集めを具備しているシステムの利点について考える．次のような C プログラムを考える．

```
char *mystrcat(char *str1, *str2) {
  char *cp;
  cp = malloc(strlen(str1) + strlen(str2) + 1); /* 確保 */
  strcpy(cp, str1); strcat(cp, str2);
  free(cp); /* 解放したと宣言し，システムへ返却 ‥ 間違え*/
  return(cp);
}
.....
  myp = mystrcat("This is ", "a pen.");
```

mystrcat の呼び出し手は，確かに mystrcat の呼び出しの直後には 2 つの文字列を連結した文字列を受け取ることができる．しかし，その後でオブジェクトの動的な確保を行うと，解放してしまったオブジェクトに格納されていた文字列のデータは破壊されるかもしれない．その場合，myp を使って連結した文字列にアクセスする際に，再度予期しなかったデータをアクセスするはめになるだろう．このようなオブジェクトの解放の間違いが，プログラムの検出し難い虫の発生の大きな要因であると言われている．ごみ集めを具備するシステムでは，このような事故が発生しないように，しかも自動的に，暗黙のうちに解放されたオブジェクトの再利用への準備，つまりシステムへの返却が実施されることが保証されている．これが，ごみ集めを具備するシステムの利点である．

以下「プログラムの実行環境」という語を,「実行中のプログラムのある時点で，参照可能なオブジェクトの全て」という意味で用いる．そして，あるオブジェクトがプログラムの実行環境に含まれることを,「オブジェクトを使用中である」という．また,「プログラムの実行環境の根」という語を,プログラムの実行環境のうち，動的メモリ領域に確保されたもの以外のオブジェクトという意味で用いる．実行環境の根は，通常は静的に確保されたオブジェクト，プロセッサのレジスタ，それにスタックの内容から構成される．

2.2　ごみ集めの作業の目標と戦略

この節では，ごみ集めという処理の目標と，本論で議論する 2 つの方式の概略について説明する．それぞれの方式の派生アルゴリズムの各論については，2.3 節で議論する．

2.2.1　基本的な目標と分類

ここで，ごみ集めが行う作業の目標について考える．ごみ集め実行の目的は，実行中のプログラムから参照されることがないオブジェクト（つまり ごみ）が占めていた記憶場所をシステムに返却し，その記憶場所を再びオブジェクトの確保に使える状態にすることである．この作業を「回収」という．それを実現するのに，基本的には図 2.1 の第一段の分類のような 3 つの戦略があることが知られている．問題に応じてそれらの混合戦略を用いることもある．3 つの戦略の概略は，次のとおりである．

参照計数器法（reference counting method） [17]
　オブジェクト毎に計数器を用意する．そして，オブジェクトへの参照が増えるような操作の度にそれを増す．また，ポインタの書き換えやスタックの縮みのようなオブジェクトへの参照が

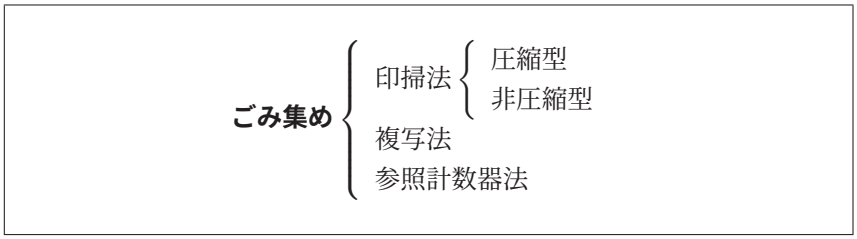


図 2.1: ごみ集め方式の分類

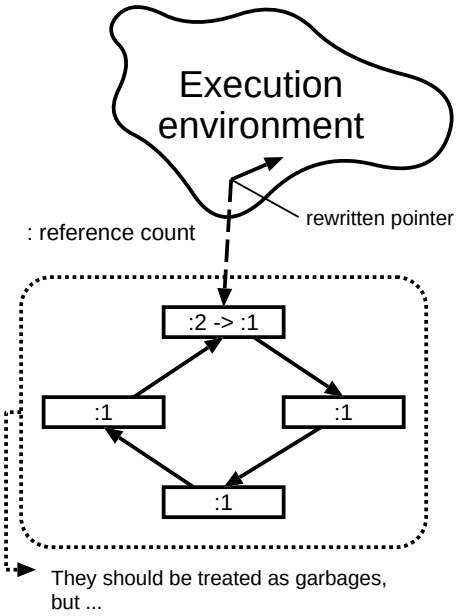


図 2.2: 参照計数器法での循環参照

外れる操作が行なわれる度にそれを減ずる．参照を外されたオブジェクトがごみになったかどうかを，計数器の値が 0 になったかどうかで判定し，ごみになった時点でそれを回収する．例えば実行環境に対して何らかの書き換えを行う際に，上書きされて「押し潰された」データがポインタなら，それが指しているオブジェクトは参照が 1 つ外されたといえる．

複写法（copying method）

参照可能なオブジェクトを，参照関係を保ったままもう一つの動的記憶領域に書き出すという方針．以下,「複写法に基づくごみ集め」のことを「複写式ごみ集め」ということにする．

印掃法（mark and sweep method）

参照可能なオブジェクトの全てに「使用中」という意味の情報（印）を対応させ，動的記憶領域全体を走査し，使用中でないオブジェクト（つまり ごみ）を発見し，再利用する．以下,「印掃法に基づくごみ集め」のことを「印掃式ごみ集め」ということにする．

参照計数器法には，図 2.2 のような循環参照するごみを回収できないという致命的な欠点があるため，そのようなごみが発生しないか，その量が無視できるほど小さい場合にのみ用いられている．また，この問題の解消のために，複写法や印掃法との組み合わせが提案されている．本論文では，参照計数器法についてこれ以上言及しない．次の節で印掃法と複写法の比較を行う．

2.2.2　印掃法

一般的な印掃法では，ごみ集めの作業を大きく以下の 3 つの段階に分類できる．

1. **印付け準備段階**
   印付け用スタックに実行環境の根からヒープへ伸びているポインタを全て積む．
2. **印付け段階**
   使用中オブジェクトの全てに印をつける．
   スタックを使って辿りながら，印がついていないオブジェクトに印付けしていく．スタックが空になった時点で終了する．（スタックを使わないで印付けを行う方法 [63] もある．）
3. **回収段階**
   ヒープ全体を走査して全てのオブジェクトを調べ，それが使用中なら印を外し，そうでないなら再利用のために何らかの操作を施す．

その概略を図 2.3 に示す．

各オブジェクトには，ごみであるかどうかを判別するための情報が付けられており，その表現には通常 1 ビットで十分である．印掃法は，さらに以下のように 2 つに分類される．回収段階での作業の内容が，両者で異なる．

「圧縮型」 図 2.3 の（a）のように使用中オブジェクトをひと固まりにして，確保のための連続した領域を作り出し，ここからオブジェクトの確保を行うもの．

「非圧縮型」 図 2.3 の（b）のように，ごみになったオブジェクトをリストにして，そのリストからオブジェクトを確保するもの．

通常，ごみ集めの対象としてサイズが一定でないオブジェクト（可変長オブジェクト）を扱う場合，圧縮型を用いることが多い．（その例外として，例えば buddy system[39] が挙げられる．）圧縮型では，オブジェクトを移動するので，ポインタの補正が必要である．この事項については，2.3.2 節で詳述する．

一方，サイズが一定のオブジェクト（固定長オブジェクト）を回収の対象とする場合，非圧縮型を用いることが多い．非圧縮型の印掃式ごみ集めと，圧縮型のごみ集めを組み合わせて，可変長オブジェクトの取り扱いを可能にした方式 [71] も考えられる．

通常，印掃法では，オブジェクトの使用領域を圧縮するのに，滑り圧縮法を用いる．これは，図 2.4 のように，オブジェクトの配置順序，つまり確保された順序を保存しながら，詰め合わせを行う圧縮法である．このような特性を「生成順序保存」，あるいは「GOP」（Generated Order Preserving）という．これは，ある種の言語処理系，例えば WAM（Warran Abstruct Machine）に基づく Prolog[6] の実装では必須の事項である．

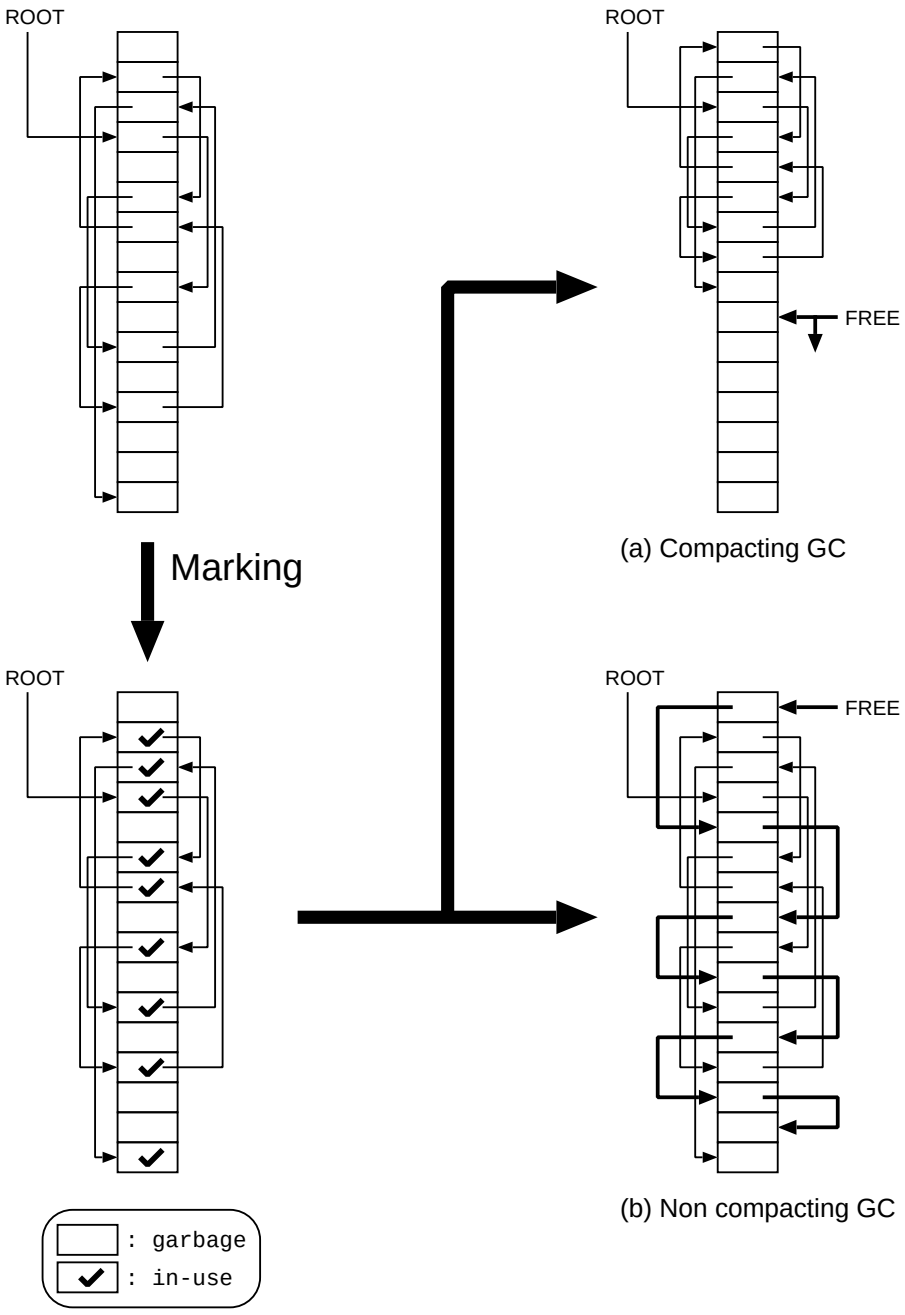


図 2.3: 印掃法に基づくごみ集め（a）非圧縮型 （b）圧縮型

印掃法でのオブジェクトの使用領域圧縮の方法として，入れ換え法もある．これは，Steele の並列ごみ集め [67] で示されている方式であるが，回収の対象が固定長オブジェクトに限られ，生成順序が保存されないので，並列ごみ集めのような特殊な目的に限り使用される．

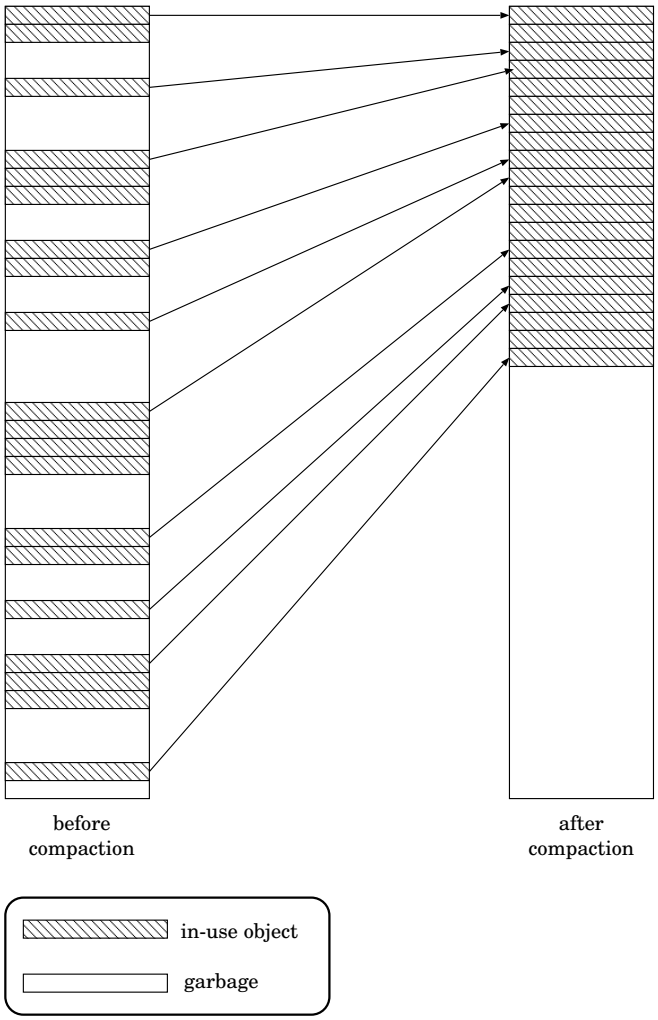


図 2.4: 滑り圧縮と生成順序保存

2.2.3　複写法

複写法を最初に提案したのは，Fenichel ら [20] である．また，これの in-core 版を Cheney[14] が発表した．このアルゴリズムが提案された当時は，一次記憶の単価が非常に高価であったので，Fenichel の論文では複写先領域は 2 次記憶に取り，ごみ集めが完了すると主記憶に読み戻している．しかし，本質的には両者は同じ動作である．この方式では，オブジェクトは，複写先領域に幅優先で複写される．

複写法では，ヒープを半分に分け，ごみ集めが完了する度にその役割を交換（flipping）する．それぞれを「複写元領域」と「複写先領域」という．そのために，ヒープの使用効率が印掃法に比べて約半分になり，ヒープを全て使い切った時にごみ集めを開始するスケジューリングでは，ごみ集めの発生間隔が 2 倍以上になる．また，一般的な複写法では，印掃法と異なり，オブジェクトを辿る段階と，それらを複写する段階に分かれていない．複写元領域の複写済のオブジェクトが置かれていた場

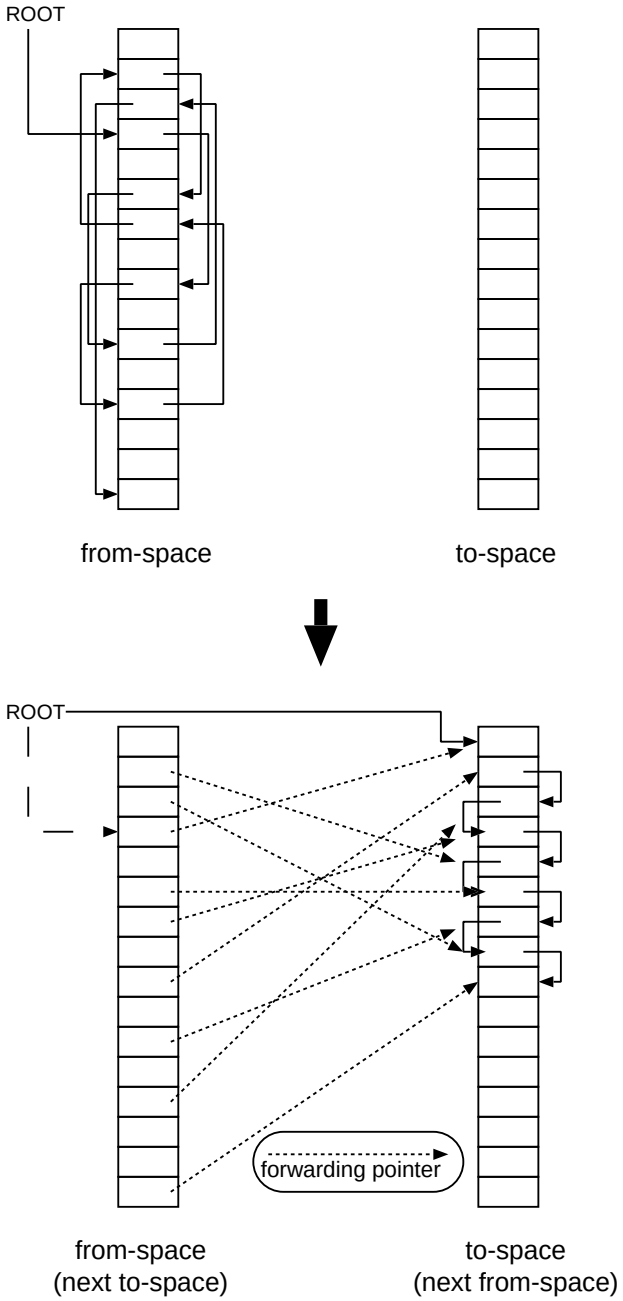


図 2.5: 複写法に基づくごみ集め

所には,「行き先ポインタ」(forwarding pointer) が置かれ，それは複写先領域の複写先を指している．そのため，オブジェクトのデータ構造は，行き先ポインタと通常のオブジェクトを識別できるようにする．

一般的な複写法では，ごみ集めの作業は大きく以下の 2 つの段階に分かれる．

表 2.1: 印掃法と複写法の比較

比較項目	印掃法	複写法
発生間隔	$S - A$	$S/2 - A$
処理時間	S に比例	A に比例
生成順序	保存する	保存しない

S: ヒープのサイズ
A: 使用中オブジェクトの総量
（通常 $S \gg A$ である）

1. **複写準備段階**
   複写先領域に，実行環境の根から複写元領域へ伸びているポインタを全て複写する．
2. **複写段階**
   複写先領域のオブジェクトを順番に，ポインタかどうかを調べる．複写前のオブジェクトを参照しているポインタであれば，そのオブジェクトを複写先領域に複写し，ポインタの値を新しい領域の番地に補正し，行き先ポインタを設定する．複写済のオブジェクトを参照しているポインタであれば，この行き先ポインタの値に補正される．

その様子を図 2.5 に示す．

複写法は特別な工夫なしで可変長オブジェクトを取り扱うことが可能である．そして，複写法では，根からオブジェクトを辿るために用いるスタックは不要である．しかし，一般的な複写法はオブジェクトをほぼ幅優先に複写するので，小出の複写ごみ集め [41]，あるいはそれに類する方法を除くとオブジェクトの生成順序は保存されない．

応用によっては，幅優先よりも深さ優先に複写を行うごみ集めの方が，プログラムの実行効率の向上に貢献する場合がある．中島らは，スタックのための別領域が不要で通常の複写式ごみ集めと同じ使用メモリ量で済み，深さ優先でオブジェクトを複写する複写式ごみ集め [57] を提案している．

2.2.4　圧縮型の印掃式ごみ集めと複写式ごみ集めの比較

圧縮型ごみ集めについて，印掃式ごみ集めと複写式ごみ集めの特性をまとめ，比較したものが，表 2.1 である．この表での比較で判ることは，印掃法はごみ集めの処理時間がヒープのサイズ S に依存することを除けば，他の全ての点で複写法より好ましい特性をもつことである．

2.3　ごみ集めの関連研究

この章では，印掃法と複写法のアルゴリズムで，本研究に関連する成果を紹介する．

2.3.1 印付け方式

通常の印掃式ごみ集めの印付けで用いられるのは，スタックを用いた繰り返しか，あるいは再帰による，深さ優先的な印付け [39] である．

ポインタ逆転法

Schorr らが提案した方法 [63] で，記憶装置が高価であったり，あるいは，メモリ空間が狭くてスタックのために十分大きいメモリ領域が確保できない場合に用いられた方法である．この方法では，スタックを用いる方法の 2 倍以上の時間がかかり，しかも各オブジェクトには，ポインタを反転したということを示すための，付加情報のためのフィールドが必要となる．

SNC 法

Kurokawa は，有限の大きさのスタックと，リストの構造に関する一般例から得られる，スタック消費節約のための経験則との組み合わせで，高能率な印付けを達成する方法 [43] を示した．

走査法

Dijkstra らの印掃法並列ごみ集め [19] で提案された印付け法である．各オブジェクトは，白，灰，黒の状態から成る印を持つ．そして，印が外された状態を白とする．手順は以下の通りである．

```
実行環境の根から直接辿れるオブジェクトに灰印を付ける;
repeat
  for p := ヒープ中の全てのオブジェクト do begin
    if p が灰印 then begin
      for q := p から指されているオブジェクトの全て do
        if q が白印 then q に灰印を付ける
      p に黒印を付ける
    end
  end
until 灰印のオブジェクトがヒープに存在しない;
```

いくつかのショートカット，例えば灰印のオブジェクトが見つかったら p をヒープの先頭に戻してしまう等，があるにせよ，この方式では，灰印のオブジェクトが無くなるまで，ヒープの走査を繰り返さねばならない．このための時間は，回収のためのヒープの走査が 1 回であるのに比べて，十分大きい処理時間がかかるはずである．

2.3.2 圧縮型印掃式ごみ集めの関連研究

第 3 章で議論するごみ集めに関連する研究成果を紹介する．印掃式ごみ集めで使用領域を圧縮 (詰め合わせ）する場合，一般には図 2.4 のような滑り圧縮を用いる．使用中オブジェクトは移動するの

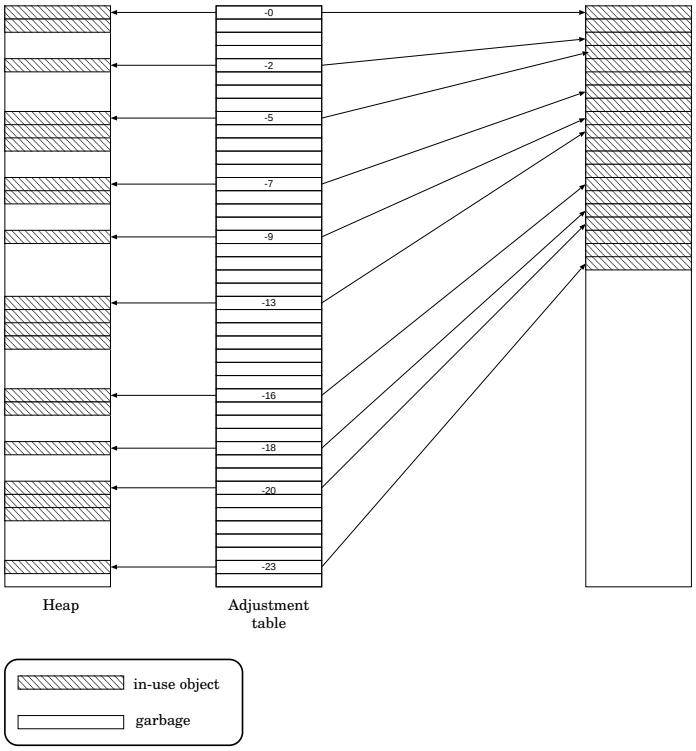


図 2.6: 表を用いるポインタの補正法

で，それらを指しているポインタを補正しなければならない．この種のごみ集めの研究の焦点は，このポインタの補正という問題に合わされていた．初期においては，図 2.6 のように，オブジェクトに 1 対 1 に対応する大きな表を用意して，これにオブジェクトの行き先を登録するという方法が考え出された．この方法を用いれば，1 つのオブジェクトに対応する表のエントリの作成や，1 つのポインタの補正を，定数時間で実行できることは自明である．しかし，この方法は（多くの場合）ヒープと同じサイズの表を持つだけのメモリが必要であるので，非常に不経済である．メモリの使用効率が複写法の場合と同じ 50%になってしまう．

これから紹介するのは，この表のような大きなメモリを必要としないポインタの補正方式である．

参照の鎖を構成する方法

Morris は，ポインタ補正のための特別な表を必要としない滑り圧縮法 [56] を提案した．図 2.7 を用いて，この方式でのポインタの補正を説明する．オブジェクトが配置された番地の大小が，図では配置の高低に対応しているとする．使用中のオブジェクトは，ヒープの上の方に向かって詰め合わされるとする．この例では，オブジェクト `A` を指している，上向きのポインタを補正する．オブジェクトの各要素には，その内容が「引越し」しているということを表すフラグ（図では V ）が付加されている．この V フラグが立つのは，ごみ集めの回収段階のみである．手順は以下の通りである．

`Step0` 印付けが終了し，使用中のオブジェクトは全て黒状態である．この時点で，ごみ集めは使用中

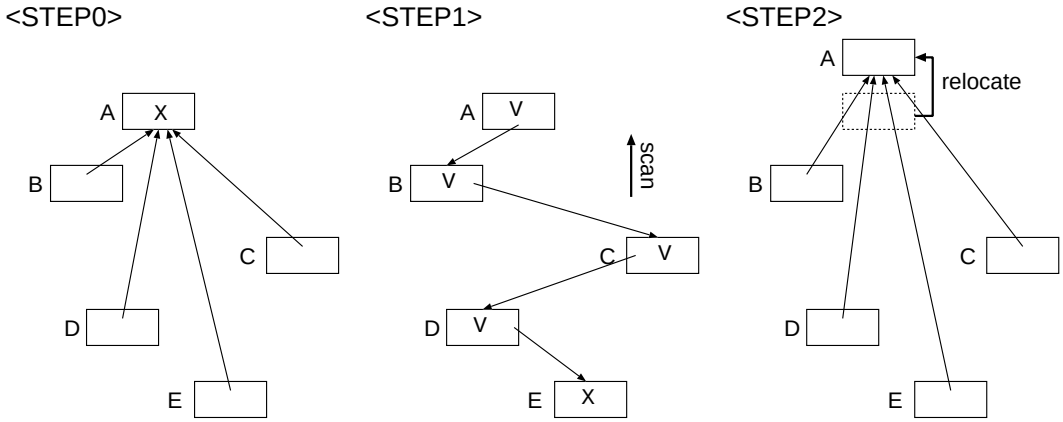


図 2.7: Morris の滑り圧縮法

オブジェクトの総量 A を把握している．A は，オブジェクトがどの番地に移動するかを知るために必要な情報である．

`Step1` ヒープを下から上に向かって走査し，`A` を指しているポインタの鎖を作っていく．最初に出会った `E` には，`A` にあったデータがコピーされ，`A` から `E` を指すポインタが，V フラグ付きで `A` に書き込まれる．

`Step2` V フラグが立っている要素を見つけると，それを V フラグが立っていない要素に合うまで，鎖を辿っていく．鎖の中の要素には，`A` の移動先の番地を書き込み，リロケーションが完了していく．V フラグが立っていない要素は，`Step1` で元々`A` に書かれていたデータであるので，それを `A` に書き戻す．

これと同様のことを，ヒープの上から下に向かって行なえば，ポインタの補正が完了する．

引っ越しフラグが不要な Morris のアルゴリズム

Jonkers は，Morris の方法の改良版にあたる，引越しフラグを必要としないごみ集め [36] を提案している．ポインタ補正と回収作業は，合わせて 2 回ヒープを操作することで達成される．ただしこの方法では，各オブジェクトが少なくとも 1 ワードはポインタ以外のデータを格納することができるということを仮定しており，例えば，Lisp の CONS セルのようなものは，余計に 1 ワードを付加しなければならない．

均衡 2 進木をヒープに埋め込む方法

印掃式ごみ集めで，印付け段階の後のヒープの様子は，一般的には図 2.4 のように，ごみ（白）になったオブジェクトの塊と，使用中（斜線）のオブジェクトの塊が交互に配置した状態になっているはずである．寺島らはポインタ補正のためのデータをごみが置かれている箇所に埋め込むという手法 [73] を第一の方法として提案している．この方法では，ポインタの補正表は均衡 2 進木から構成され

ており，この木を二分探索することによって，所望のポインタ補正値が得られる．1 つのポインタの補正にかかる時間は，黒状態のオブジェクトの塊の個数を C として，$\log C$ に比例する時間である．

小型化されたポインタ補正表

寺島らは，別のポインタ補正法 [75] を提案している．これは，この章の冒頭で述べた補正表を，オブジェクト毎ではなく，オブジェクトのいくつか（2^N）の塊ごとに 1 つの割合で用意して，余計なメモリの消費を防いでいる．(数%で済んでいる．）ポインタ値の補正の度に，表を引いておおまかな補正値を求め，塊の中の黒状態のオブジェクトの個数を数えることで，正確な補正値を求める．よって，1 つのポインタの補正にかかる時間は，定数時間で済む．

2.3.3　生成順序を保存するごみ集め

一般に，印掃法で可変長オブジェクトを扱い得る圧縮型ごみ集めを構成すると，滑り圧縮を用いる．（入れ換え法 [67] では，サイズが均一なオブジェクトしか扱えない.）滑り圧縮では，オブジェクトの生成順序が保存される．

生成順序の保存性には幾つかの利点がある．Prolog 処理系の 1 つの実現法である WAM（Warran based Abstract Machine）では，choice point の記録のためのオブジェクト間で，確保の順序の新旧を判定する必要がある．これを実現するのに，生成順序の保存がもつ，配置番地の大小と確保の順番の新旧の一致という特性を用いれば，余計なデータ構造の導入を必要とせずに実装可能である．

ここでは，A を使用中のオブジェクトの総量（バイト数あるいはワード数），n を使用中オブジェクトの総数であるとする．一般に 1 つのオブジェクトは複数のバイトやワードからなるので，$A > n$ となる．

生成順序を保存する複写式ごみ集め

先にも述べた通り，通常の複写法のごみ集めでは，一般的には，オブジェクトは幅優先に複写される．この性質は，ある種の言語，例えば WAM に基づく Prolog[6] の実現には不都合である．これらでは，オブジェクトが生成された順序を保存しなければならない．小出ら [41] は，通常の複写式ごみ集めと同じ使用メモリ量で，生成順序を保存しながら複写するごみ集めを提案した．このごみ集めは，A を使用中オブジェクトの総量として，$A \log A$ に比例する時間で実行可能である．ここで示されている以下の 2 つの技法は，第 3 章で述べる方式でも用いられている．

- 到達可能なオブジェクトを辿る際に，各オブジェクトの番地を記録していくこと．
- オブジェクトの移動の前に，記録された番地を小さい順にソートすること．

2.3.4　ヒープのサイズに処理時間が依存しない滑り圧縮ごみ集め

ヒープのサイズに処理時間が依存しない滑り圧縮ごみ集め特性をもつごみ集めに関する他の研究を紹介する．

ごみの量に非依存の時間で完了するごみ集め

Sahlin[62] は，n を使用中オブジェクトの総個数として，$n \log n$ に比例する時間で完了する滑り圧縮型印掃式ごみ集めを提案している．この方式でも，第 3 章で述べる「クラスタリング」に相当する作業を実施しているが，そのために印付けと同様の「使用中オブジェクトの辿り」を，印付けとは別の段階として行なっている．このために必要な時間は，印付け並に大きいと考えられる．また，第 3 章で述べる方法と同様に，使用中オブジェクトの固まりの開始番地を小さい順番にソートしているが，これをリスト構造のソーティングで実施しているので，配列のソーティングで実施する場合に比べて，処理時間が大きいことが予想される．

Linear Probing Sort を用いたごみ集め

Carlsson ら [12] は，使用中オブジェクトの開始番地をソートすることで，滑り圧縮を実現するごみ集めを提案している．これでは，ソーティングに Linear Probing Sort という技法を用いており，A に比例する時間で完了すると主張しているが，それを裏付ける実験データ等は示されていない．また，$2A$ 個のポインタ分の付加的な記憶が余計に必要である．これに対して，第 3 章で述べる方法での付加的な記憶量は A で済んでいる．

2.4 世代別ごみ集めとその関連研究

2.4.1 世代別ごみ集めの概略

これまで説明してきたごみ集めでは，ごみ集めが呼び出される度に使用中オブジェクトの全てに印をつけたり，複写を行わねばならない．ところが,「言語やプログラムを問わず，ほとんどのオブジェクトの寿命は短い一方で，ほんの少しの割合のオブジェクトの寿命は比較的長い.」という経験則が見出された [5][49][76][79]．世代別ごみ集めはこの経験則に基づいて，生成されてから「時間」がある程度の経っているオブジェクトと，そうでないオブジェクトを区別して，時間が経っているオブジェクトは今後も使用中になると仮定して，それに対するごみ集めの作業を減らすという戦略が提案された．この戦略に基づくごみ集めを「世代別ごみ集め」（generational GC）と呼ぶ．ここでいう「時間」とは，実時間のことではなく，対象とするオブジェクトが生成されてから，現在までの間に生成されたオブジェクトの個数，あるいはオブジェクトの量（生成に使われた記憶領域のサイズ）のことで，以後「確保時間」（allocation time）と呼ぶ．以下の議論では，確保時間の長短に応じて「古い」「新しい」という．

Liberman[49] らの方式を除いて,「古い領域」と呼ばれる古いオブジェクトを置く領域がいっぱいになるまでは古いオブジェクトはごみ集めの対象にならない，つまり使用中であると仮定する．ある程度古いオブジェクトを選んで，古い領域に移動することを,「終身雇用」（tenuring），あるいは,「殿堂入り」というが，ここに置くオブジェクトを不注意に選択すると，全体的なごみ集めを頻発する原因になる．この選択問題を終身雇用問題（tenuring problem），あるいは，殿堂入り問題といい，世代別ごみ集め研究の主要なテーマである．

以後「逆方向ポインタ」（reverse pointer）という語を，確保時間の長いのオブジェクトから確保時間の短いオブジェクト参照しているポインタの意味で用いる．また，逆方向ポインタのうちで世代を跨ぐものを「世代間ポインタ」と呼ぶ．世代別ごみ集めに一般的な問題として，世代間ポインタの補

表 2.2: 世代別ごみ集め方式一覧

方式	終身雇用までの世代数	半領域の種類と個数	世代記録	世代間ポインタ	
				記録法	検出法
Liberman[49]	G	小 $\times G$	半領域	間接参照	書き換え時
Ungar[76]	G	大 + 小 $\times 3$	オブジェクト	RS	書き換え時
Appel[5]	1	2（伸縮）	領域	RS	書き換え時
Wilson[79]	2	大 + 小 $\times 2$	領域	RS	
小出 [40]	G	3（伸縮）	生成順序	印付け時	

G: 世代数
RS: remember set （世代間ポインタを登録する表）

正が挙げられる．新しい世代のオブジェクトがごみ集めによって回収されないようにしたり，移動したときの補正を行うために，全ての世代間ポインタを何らかの形で記録しておく必要がある．

このためのひとつの方策は，間接ポインタの表を用意し，世代間ポインタを全て間接参照にすること [49] である．これを用いると，オブジェクト参照の度に，間接参照か否かを調べるオーバヘッドがかかり，オブジェクトへ書き込みの度に，間接ポインタの表に登録すべきものか否かを調べ条件を満たせば間接ポインタの表に追加する手間がかかる．ごみ集めの際は，間接ポインタの表から指されているオブジェクトの全てが「使用中」であると仮定して，印付けや複写を行う．

もうひとつの方策は，世代間ポインタの場所の全てを remember set（以下 RS と略す）[76] と呼ばれる表に記録しておくことである．この戦略を用いると，オブジェクトへ書き込みの度に，RS への追加を行うか否かを決定し，必要なら追加する手間がかかる．ごみ集めの際は，RS から指されている全ての（古い領域に配置された）オブジェクトを調べ，ごみ集めの対象となる領域を指しているポインタであれば指されたオブジェクトは「使用中」と仮定して印付けや複写を行い，そうでなければ RS からそのエントリを削除する．

世代別ごみ集めには，いくつかの方式が提案されているが，それらの論点は終身雇用のポリシ以外は以下の点に絞られる．

- 世代の分類数
- 半領域の種類と個数
- 世代情報の記録
- 世代間ポインタの管理法（記録法と発生検出法）

2.4.2　既存の世代別ごみ集め

今までに公表されてきた世代別ごみ集めのうちで主要なものの概略を表 2.2 に示す．これらはすべて複写法に基づいている．

以下，個別に内容を見ていく．各図で新しいオブジェクトは NEW と記された領域から生成され，OLD と記された領域はごみ集めの作業を割愛する領域（終身雇用オブジェクトを置く領域）である

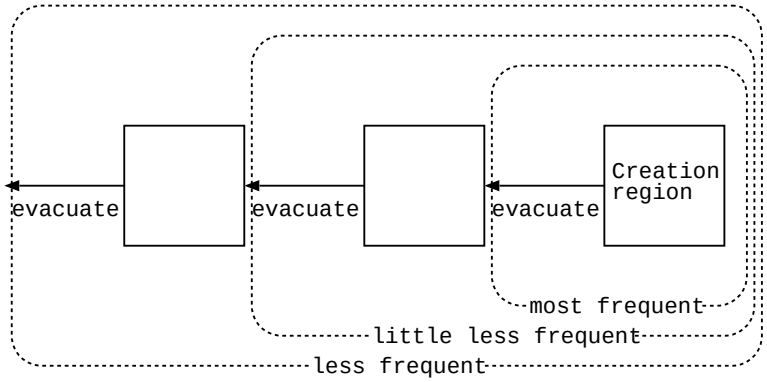


図 2.8: Liberman の世代別ごみ集めのヒープレイアウト

とする．また,「救助」（evacuation）という用語は，領域内に存在する生存しているオブジェクトであって最終的に複写されるもの全体の複写という意味で使う．

Liberman らの実時間ごみ集め

世代別ごみ集めのさきがけで，Baker の実時間ごみ集めの一般化と考えられる．図 2.8 のようにヒープの組を複数用意し，左側ほど救出の頻度が低くなるようにごみ集めのスケジューリングを調節する．世代間ポインタの発生はポインタの書き換え時の検査で行い，それらはすべて間接参照とする．

この論文はアイディア論文と考えられ，具体的なヒープのレイアウトやサイズ等についての提案がない．

Unger の実時間ごみ集め

Liberman とほぼ同時期に発表された手法で，図 2.9 のように大きな 1 つの古い領域と，小さな 3 つの領域から構成される．世代管理は，ごみ集め対象領域の各オブジェクトに付されたカウンタでごみ集めを生き延びた回数をカウントし，それが閾値を超えたら終身雇用する．カウンタのための余計な記憶と，カウンタの管理のオーバヘッドが問題となる．

Appel の世代別ごみ集め

Appel が実装した Standard ML of New Jersey のために設計された世代別ごみ集めである．大きなヒープを図 2.10 の (a) のように OLD，reserve，NEW の 3 つに分割する．reserve の右壁は，reserve の左壁（つまり OLD の右壁）と，ヒープの右端の中央に設定する．これにより，仮に NEW の全てのオブジェクトが生存している場合でも破綻が起きない．free を使い切ってごみ集めが起きる度に (b) のように NEW の生存オブジェクトは reserve の先頭（左端）の X に救出され OLD につけられる．(c) のように OLD がいっぱい（ヒープの半分を超える）になると，(d) のように OLD' に救出され，x と OLD' がヒープの先頭に移動される．この方式の興味ある点は，OLD のオブジェクトをコピー

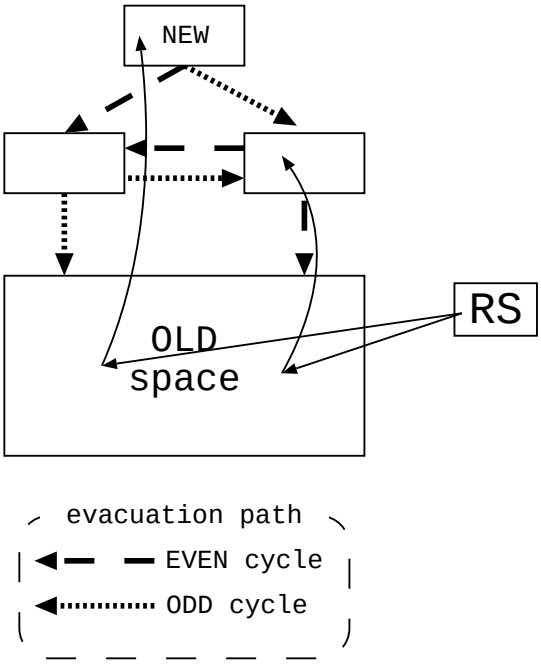


図 2.9: Unger の世代別ごみ集めのヒープレイアウト

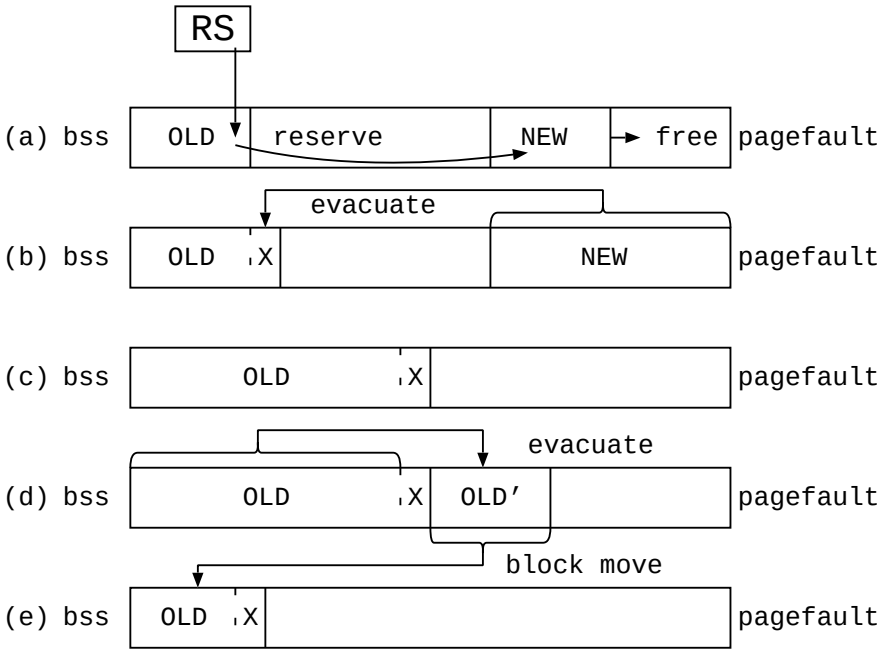


図 2.10: Appel の世代別ごみ集めのヒープレイアウト

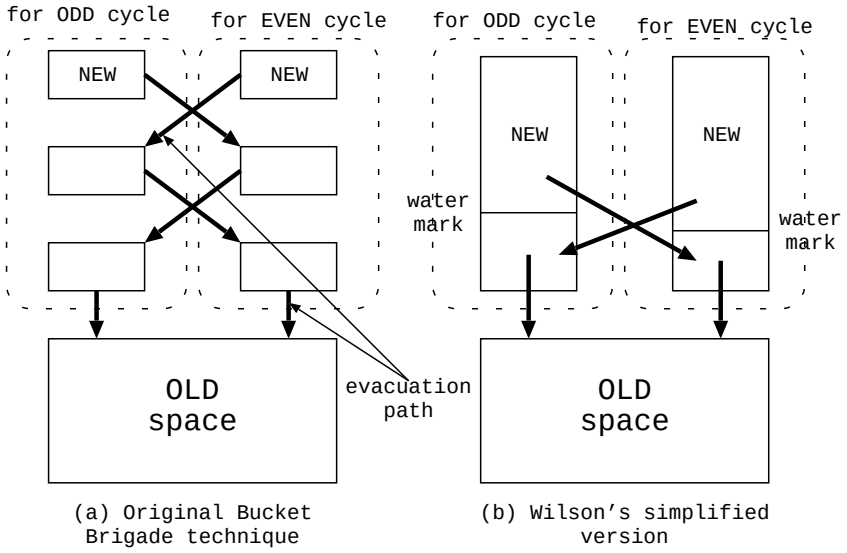


図 2.11: Shaw と Wilson の世代別ごみ集めのヒープレイアウト

する時に，最終的なコピー先でない場所にポインタ値を補正しながらコピーし，後で最終的な目的地にブロック移動している点にある．

このごみ集めは，ML という言語一般の特性から以下のような仮定と対策を講じている．

- ML では書き換えはめったに起きない．
  →書き換えが起きる度に，ポインタの始点と終点をリストに記録する．
- ML ではコンパイルされたコードでも，オブジェクトの確保が 30 命令に 1 回起きるほど頻繁である．
  → free の使い切りはポインタの検査でなくページフォールトで検出する．

この方式では，最近生成されたオブジェクトもすぐに OLD へ終身雇用されてしまう，つまり十分時間が経っていないオブジェクトを終身雇用してしまうので，(c)〜(e) の全体的なごみ集めが比較的短い周期で必要になることが予想される．

Shaw と Wilson の実時間ごみ集め

この論文では，未成熟オブジェクトの終身雇用を避ける問題を，Unger の方式のようなオブジェクト毎の世代カウンタを設けることなしに解決することに主眼を置いている．図 2.11 の (a) に示す Shaw の BucketBrigade[64] は，Liberman らの方式を古い領域の前につけたような構成になっており，小さい半領域のサイズと段数を調整することで，終身雇用までの確保時間を調節できる．Wilson はこれの改良として，調査結果から得た「せいぜい 1 回のごみ集めを生き延びれば終身雇用しても良い.」という経験則から，指標（water mark）を導入して Shaw の方式を簡略化した (b) のような構成を提案した．指標は NEW から救助されたオブジェクトと，一回ごみ集めを生き延びたオブジェクトを区別するために設定され，図で指標より下のオブジェクトで生き延びたものが，古い領域に救助される．

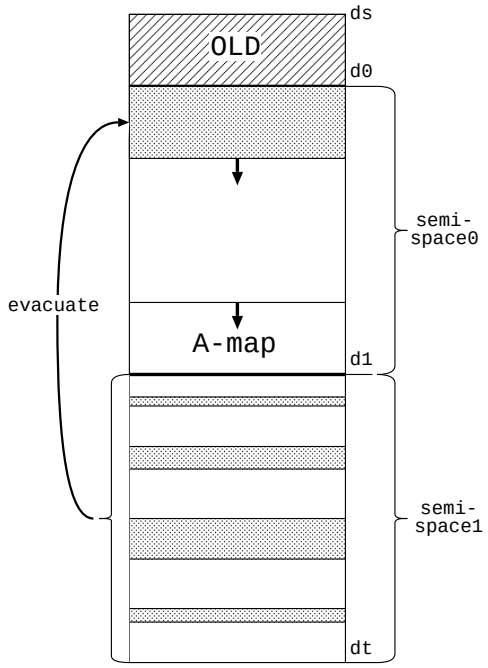


図 2.12: 小出らの世代別ごみ集めのヒープレイアウト

世代間ポインタの追跡には，ページングハードウエアを使った割り出しでカード（32 ワード）毎に古い領域への書き込みを記録し，ごみ集め時に書き込まれたカードの走査を行って，古い領域以外を指している世代間ポインタの補正を行っている．

小出らの世代別ごみ集め

生成順序を保存する複写式ごみ集め [41] を元にした，世代別ごみ集めが提案されている．図 2.12 に示すようなヒープレイアウトになる．何世代かのごみ集めを生き抜いたオブジェクトが半領域 0 に救助されたときに，終身雇用される．`A-map` と呼ぶ領域に，印付けと並行して古い領域も含めた使用中オブジェクト全体のアドレスを収集し，古い領域内の世代間ポインタの補正や，世代間ポインタの指し先の最大値を求める．remember set や間接ポインタのようなデータ構造を使わない代わりに，世代間ポインタと参照先の対が古い領域で閉じるように，半領域 0 への救助の際に新しい領域と古い領域の境界線 d0 を設定する．

この方式では，ポインタの書き換えの際のオーバヘッドが無い代わりに，古い領域のオブジェクトへも印付けが必要である．

2.5　従来の世代別ごみ集めの問題点と第 3 章と第 4 章で解決する問題

概観した世代別ごみ集めでは，小出らの方式を除いて，次のような問題がある．

- 世代間ポインタを管理するためのデータ構造が各図に示した領域とは別に必要である．

- 終身雇用問題の解決のための世代管理で，複雑なデータ構造を用いたり，カウンタの管理などのオーバヘッドが大きい．
- また，論文では触れられていないものがあるが，Appel と Liberman ら以外の方式では，全体的なごみ集めは印掃法で行うものと思われる．

第 3 章では，印掃式ごみ集めで必須であると思われていた，ヒープ全体の走査を省くことができる印掃法滑り圧縮ごみ集めの，アルゴリズムと実装について述べ，評価を行う．

第 4 章では，第 3 章の方式を拡張して，滑り圧縮の生成順序保存という特性を利用した，オーバヘッドが低い世代別ごみ集めを構成する方法と，その実装について述べ，評価を行う．

2.6 この章のまとめ

この章では，ごみ集めの問題を定式化し，印掃法と複写法というごみ集めの 2 つの基本手法を説明し，それぞれについての従来の研究成果を紹介した．そして，世代別ごみ集めの問題を定式化し，従来の手法を紹介した．

第3章　改良型Morrisアルゴリズム

この章では，使用中オブジェクトの量に比例する時間で完了する滑り圧縮型の印掃式ごみ集めアルゴリズムを提案する．

説明に使用するパラメータを表 3.1 のように定義する．

1 つのオブジェクトは，いくつかのアトミックなデータである「ワード」（word）の集まりであるとする．ワードの構成を図 3.1 に示す．各ワードは，Tag フィールドとアドレス型あるいはデータ型のフィールドから構成され，アドレス型の下位 2 ビットにごみ集め時に用いられる M と V フィールドが割り当てられる．それぞれのフィールドの意味は，次の通りである．

`M` **フィールド：** 1 ビットから成り，印付けに使われる．

`V` **フィールド：** 1 ビットから成り，2.3.2 節で説明した Morris の方法の「引越しフラグ」に使われる．

`Tag` **フィールド：** `Data` フィールドのデータの種類（ポインタや整数等）を識別するために使われるタグであるが，ここでは詳しい定義を行なわない．ここで，オブジェクトが使用中であるかどうかは，オブジェクト毎ではなくて，ワード毎に判別可能であることに注意されたい．これは，これから説明する Morris の方式で必要な事項である．

`Data` **フィールド：** （図の点線部）`Tag` フィールドで示されたデータが格納される．

3.1　背景

この章で説明する方式は，Morris の方式の改良版である．Morris の方式の全体的な流れを追うと，図 3.2 のようになる．

図では，オブジェクトが配置された番地が小さいほど，図の上の方に描かれているとする．そして，ここでの滑り圧縮の方針では，オブジェクトを番地が小さい方，言い替えれば図では上の方に詰め合わせるとする．また，オブジェクトの確保は，小さい番地の方から大きい番地の方に向かって実施されるとする．つまり，新しく確保されたオブジェクトほど，大きい番地に配置される．

表 3.1: この章の説明で使用するパラメタ

記号	意味
S	ヒープのサイズ（単位：ワード）
L	使用中オブジェクトの総数
A	使用中オブジェクトの総量（単位：ワード）
N	クラスタ（後述）の個数

データ型	フィールド（32 ビット）	
	Tag	アドレス/データ
CONS データ	000<----27 ビット---->MV	
シンボル	001<----27 ビット---->MV	
ブロック	010<----27 ビット---->MV	
長整数	011<----27 ビット---->MV	
短実数	1x00<---28 ビット------>	
短整数	1x01<---28 ビット------>	
文字コード	1x10<---28 ビット------>	

図 3.1: ワードの構成

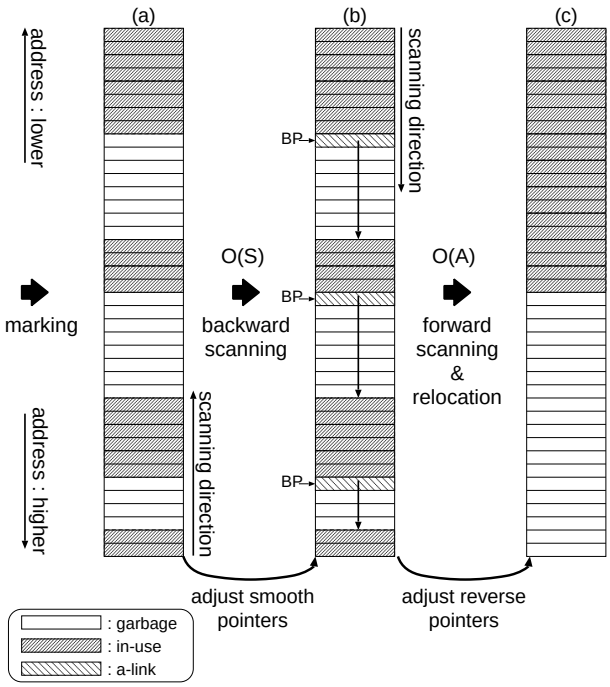


図 3.2: Morris の方式の概略

ポインタを「順方向ポインタ」（forward pointer）と「逆方向ポインタ」（reverse pointer）の 2 種類に分類する．順方向ポインタは，配置番地の大小で，そのワードよりも小さいか同じオブジェクトを指しているポインタである．逆方向ポインタは，同じくそのワードよりも大きいワードを指しているポインタである．2.4.1 節にも同じ用語を定義しているが，2.3.3 節で述べたように滑り圧縮ではヒープの番地が小さい方（つまり図の上の方）にあるオブジェクトほど確保時間が長いという特性があるので，この用語の定義は 2.4.1 節での定義と矛盾しない．そして,「クラスタ」という用語を定義する．これは，使用中オブジェクト（あるいはワードといっても良い）の連続した塊のことで，図の斜線部で示す．

それでは従来の Morris の方式を，図 3.2 の左側から順を追って説明する．先ず，ごみ集めは，印付け準備段階と印付け段階で，使用中の全てのオブジェクトの各ワードに印を付ける．その後の状態が図の (a) である．

次に，ヒープを番地が大きい方から小さい方に向かって，ワード毎にヒープのデータを調べ，順方向ポインタの値を補正していく．その手順は，2.3.2 節で説明した通りである．この時，各クラスタの最後端の次のワード（図の「BP」で指されたワード）には，そのクラスタの次のクラスタの最前端の番地を記録しておく．これらを「a-link」と呼ぶ．これにより，次の段階でヒープ全体を走査する手間を省くことができる．この段階が終ると図の (b) のようになる．

最後に，a-link を使ってクラスタのみを走査しながら，ヒープを番地が小さい方から大きい方に向かって，逆方向ポインタの補正を行ないながらオブジェクト（ワード）を番地が小さい方に向かって詰め合わせる．a-link を用いる工夫は，Morris の論文 [56] には示されてはいないが，処理系の実現者たちの間では常識的な技法である．この段階が終ると図の (c) のようになる．

従来の a-link によって図の (b) の状態から (c) の状態への移行時のヒープ全体の走査は解消されているものの，(a) の状態から (b) の状態への移行時の全体の走査は必要である．つまり，ごみ集めのためにヒープ全体をアクセスすることになる．

しかし，次に挙げる事項を考えると，(a) の状態から (b) の状態への移行時でも，ヒープ全体の走査を行なわないで，使用中オブジェクト，つまり図の斜線の部分のメモリだけをアクセスすることで，作業を完了できることが望ましい．

- 現在の計算機では，ますます主記憶のアクセス時間とキャッシュ記憶のアクセス時間の比が大きくなりつつある．
- ごみ集め発生の時間間隔を大きくするには，ヒープのサイズを大きくすれば良いことは，自明である．
- 一括型のごみ集めでは，A の大きさは実行する応用プログラムの性質のみに依存し，S の大きさには依存しない．さらに，キャッシュ記憶の容量はますます大きくなりつつあり，使用中のオブジェクトの全てがキャッシュ記憶に収まる可能性がある．

S に比べて A が充分小さい（高々10%）と仮定するなら，ヒープの一部に連続領域（O-表と呼ぶことにする）を予約しておき，印付け段階で，印付けしたオブジェクトの全ての最後尾のアドレス（図 3.2 の BP の値）を O-表に記録し，O-表の内容に従って，回収段階を実施するという方策を採れば，ヒープ全体の走査を回避できる．

しかし，滑り圧縮では，Morris の方法以外の 2.3.2 節で紹介した方法を用いるにしても，オブジェクトに配置番地の大きい順や，あるいは小さい順にアクセスして行かねばならない．従来の滑り圧縮ごみ集めが，ヒープを番地の大きい側から小さい側，あるいはその逆へと走査していたのは，このためであった．つまり，表は内容（アドレス）の大小に関してソートされていなければならない．このアイディアに基づくごみ集めは，2.3.3 節や 2.3.4 節で紹介した通りである．しかし，Sahlin のごみ集め [62] を除いて，ソーティングの対象とするデータの件数は，L である．$L \ll S$ でない限り，ヒープ全体を走査するのにかかる時間よりも，ソーティングにかかる時間の方が上回ってしまうだろう．そこで，ソートするデータの件数を減らすことを考える．

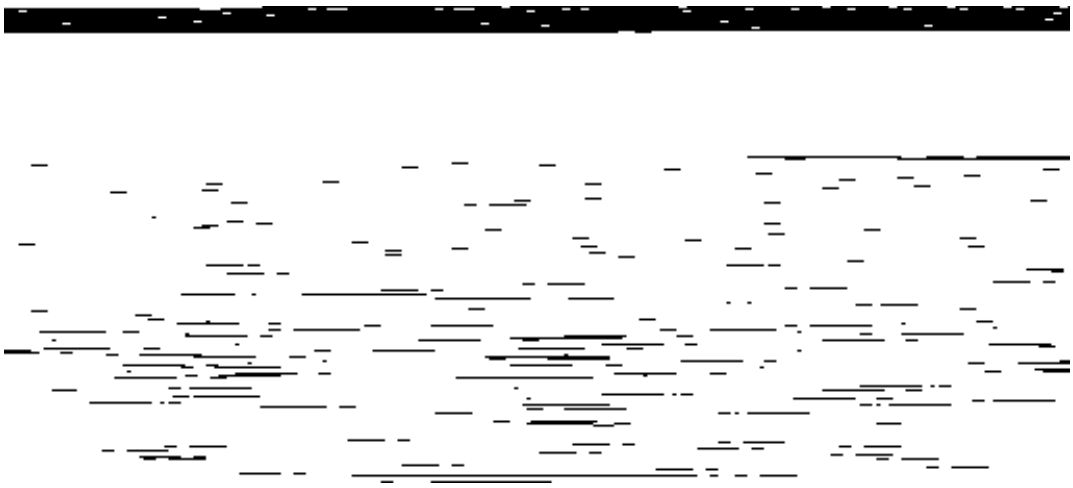


図 3.3: ヒープ中の使用中オブジェクトとごみ

3.2　提案方式

ここでは，Morris のアルゴリズムに対する改良案として提案する方式の基本アイディアを説明し，アルゴリズムを示し，その手間の見積もりを行う．

3.2.1　クラスタリング

図 3.3 は，Chang の TPU（Theorem Proof Unit）という定理証明系 [13] を PLisp[74] 処理系上で実行させて，1 回目のごみ集め起動時の印付けを終了した時点での，印フィールドだけをダンプした結果を図にしたものである．黒い点が「使用中」，白い点が「ごみ」に対応しており，長いビット列を，つづら折りにして図示している．この図から判るのは，使用中オブジェクトはヒープ中に霧散しているのではなく，ある程度の塊になっているということである．これは，このプログラムに限ったことではないということは，後で示すヒープの状態の集計結果により明らかになる．

つまり，ソートするデータの個数を減らすには，オブジェクトやワード毎の番地ではなく，クラスタ毎の番地で O-表を作成し，ソートすれば良い．O-表のエントリをクラスタ毎にする操作を,「クラスタリング」と呼ぶことにする．

ところで，先に触れたように Sahlin[62] は印付け終了後，図 3.2 の (a) の前の時点で，印付けと同様の方法で使用中オブジェクトを辿りながら，各オブジェクトに用意したもう 1 ビットのフラグ F を利用して，クラスタの最前端を探す方法を提案している．これは，クラスタリングと等価な効果をもたらす．しかし，後でデータを示すように，印付け作業はクラスタリングの手間に比べて大きい計算時間を要する．また，リストのソーティングを行う必要があるので，記憶の参照が分散し，ソーティングの実行時間は，提案する方式のように表をソーティングする場合に比べて大きくなると予想される．

クラスタリングは，本質的にクラスタの最後端のアドレス以外を O-表から取り除く作業であり，以下の 2 つの段階から成る．(Morris の圧縮を，ヒープの番地が小さい方に向かって実施するので，最初は図 3.2 の (a) のように，番地が大きい方のオブジェクトからアクセスを始める必要がある．)

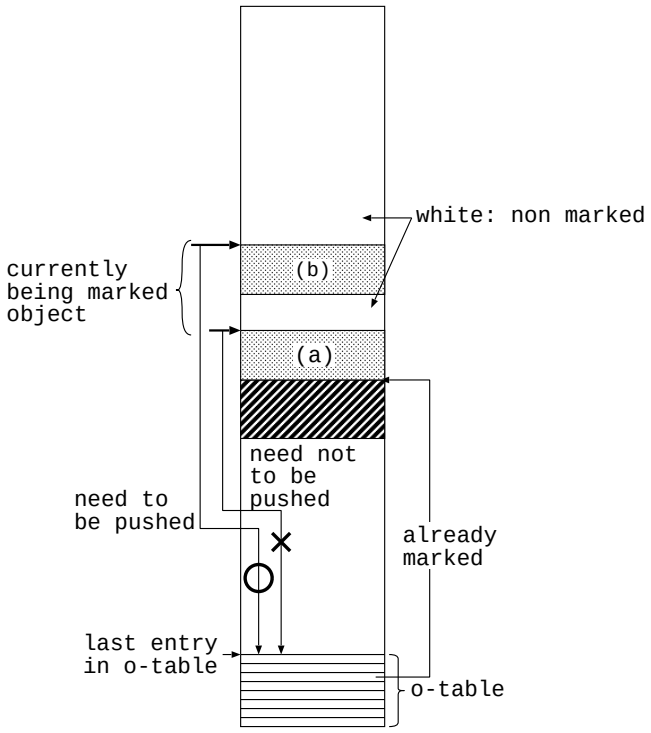


図 3.4: 印付け時のクラスタリング

「印付け中のクラスタリング」(marking time clustering)

印付け段階でごみ集めは，あるオブジェクトに印付けをするとき，オブジェクトの先頭から始めて，後尾に向かって，それに含まれるワードの全ての M フィールドに印を付ける．後端の番地が，クラスタの最後端のものでないので，登録する必要が無いと判定可能であるのは，図 3.4 の (a) を印付けする場合である．それは，この番地は，そのすぐ次に既に印付けされたオブジェクトがあるので，クラスタの最後端ではないと判断しても良い．しかし，図の (b) を印付けする場合は番地を登録しなければならない．それは，すぐ次のオブジェクトは最終的には印付けされるかも知れないが，(b) を登録するか否かを判定する時点では，そのオブジェクトは印付けされていないためである．

「ソーティング前のクラスタリング」(pre-sorting clusterling)

あるオブジェクトに印付けした時点ではクラスタの最後端であるように見えて，その後に同じクラスタのより番地が大きいオブジェクトに印付けされた場合を考える．このような順序で印付けがされると，印付けが終了した時点で，O-表にはクラスタ最後端以外を指す番地が登録されていることになる．そこで，O-表を上下から挟み撃ちにするように 1 回走査して，そのようなエントリを削除する．これにかかる手間が高々$O(A)$ であるのは，明らかである．

```
boolean isMarkrd(word *x); // x で指されたオブジェクトが印付け済み
word *backend(word *x);    // x で指されたオブジェクトの後端のアドレス

int otfull = FALSE;        // O-表がいっぱいであるというフラグ
word *oh = O-表の先頭;
word *ot;                  // O-表の後尾

// オブジェクトを 1 つ印つけ
void mark1(word *obj) {
  word *w;
  if (!isMarked(obj)) {
    *obj を印付け;
    for (w = *obj のポインタ成分) mark1(w);
    if (!M(*(backend(obj)+1)))
       *ot++ = obj;        // クラスタの下端かもしれない→ O-表に加える
  }
}

void GarbageCollector() {
  word *w, *t;             // 作業用変数
  // 使用中オブジェクトに印付け : O(L)
  ot = oh;
  for (w = 実行環境の根の全て) mark1(w);
  // Pre-sorting clustering : O(A)
  w = oh;
  while (w < ot) {
    while (isMarked(*(w+1))) w++;
    while (!isMarked(*(ot-1))) ot--;
    *w = *(ot-- -1); }     // 移動
  ot = w;
  oh から ot の間をソート;    // quick 法なら平均で O(N log N)
  // Backward scanning : O(L)
  t = *oh;
  for (w = oh; w < ot; w++) {
    *(t+1) = w;            // a-link 作成
    *w のクラスタに対して Morris 法の STEP1 を行う;
  }
  // Forward scanning & relocation : O(L)
  a-link を辿りながら各クラスタに対して Morris 法の STEP2 を行う;
}
```

図 3.5: 改良型 Morris アルゴリズム

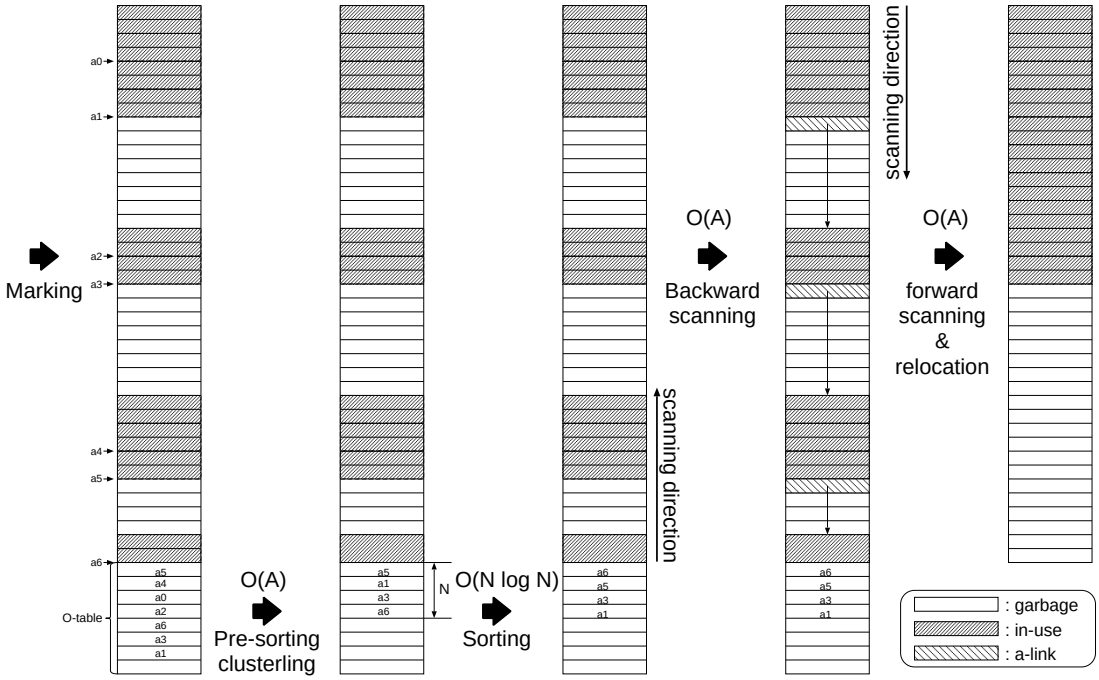


図 3.6: 改良型 Morris アルゴリズムにおけるヒープの様子

3.2.2 アルゴリズムと手間の見積もり

提案方式のアルゴリズムを示しながら，その計算量を見積もる．アルゴリズムの概略を，それぞれのフェーズの計算量と共に図 3.5 に，その様子の図 3.2 との共通部分を省略したものを，図 3.6 に示す．

クラスタリングを終了したら，O-表を番地の大きい順にソートする．これにかかる手間は平均 $O(N \log N)$ に比例する時間である．一般的には，図 3.3 のようにクラスタが構成されるので，$L \gg N$ となる．また，ソーティングする前の O-表の順序もソーティングアルゴリズムに対して最悪のパターンにはならないので，ソーティングのための時間は，後の実験結果が示すようにごみ集め全体の実行時間に比べて十分小さい．

最悪の場合，つまりクラスタが全く構成されず，しかも表の並び順がソーティングアルゴリズムに不利なパターンである場合を考える．ソーティングに使うアルゴリズムにも依るが，計算量は $O(L^2)$ となる．

その後の手間は，O-表を使ってクラスタ間を渡り歩くという点を除けば，図 3.2 を使って説明した場合と同じである．

3.2.3 O-表の溢れに対する対応

印付け段階で O-表が溢れてしまう可能性がある．その場合，ごみ集めの実行時間が急に増えてしまうことを問題としなければ，通常の Morris のアルゴリズムを実行することで，ごみ集めを完了することが可能である．

3.3　実験と評価

性格が異なる 3 つのベンチマークプログラムを使って，実験を行なった．用いた言語処理系は，PLisp[74] で，全てが C 言語で記述されており，高いポータビリティを得ている．

PLisp のごみ集めの部分を改造して，改良型 Morris アルゴリズムを実装した．ソーティングは，単純なクイックソートを用いた．

ごみ集めの各段階にかかった時間の計測は，Unix 系オペレーティングシステムに備わっている C 言語のシステムコール関数 `getrusage()` によって計測した．この関数から得られる時間の精度は，定期的なタイマ割り込みによるプロセスの切り替えの時間間隔に依存しているが，数ミリ秒程度の精度は保証されている．ヒープの様子は，ごみ集めの度にヒープの各ワードの `M` ビットを寄せ集めたデータをファイルに書き出し，それを別のプログラムを使って調査した．

ベンチマークプログラムは，以下の通りである．

- TPU（Theorem Prove Unit）[13]
  定理証明系の一種．全部で 9 個の問題から成る．
- BOYER
  定理証明系の一種．TPU とは異なる動作を持つ．
- BITA
  中西正和氏による LISP 向けベンチマーク．ヒープをスタック状に用いる．

これらの実験では，3.2.3 節に述べたような O-表の溢れはなかった．

3.3.1　クラスタリングの効果

クラスタリングによる，ソートするデータの件数の減少を表 3.2 に示す．

この表から，クラスタリングの効果がたいへん大きいことが判る．BITA のように，クラスタリングにとって厳しいベンチマークでも，実施しない場合の 15%程度にまでソートするデータの件数が減少している．これは，オブジェクトのアドレスのソーティングにかかる時間が 1/7 以下に減少することを意味している．また，例外はあるが，滑り圧縮の効果で使用中オブジェクトが固められるために，2 回目以降はよりデータの件数が減少という傾向がある．

3.3.2　クラスタリングとソーティングにかかる時間

表 3.3 に，改良型 Morris アルゴリズムの処理時間を，各段階毎に示す．ソーティング前の O-表を走査するクラスタリングは，プログラムによって減少の度合が大きいものと小さいものがある．これは，プログラムがオブジェクトの書き換えを多用しているか，そうでないかに依存していると思われる．前者の場合，逆方向ポインタの割合が多くなるので，クラスタの番地が小さい方（確保時間が長いもの）から印付けされていく傾向が強くなり，図 3.4 の (b) のようなパターンが多くなると推測される．後者の場合，ポインタのほとんどが順方向ポインタとなるので，前者の場合の逆の傾向が強くなり，図の (a) のようなパターンが多くなるなると推測される．

表 3.2: クラスタリングの効果（S の単位：ワード）

プログラム：TPU（9 個の問題），$S = 130K$

ごみ集めの起動（回目）	L	O-表の登録数（L に対する割合）	
		印付け後	N
1	9825	400(4.1%)	275(2.8%)
2	6808	141(2.1%)	33(0.5%)
3	3720	309(8.3%)	275(7.4%)

L，N，S：表 3.1 の通り．

プログラム：BOYER，$S = 260K$

ごみ集めの起動（回目）	L	O-表の登録数（L に対する割合）	
		印付け後	N
1	84937	2791(3.3%)	2517(3.0%)
2	119995	2682(2.2%)	2390(2.0%)

L，N，S：表 3.1 の通り．

プログラム：(progn (bita '(a b c d e f g h i) nil) nil)，$S = 130K$

ごみ集めの起動（回目）	L	O-表の登録数（L に対する割合）	
		印付け後	N
1	23797	3602(15.1%)	3539(14.9%)
2	43495	3730(8.6%)	3077(7.1%)

L，N，S：表 3.1 の通り．

プログラム：(progn (bita '(a b c d e f g h i) nil) nil)，$S = 260K$

ごみ集めの起動（回目）	L	O-表の登録数（L に対する割合）	
		印付け後	N
1	47305	7160(15.1%)	7086(15.0%)

L，N，S：表 3.1 の通り．

ソーティング自体にかかる時間は，印付けや圧縮にかかる時間に比較して，十分小さいことがわかる．これは，実用的には，ソーティングにかかる時間がこのアルゴリズムの処理時間を決める大きい要因ではないことを意味している．

3.3.3 従来の方式との比較

ごみ集めにかかる処理時間を，Morris 法や複写法と比較した結果を図 3.7 と図 3.8 に示す．例題には TPU を用いた．総処理時間を (a) に，一回のごみ集めの平均処理時間を (b) に示す．(a) の「load

表 3.3: ごみ集めの各段階毎の処理時間

プログラム	S	印付け	クラスタリング とソーティング	圧縮
BITA	130K	162	53	453
	260K	118	61	330
BOYER	260K	451	35	1304
	400K	245	32	718
TPU	130K	51	6	133
	260K	16	0	42

単位：ミリ秒

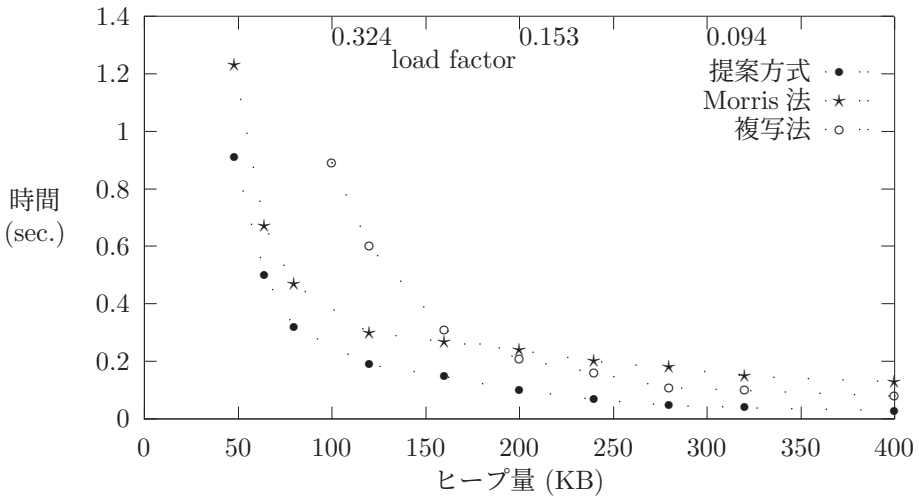


図 3.7: ごみ集めの総処理時間の比較

factor」とは，使用中オブジェクトの総量 A がヒープのサイズ S に占める割合である．

従来の滑り圧縮（Morris 法）や複写法と総時間で比較すると，ヒープのサイズが小さい場合でも大きい場合でも，提案方式の方が少ない時間で済む．平均時間で比較すると，Morris 法がヒープサイズが大きくなるに従って大きくなるのに対して，提案方式と複写法は，ほぼ一定である．平均時間は複写法より大きいが 4 割増程度で，ごみ集めの回数が半分以下なので，総時間では提案方式の方が小さかった．このように提案する方式は，ヒープのサイズが使用中オブジェクトの量に比べて大きい場合でも小さい場合でも効率的で，特に小さい場合に威力を発揮するといえる．

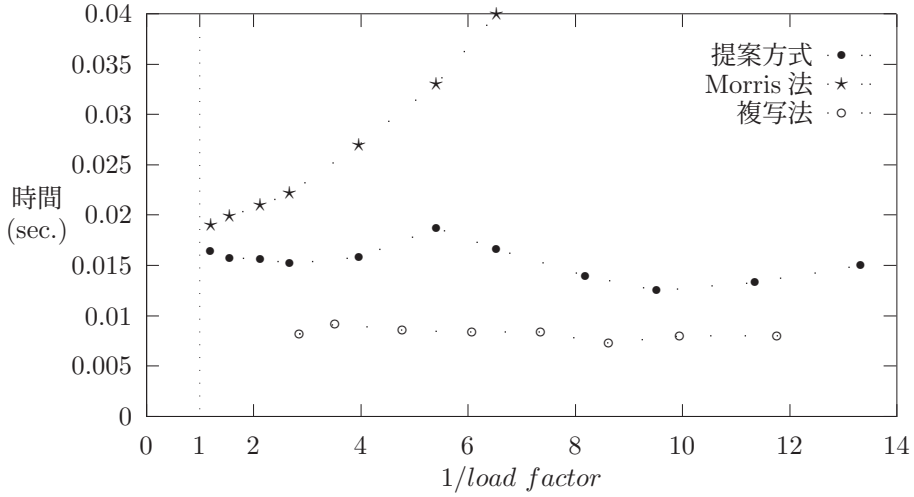


図 3.8: ごみ集めの平均処理時間の比較

3.4 この章の結論

実用的には，使用中オブジェクトの量に比例する時間で完了する，印掃法の滑り圧縮型ごみ集めアルゴリズムを得た．懸念されるソーティングにかかる時間は，クラスタリングと呼ばれる技術によってごみ集めの実施時間を大きく変えるほどにはならないことがわかった．

第4章　生成順序保存を利用する世代別ごみ集め

この章では，生成順序保存を利用した世代別管理法を提案する．

2.4.1 節で紹介したように，世代別ごみ集めの通常動作（OLD 領域がいっぱいになる前の動作）では，複写法を用いるのが常套手段であった．これは，第 3 章で議論した方式が提案される以前は，印掃法のごみ集めでは，ヒープ全体の走査が必要あるというのが定説であったからである．しかし，印掃法の滑り圧縮ごみ集めでもヒープ全体の走査を回避できる，実用的な方法を見出すことができた．そこで，印掃法を基本としても世代別管理法を有効に成立させることができるようになった．

複写法に基づいたごみ集め [40] で，生成順序の保存を利用して，世代管理を行うことが提案されているのと同様に，生成順序を保存する滑り圧縮でも，ヒープの上部（番地が小さい方）にあるほど確保時間が長い，つまり古いオブジェクトであるという特性を利用して，世代管理を行えるはずである．2.4.1 節の経験則に従えば，ヒープの上部にあるオブジェクトほど，ごみになる可能性が低いということになる．

4.1　生成順序保存利用の有効性

図 3.3 の TPU の例におけるヒープ中のごみ（白い点）の分布を見ると，上部の黒い塊（比較的確保時間が長い使用中のオブジェクト）の中にある白い部分（ごみ）は僅かである．これは，ヒープの上部にある使用中オブジェクトの塊がごみになることはまれで，仮にそれらを回収できないとしても，それらがヒープを「無駄食い」する割合は僅かであるということを意味する．そこで，この上部の黒い塊を世代別管理における古い領域とみなし，これに含まれるオブジェクトに対する印付けや移動といったごみ集め作業を省略することで，世代別管理を実現する方法を考える．

4.1.1　オブジェクトの古さの測定

オブジェクトの適切な終身雇用のためには，オブジェクトの古さをある程度精度良く測定できなければならない．小出の方式を除いて，従来の世代別ごみ集めでは，オブジェクトの古さを測定するのに，ごみ集めを生き抜いた回数，つまり世代を使っていた．しかし，本来は「確保時間」，つまりオブジェクトが確保されてからその時点までに確保されたオブジェクトのメモリ量を使うべきである．その精度を上げるためには，生成領域（オブジェクトを確保する領域）も含めて，特定の世代のオブジェクトを救出（2.4.2 節を参照）するための領域の大きさを必要な精度にまで小さくしなければならない．

しかし，生成領域を小さくすることは，ごみ集めの周期を短くすることになる．Wilson のアイディアでは，指標を用いて 1 つの領域を 2 つに仕切り，同じ領域内の世代が異なるオブジェクトを区別し

ている．しかし，各世代の生存オブジェクト数を救出の前に知ることができないので，1 以上の指標を設けることは不可能である（彼は 1 つで充分だと結論した）．

Unger の方式で，世代を数えるカウンタの値を何オブジェクト目あるいは何ワード目に生成されたものかという値にすれば，最後に確保したオブジェクトのその値との差，つまり確保時間を求めて，終身雇用するタイミングを精密に知ることができるはずである．しかし，各オブジェクトにそのための情報を付加せねばならず，例えば Lisp の CONS ならば，1.5 倍の記憶使用量になってしまう．滑り圧縮では，それと同様のことを，ヒープの下端からオブジェクトの位置までの距離（ワード数）を測ることで実現できる．

4.2　提案方式

提案する方式の実装は第 3 章に示したアルゴリズムを拡張する形で行った．以下に述べるアルゴリズムでは，ごみ集めの対象外にされた古いオブジェクトから，ごみ集めの対象になる新しいオブジェクトを参照しているポインタの管理のための，remember set や間接参照等を必要としない．

提案方式の世代別ごみ集めとしての振る舞いは，Appel の方式に近い動きをする．ただし，Appel の方式では，複写法に基づいているのでそうせざるを得ないのだが，圧縮の結果できたヒープの空き領域を半分にして生成領域としていたのに対して，提案方式は空き領域を全て生成領域に振り向けることができるので，ごみ集めの発生間隔を倍以上にできる．また，Appel の方式では，古い領域がヒープの半分に達すると，全体的なごみ集めを行うのに対して，提案方式では，古い領域としてヒープ全体を利用できるので，全体的なごみ集めの発生間隔も倍にできる．ただし，このタイミングは次に詳述するように，1 つのパラメタで制御可能である．

そして，Appel の方式では remember set（RS）を用いて世代間ポインタの管理を行っているが，提案方式では世代間ポインタが置かれている位置の最も上（番地が小さいもの）を更新するだけにして，RS のための領域を必要としない代わりに，古い領域の世代間ポインタを線形探索で調べる必要がある．

4.2.1　アルゴリズム

ヒープの上端と下端のアドレスをそれぞれ HT，HB（HT $<$ HB）とする．次の変数やパラメタを用意する．

指標 cmfa**：** 逆方向ポインタでヒープの最も上に位置するものを指す．

指標 mfa**：** HT から mfa までの領域のオブジェクトは印付けや移動の対象にならない．これを「古い領域」と呼ぶ．mfa より下 HB に至るまでの領域を「生成領域」（generation area）と呼び，オブジェクトの確保に使われ，ごみ集め時には印付けや移動の対象となる．

パラメタ salvage_point **と指標** svgp**：** $0 \leq$ salvage_point ≤ 1 を満たし，svgp は svgp $=$ salvage_point $\times$ HB $+ (1 -$ salvale_point$) \times$ HT として与えられる．mfa が svgp を越えたら，つまり古い領域のサイズがヒープの salvage_point 以上になったら，mfa を HT に戻し，全体的なごみ集めを行う（系をご破算にする）．

```
// 初期化
void init() {
  mfa = HT;
  cmfa = HB;
  cross = false;
}

// フィールド f に対する書き換え時の処理
void setf(word *f, word d) { // d はポインタだと仮定
  *f = d;                    // 書き換え
  if ((f < d) &&             // 逆方向ポインタ
      (f < cmfa))            // 始点が cmfa より小さい
    cmfa = f;                // cfma を更新
  if ((f < mfa) &&           // 始点が古い領域内で
      (mfa < d))             // 終点が古い領域の外
    cross = true;            // 古い領域を跨ぐ逆方向ポインタあり
}

// ごみ集め
void GC() {
  if (cross == true)         // 古い領域から飛び出していたら
    mfa = cmfa;              // 逆方向ポインタを全てごみ集め！
  if (mfa > svgp) {          // 古い領域が水準を越えたら
    mfa = HT; cmfa = HB; }   // ご破算
  cross = false;             // この時点で古い領域を跨ぐ逆方向ポインタは
                             // 全てごみ集め済み
  mfa とヒープ先頭の間のオブジェクトを辿らないようにして印付け;
  クラスタ化とソーティング;
  後ろ向き走査;                // ヒープを下から上への走査 (クラスタのみ)
  if (C0 が古い領域に隣接      // ごみ集めを 1 回生き延びた
    mfa = C0 の次のアドレス;
  // 前向き走査では古い領域も含める.
  // 古い領域で最初の逆方向ポインタを cmfa に設定.
}
```

図 4.1: 生成順序保存に注目した世代別管理

フラグ cross： mfa を跨ぐ逆方向ポインタ, つまり古い領域から生成領域のオブジェクトを指すポインタが発生したことを示す.

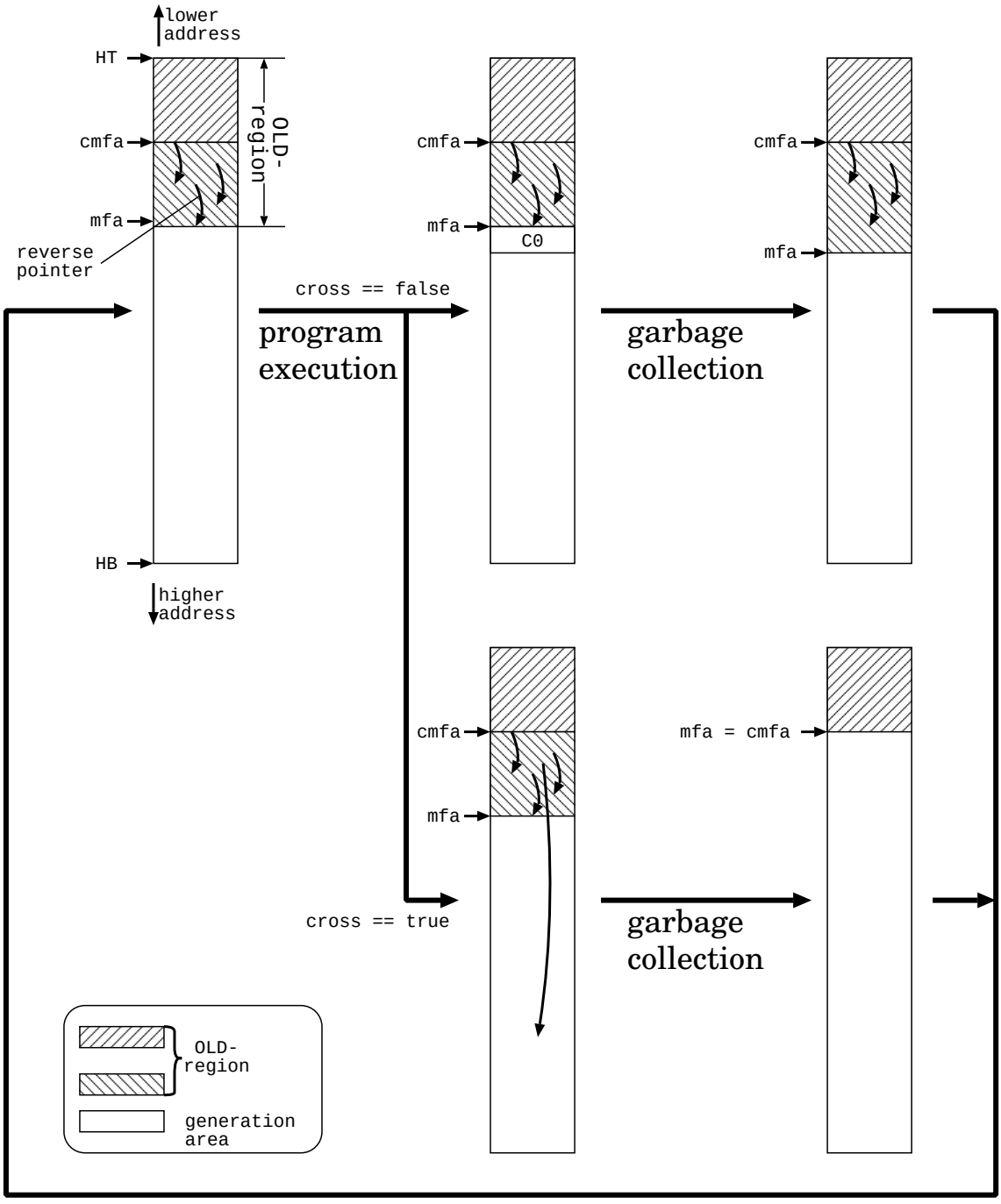


図 4.2: 古い領域の変化

図 4.1 にアルゴリズムを，図 4.2 に古い領域の変化の例を示す．図では古い領域のことを `OLD-region` と標記している．

4.2.2 古い領域の成長

最も上にあるクラスタを C0 とする．このクラスタは，1 回のごみ集めを生き延びたら直ちに古い領域に取り込まれる．図 4.1 では，古い領域を下向きに成長させるには，古い領域内の逆方向ポインタが生成領域を指していないという条件の下で，生成領域の最上位にあるクラスタ（図の C0）が一回のごみ集めを生き延びた場合という厳しい基準になっている．これを,「ある程度の確保時間以上使用中である場合」つまり「生成領域につけた水準線より上にある場合」に改めるには，前向き走査の段階で移動するオブジェクトの移動前アドレスが水準線より上にあるかどうかを調べ，上にある場合は mfa を移動後アドレスに設定することで実現できる．

4.2.3 古い領域の縮退

提案するアルゴリズムの特徴は，古い領域が成長するだけではなく，縮退もすることである．従来の世代別ごみ集めでは，古い領域に相当する古い領域は単調に成長し，その領域がいっぱいになると，系をご破算，つまり全体的なごみ集めを行っていた．

古い領域が縮退するのは，cross が真，つまり古い領域から生成領域のオブジェクトへのポインタが発生した場合である．そのときは，古い領域の下限を cmfa まで引き上げる．

4.2.4 世代間ポインタの管理

古い領域は，印付けと後ろ向き走査からは外されるが，ヒープを上から下に向きに走査して，cmfa を更新する．そのために，古い領域は完全にごみ集め作業から外されるとは言えないが，連続した記憶領域の走査になっており，簡単なアドレスの比較しか行わないので，ほとんど無視できる範囲内のオーバヘッドであると考えられる．

古い領域のオブジェクトにまで印付けを行う小出らの方式以外は，世代間ポインタを，間接参照か RS といった補助データ構造を用いて，管理しなければならない．しかし，提案方式では，古い領域の仕切り線 mfa を，最も上にある逆方向ポインタにまで上げること（古い領域の縮退）でその代わりとしており，そのような補助データ構造を不要にしている．

このようにすると，古い領域が小さくなり，世代別管理のメリットが失われる可能性がある．そこで，小出の方式のように積極的に閉包を作って古い領域を大きくする戦略も考えられる．しかし実際に試したところ，Lisp の処理系と次に述べるベンチマークに限れば，ここに示した方式の方が良い結果を得られた．

4.3 実験と評価

3.3 節でも用いたベンチマークプログラム TPU を使って実験を行った結果を示す．用いた言語処理系は 3.3 節と同じ PLisp で，第 3 章の改良に加えて，さらに提案する世代別管理を組み込んだ．時間の計測方法も 3.3 節と同じ方法を用いた．

表 4.1 に TPU の実行を記憶領域のサイズを変えながら測定した結果と，同じ条件での複写法に基づく圧縮との比較を示す．

TPU の場合のごみ集めの総処理時間を図 4.3 に，同じく平均処理時間を図 4.4 に示す．salvage_po

表 4.1: TPU のごみ集めの結果

生成順序保存を利用した世代別 GC									
記憶領域のサイズ (S)	160 KB (40K)			320 KB (80K)			640 KB (160K)		
salvage_point	1.0	0.2	0.0	1.0	0.2	0.0	1.0	0.2	0.0
全 GC 時間（秒）	0.12	0.18	0.28	0.10	0.11	0.13	0.04	0.04	0.04
平均 GC 時間（ミリ秒）	8.6	12.8	20.0	16.6	18.3	21.7	20.0	20.0	20.0
GC 回数	14	14	14	6	6	6	2	2	2
部分 GC 回数 ‡	13 [9]	8 [7]	0 [0]	5 [3]	5 [3]	0 [0]	1 [1]	1 [1]	0 [0]
全 GC 回数 ‡	1 [1]	6 [3]	14 [11]	1 [1]	1 [1]	6 [4]	1 [1]	1 [1]	2 [2]
使用中オブジェクトの総量 (A)	8840	7667	7646	8196	8196	8120			9100
古い領域のサイズ	7019	5483	5546	4982	4982	4900			2940
クラスタの個数 (N)	57	59	64	90	90	90			165
O–表エントリ数	224	250	328	430	430	465			1259
A/S	0.221	0.192	0.192	0.103	0.103	0.101			0.057

複写圧縮			
記憶領域のサイズ (S)	160 KB (40K)	320 KB (80K)	640 KB (160K)
全 GC 時間（秒）	0.31	0.14	0.06
平均 GC 時間（ミリ秒）	7.21	8.75	8.56
$A/(S/2)$	0.407	0.199	0.092

‡x[y]：ごみ集めが x 回起き，そのうちの y 回はソートを行った．

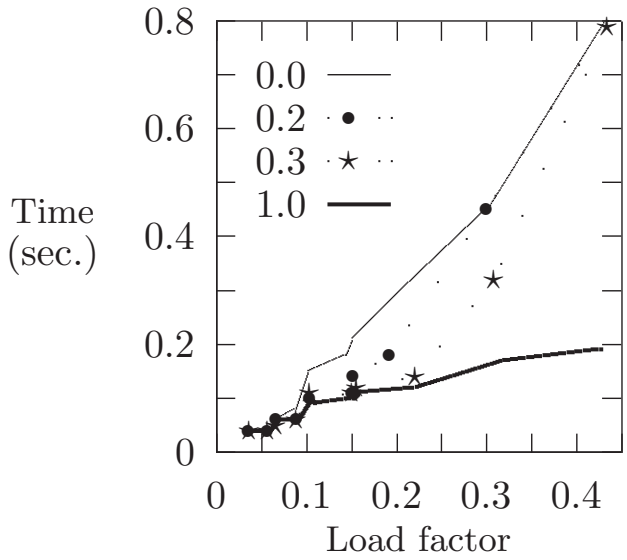
int の値（0.0, 0.2, 0.3, 1.0）毎に，プロットの線を変えている．Load factor は，ヒープのサイズに対する使用中オブジェクトの割合（占有率）である．

世代別管理の適用による性能改善をみるには，salvage_point の値が 0 の時の値を世代別管理がない場合とし，比較の基準とする．例題として用いたような一括処理的な例題（問題が投入されてから解が得られるまでのごみ集めの処理時間が問題となる例題）に限られるが，他のプログラムによる実験結果でも同じ傾向を示した．

salvage_point の変化についてみると，この値を大きくするほど，ごみ集めの総処理時間も平均処理時間も小さくなった．また，この値の変化に関係なく，ごみ集めの総回数は一定であった．

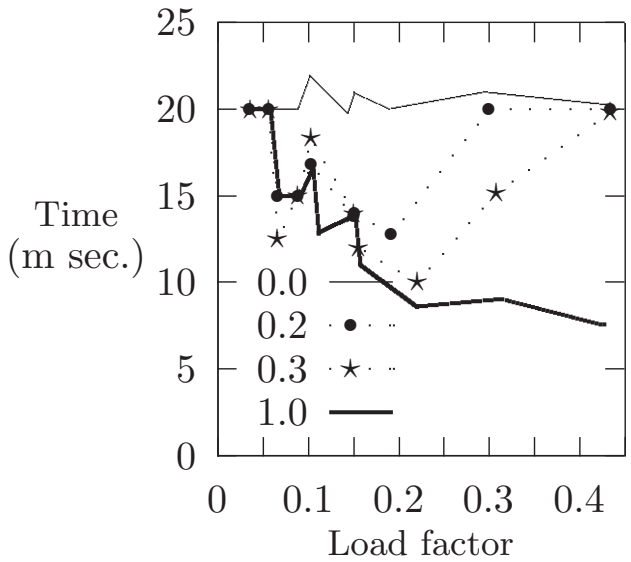
Load factor（$= A/S$）の変化についてみると，この値が大きいほど世代別管理の効果が大きい．$A/S \simeq 0.2$ の時 2 倍以上になったが，$A/S \simeq 0.05$ になると効果が見られなくなった．

複写法による圧縮と比べても，記憶領域のサイズや salvage_point の値に依らず，ごみ集めにかかる総時間は小さくなった．load factor が 0.2 程度で，ごみ集めの処理時間は 1/3 程度にまで小さくなった．ただし，ごみ集めの平均時間は，複写法に比べると大きい．



（各線の 0.0, 0.2 等の値は，salvage_point に対応）

図 4.3: TPU の総処理時間



（各線の 0.0, 0.2 等の値は，salvage_point に対応）

図 4.4: TPU の平均処理時間

4.4 この章の結論

第 3 章で述べたごみ集めが，オブジェクトの世代別管理の導入により，更に高速化した．ここでは第 3 章の滑り圧縮を元にした実装で評価したが，提案する世代別管理法を他の滑り圧縮アルゴリズム

[75] との組み合わせることも可能である.

第5章　SIMD最適化と関連研究

本章では，本研究で取り扱う SIMD 命令について説明し，SIMD 命令向けコンパイラ最適化の関連研究を示す．以後，この最適化のことを，単に SIMD 最適化と呼ぶ．

5.1　本研究で扱う SIMD 命令

レジスタ分割型のショートベクタ処理用 SIMD 命令（以下 SIMD 命令と略）セットは，1996 年に公表された PA-RISC 2.0 の MAX 拡張命令セットや，1997 年の IA-32 の MMX 拡張命令セットを皮切りとして，その後に発表されるプロセッサの多くに拡張命令セットの形で具備されるようになった．その後に発表されたものも含めて，SIMD 命令セットの仕様の概略を表 5.1 に示す．表の「扱えるデータ」欄で符号つきと符号なしを区別しているのは，比較命令や飽和付き加減算命令で直接扱えるのが

表 5.1: SIMD 命令セットのおおまかな仕様

通名	公表	搭載製品	ベクタレジスタ	扱えるデータ型	比較演算の結果
MAX-2	1995	PA-RISC 2.0	I(64) × 32	US16 × 4	(なし)
VIS		UltraSPARC	F(64) × 32	S16 × 2/4, U8 × 4/8	ブーリアン
MMX	1997	MMX-Pentium	F(64) × 8	S8 × 8, S16 × 4, S32 × 2	ビットマスク
MVI	1996	Alpha 21164PC	I(64) × 31	SU8 × 8, US16 × 4	(なし)
SSE	1999	Pentium	S(128) × 8	F32 × 4, F32 × 1	ビットマスク
SSE2	2000	Pentium4	S(128) × 8	F64 × 2, F64 × 1, S8 × 16, S16 × 8, S32 × 4	ビットマスク
3DNow!	1998	K6-2	F(64) × 8	F32 × 2	ビットマスク
Enhanced 3DNow!	1999	Athron	F(64) × 8	F32 × 2	ビットマスク
Altivec	1998	PowerPC G4	S(128) × 32	US8 × 16, US16 × 8, US32 × 4, F32 × 4	ビットマスク
Emotion Engine	1999	PlayStation2	I(128) × 32	US8 × 16, US16 × 8, US32 × 4	ビットマスク

（文献 [25] の表に加筆）

一部の命令セットには包含関係がある．(MMX⊂SSE⊂SSE2，MMX⊂3DNow!⊂Enhanced 3DNow!)
ベクタレジスタ欄： I，F，S はそれぞれ整数（汎用）レジスタ，浮動小数点レジスタ，専用レジスタを使用することを意味する．括弧内の数はベクタレジスタのサイズ（文中の N の値）
扱えるデータ型：U，S，F はそれぞれ符号なし整数，符号つき整数，浮動小数点を，続くの数はデータのサイズを意味する．
比較演算の結果：「ブーリアン」では結果が 1 ビットの 1（真）/0（偽）になる．「ビットマスク」では結果が全ビット 1（真）か全ビット 0（偽）になる．

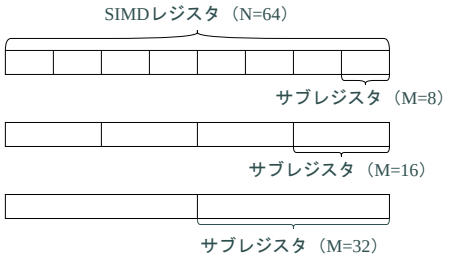


図 5.1: SIMD レジスタとサブレジスタ

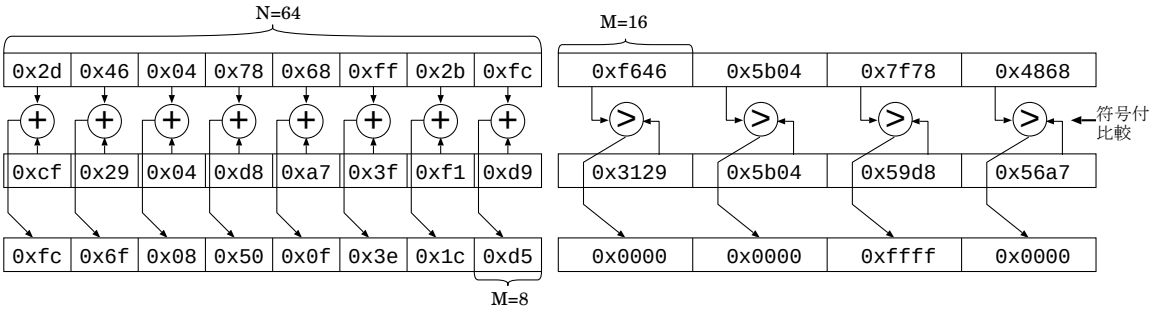


図 5.2: SIMD 命令の例

片方だけである機種を考慮に入れた.

5.1.1　レジスタ分割式ベクタ処理

同じ SIMD 型の計算機でも，スーパーコンピュータに代表されるベクタプロセッサや CM-5[33] のような超並列マシンでは，処理データ幅は，ベクタレジスタの幅や個別プロセッサの基本アーキテクチャが決めるレジスタ幅で決まり，いっぺんに処理できるデータの長さは，ベクタレジスタ長やシステムのハードウエア構成で決定される.

一方で，本研究が対象にしている SIMD 命令は,「レジスタ分割型」という語が示すように，汎用レジスタや倍精度浮動小数点レジスタ，あるいは専用レジスタを分割して，複数のレジスタとして用いるタイプのものである．具体的には，SIMD 命令では図 5.1 のように，256，128，64 ビットといった長さのレジスタ（以後 「SIMD レジスタ」と記す）をベクタレジスタのように扱い，32，16，8 ビットといった長さのレジスタ（以後 「サブレジスタ」と記す）に分割し，図 5.2 や図 5.3 に示すように，2 つの SIMD レジスタの，同じ位置のサブレジスタ間で演算を並列に実行し，結果を同じ位置のサブレジスタに書く形式になっている．以後,「SIMD レジスタのサイズを N」,「サブレジスタのサイズを M」として参照する．例えば IA-32 系のように，片方の SIMD レジスタの代わりに記憶を参照することが許される命令セットや，ディスティネーションレジスタとソースレジスタの片方が同じもの，つまり 2 アドレスのものもある.

SIMD 命令セットの大部分の実装には，特別な演算器を用意する必要はなく，分割しないときの演算器のキャリ伝播回路やシフタに，分割するサイズ毎に切断する論理を組み込むだけで済み，非常に

安価に実装できる．

多くの命令では演算の並列度は N/M となる，つまりループを N/M 周するのに相当することを 1 命令で実施できるので，同じ N の値ならば，M の値が小さいほど演算の並列度が高くなり，実行速度が高まる．

従来のベクタマシンでは，64 ビットの倍精度浮動小数点数レジスタが 64 個以上並んで，ひとつのベクタレジスタを構成している．これと比較すると，SIMD レジスタの規模は小さい．従って，ベクタレジスタとメモリ間の転送やベクタ命令の起動にかかるコストに比べて，SIMD 命令のそれは小さいので，ベクタマシンが苦手とする小規模な処理にも適用して，処理性能を向上できる．一方で演算の並列度 N/M を高めるために，第 7 章で述べるような M の値を小さくする方策が必要である．

サブレジスタのサイズで決まる処理データサイズ M は，プログラマが処理対象に合わせて最適な値を選択する．通常は N/M の値は 2〜16 程度の 2 のべき乗になり，ベクタマシンの数百や超並列マシンの数千といった大きな並列度ではないが，主記憶に対するバーストアクセス性能には充分見合っている．また，ベクタマシンのようなベクタ並列性だけではなく，フィールド間のシャッフルやマージ演算を使って，C の構造体に相当するデータ構造のフィールド間の処理の並列化も可能な場合がある．そして，小規模ながら SIMD 型処理命令の体裁をとっているので各種ループ変換 [3][8][22] のような SIMD 計算機やベクタプロセッサ向けの最適化・並列化技術を応用できる．しかし，ベクタマシンの gather/scatter 命令 [33] が可能にしているメモリ参照機能や，超並列マシンの通信ネットワークで可能なメモリ参照機能に相当する命令を持たないので，それらに特化した最適化やアルゴリズムを使うことはできない．

SIMD 命令セットを効率よく活用するには，処理データサイズ M をなるべく小さくして演算の並列性を高める必要がある．しかし，第 7 章で議論するように，高級言語には一般に「汎整数拡張」（integral promotion）という規約が設定されており，これを遵守しようとすると，SIMD 命令セットを効果的に適用できない．それに対する解決策は，第 7 章で示す．一方で，SIMD 命令セットにはビット毎の論理演算も用意されているが，この場合は処理データサイズの幅は N である．

5.1.2 比較命令

条件文（if 文）や C 言語の条件式を機械語に翻訳すると，通常命令セットでは分岐命令になる．一方で高クロック化されたプロセッサでは，一般に命令実行パイプラインが長くなる傾向にある．分岐命令を実行すると基本的にパイプラインが無効になるので，高クロックの恩恵を享受するには，分岐が少なくなるようにコンパイラが努力するか，分岐予測等のマイクロアーキテクチャ上の工夫が必須である．しかし，通常命令セットの範囲内ではコンパイラによる努力には限界がある．例えば，分岐予測は分岐の成立と不成立がランダムに決まる場合には役に立たない．

一方で，メディアデータ処理プログラムのホットスポット（処理時間の割合が高い部分）で見られる条件文に相当する箇所は，2 つの値の大きい方を取るなど，単純で then 部や else 部の基本ブロックが小さいが，分岐の偏向が少なく分岐予測がうまく働かないものが比較的多い．そのような場合には，VLIW（Very Long Instruction Word）命令セットでは常識化している，述語付き命令（predicated instruction）の活用が有効である．述語付き命令とは，各々の命令に命令実行の副作用，つまりメモリへの書き出しやレジスタの書き換えを実施する条件を指定する情報（述語）が付加されているものである．例えば IA64 や ARM では全命令が述語付きとなっており，IA-32 の PentiumPRO 以降の拡張命令セットや Alpha には述語付き命令と同様の効果を有するものが含まれている．例えば大きい方

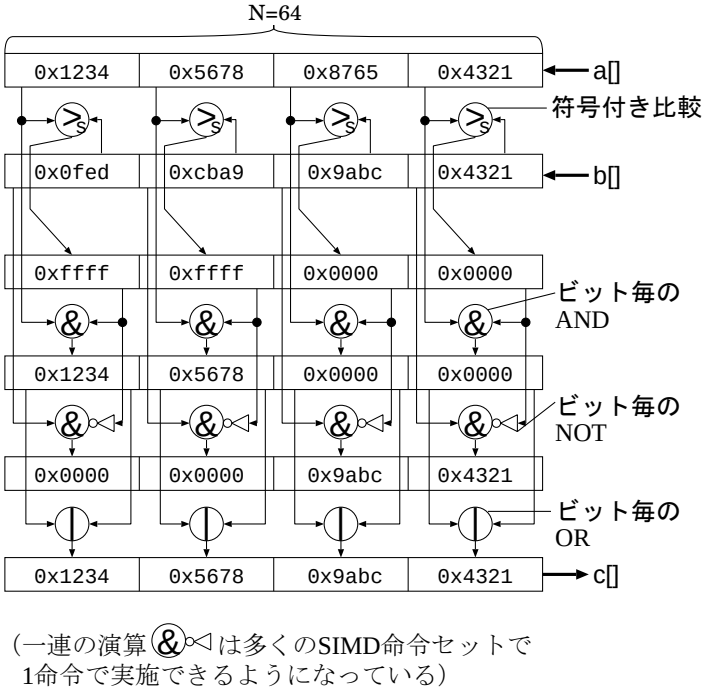


図 5.3: SIMD 命令セットにおける比較演算の例

```
c = (a > b) ? a : b; のコンパイル結果

/*  述語付き命令を使わない場合  */          /*    述語付き命令を使う場合    */
        cmp a,b     /* a - b  */              cmp a,b     /* a - b      */
        bpl label1  /* positive */            pl:mov a,c  /* if positive */
        mov b,c     /* c = b  */              mi:mov b,c  /* if negative */
        bra label2
label1:
        mov a,c     /* c = a  */
lavel2:
```

図 5.4: 述語付き命令を使う最適化の例

の値を得る演算コードに対して，コンパイラは図 5.4 のように，then 部と else 部の処理を行う命令のそれぞれに，then 部や else 部を実行する場合の条件に相当する述語を付加したものを，交互に並べ

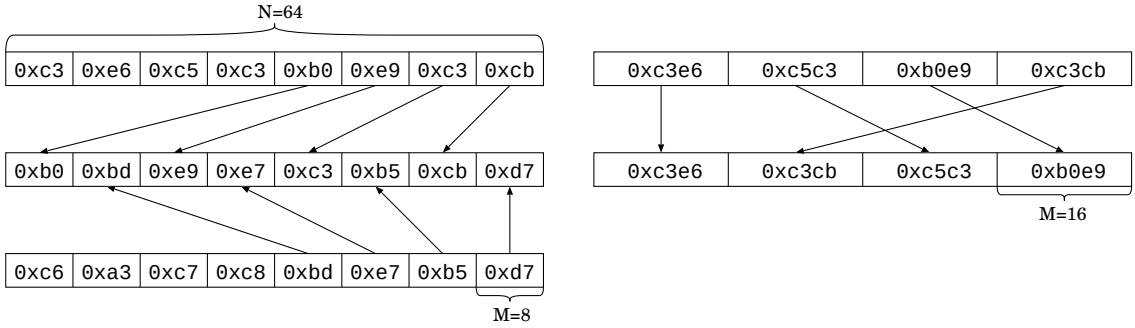


図 5.5: シャッフルやマージ命令の例

る．結果的にいずれの条件が成立しても，then 部と else 部に相当する命令を実行するので，それぞれの基本ブロックの実行にかかるクロック数が，命令パイプライン破壊のペナルティよりも小さい場合でないと，述語付き命令を使うメリットがない [61].

一方で「SIMD 命令」という語が示すとおり，N/M 個のデータに対して同じ演算を適用するので，条件文でデータ毎に処理が異なる場合に対する何らかの対応が必要である．Sparc の VIS[70] を除く現行の SIMD 命令セットのほとんどでは，比較演算の結果は図 5.3 に示すように，同じ幅のレジスタのフィールドの全ビットを 1 とする「真」と，全ビットを 0 にする「偽」として表現する．表 5.1 の「比較演算の結果」欄でビットマスクとなっているものがこれに該当する．通常は，if 文や C 言語の条件式は，if 変換 [2] を施されて選択演算に変換される．この真偽値は，図 5.3 の右側に C 言語の演算子を用いて表したような，積和の論理演算によってデータの選択をフィールド別に行う命令列のビットマスクとして使われる.（この例では大きい方の値を求める演算になっている.）

このように SIMD 命令では if 文相当の部分はデータ依存に変換されて処理するので，その複雑さにも依るが，分岐予測が当たらないような場合でも，パイプラインを乱すことがなく，通常命令で処理する場合に比べて高速に実行可能である.

また，比較命令が生成する真偽値は，それぞれ -1 と 0 の数値として用いられることもある．これをうまく利用すると，積和を用いて $-1/0$ 又は $1/0$ の値を選択するコード生成を行う場合に比べて，演算のステップ数を大きく減り高速化できる場合がある.

5.1.3 SIMD 命令セットに特有の特殊な演算

この節では，SIMD 命令に特有の命令を紹介する.

シャッフル・マージ命令

SIMD 命令セットの多くは，図 5.5 のようなシャッフル命令やマージ命令を有している．現状では，スカラベクタ変換（ベクタレジスタの各フィールドに同じ値をセットする演算）や，リダクション演算（ベクタレジスタの各フィールドの値の総和を求める演算等）等における定型コード生成に使われるに留まっている．機種によっては，例えば IA-32[34] の MMX 命令セットのシャッフル命令図 5.6 のように，フィールドの出元を選択できるような柔軟な動作をするものもある.

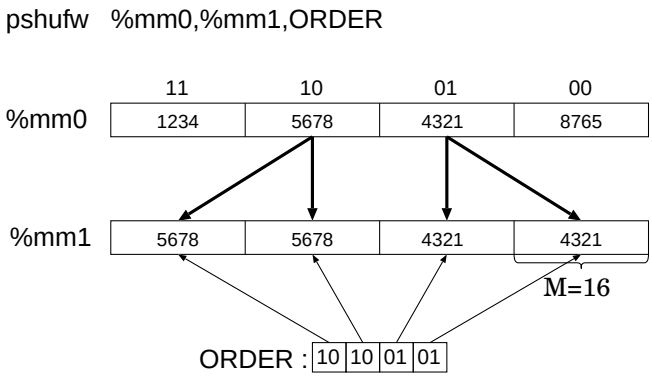


図 5.6: シャッフルやマージ命令の例（IA-32/MMX の場合）

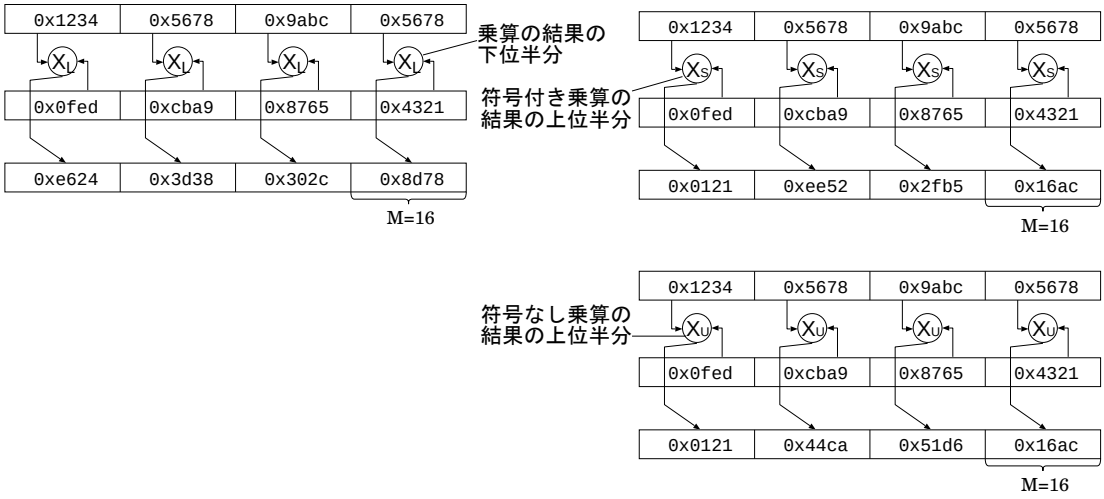


図 5.7: 乗算命令の例（IA-32/MMX の場合）

第 6 章で述べる最適化方式では，シャッフルやマージのパターンを，BONE と呼ぶ命令動作を記述するテンプレートに記述し，演算を表す中間表現とマッチして対応する命令を生成する．

乗算命令

乗算命令の結果を格納するには，通常はデータサイズの 2 倍のサイズのレジスタを必要とするが，5.1.1 節で説明したフォーマットでは，積の全ての結果を格納することはできない．乗算のデータの取り出し方や，結果の格納方式は拡張命令セットによってまちまちである．IA-32/MMX や SSE2 では，図 5.7 のように積の上半分を残す命令と下半分を残す命令が用意されている．PowerPC の Altivec（Velocity Engine）では，図 5.8 のように結果の上半分を残す命令のみが用意されている．MIPS アーキテクチャ拡張の PlayStation2 の EmotionEngine では，補助レジスタ HI と LO を使って積の結果の全てを残すようになっている．

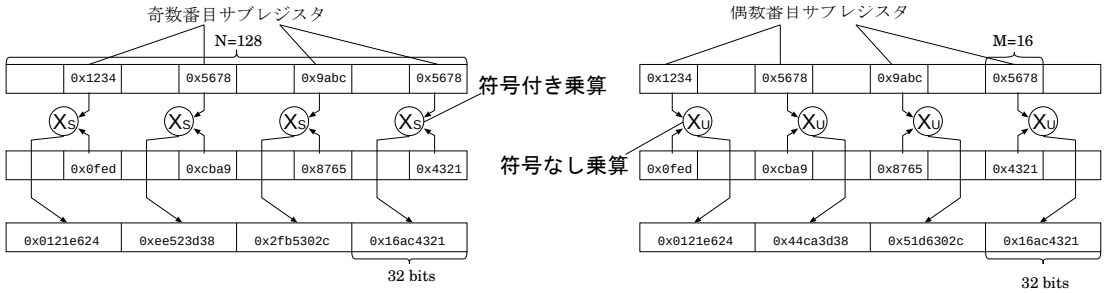


図 5.8: 乗算命令の例（PPC/Altivec の場合）

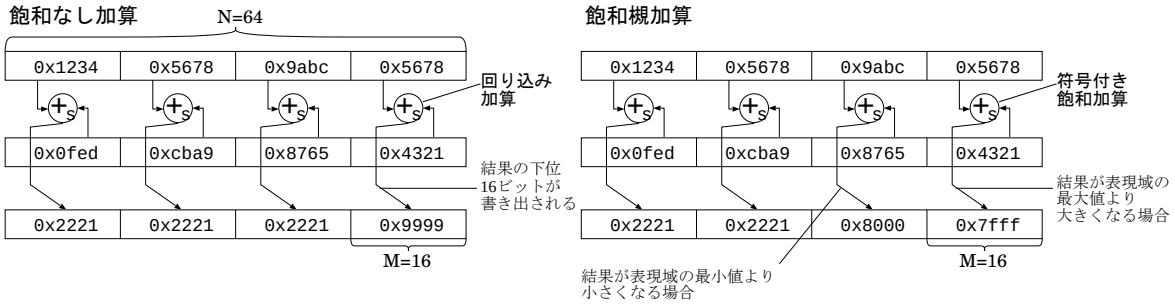


図 5.9: 飽和加算の例

飽和演算命令

演算の結果が決められたデータサイズと表示方式（符号あり/符号なし）で可能な表現範囲で溢れを起こした場合に，結果を表現可能な最大・最小値に抑え込む操作を「飽和」と呼ぶ．そして加算や減算の際に，同時に飽和処理を行う命令が用意されている機種もある．例えば図 5.9 のように 16 ビットのデータサイズ同士の加減算に対して飽和処理を行う場合，飽和処理がない演算命令を使って飽和処理を行うと，加減算の結果を 17 ビット以上使って保持して，溢れの判定を行う必要があるので，並列度が半分になり，さらにデータサイズの変換や値の選択のオーバヘッドがかかる．しかし，飽和つきの加減算命令を適用すると，16 ビットの処理データサイズでだけで済む．

特殊なリダクション命令

メディアデータの非可逆圧縮では，要素ごとに標本値と符号化後の値の差の絶対値を求め，それらの和を求めるという演算を多用している．それに対応して例えば SSE2 や VIS 等の拡張命令セットでは，図 5.10 に示すようなフィールド毎の 8 ビットデータ間の差の絶対値を求める命令を用意している．

この命令は，本来の用途以外に，0 との絶対差を求めるような命令列を生成することによって，ベクタレジスタのフィールドの総和を求めるといった用途にも応用できる．この種の命令の本来の適用範囲は比較的狭いが，上のような使い方を 8.2.3 節で示すようなプログラミング上の工夫やコンパイラの最適化で行うことによって，適用範囲を広げることができる．また，シャッフルやデータサイズ

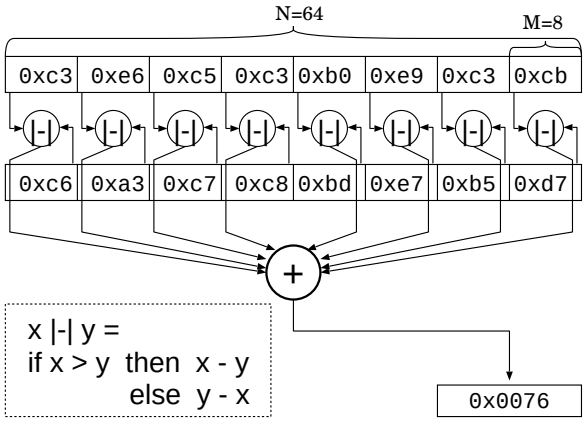


図 5.10: 特殊なリダクション命令の例

表 5.2: SIMD 命令セットの特殊命令の仕様

通名	飽和加減算	最大最小	平均	飽和縮退	シャッフル	乗算	積和	絶対差の和
VIS				S	有	8×16		累積
HP-MI	有		有		有	補助		
MMX	有			有	有	上下別	有	
MVI		有						有
SSE		有	有		有			有
SSE2	有	有	有	有	有	有	有	有
3DNow!		有			有			
Enhanced 3DNow!		有	有		有			有
Altivec	有	有	有	有	有	上	有	
Emotion-Engine	有	有			有	上下同時	有	

（文献 [25] の表に加筆）

飽和縮退：「S」はスケーリング付きであることを表す.
乗算：「補助」は乗算のための補助命令が用意されていること,「上下別」は積の上位と下位を求める命令が別であることを表す.
絶対差の和：「累積」は結果が累積されること表す.

変換を組み合わせてリダクションを行う場合に比べて，命令のステップ数が非常に少なくなり，高速化に大いに寄与する場合がある.

5.2　SIMD 命令向けコンパイラ最適化

この節では，本研究における SIMD 最適化法について，その概要を示す.

5.2.1 SIMD 最適化戦略

本研究では，SIMD 命令を生成するためのコンパイラ最適化が行うべき事項を，次の 2 つに切り分けた．

(1) SIMD 命令向けベクタ化

(2) SIMD 命令向け並列化

本研究では，このうち SIMD 命令向け並列化に重点を置いた．

5.2.2 SIMD 命令向けベクタ化

SIMD 命令向けとはいえ，ベクタ化は従来のベクタマシン向けの最適化・並列化技術 [4][80] の延長上にあると考えられる．SIMD 最適化の従来研究の多く [21][44][65] や，Intel の icc[10] 等で実装されているものは，SIMD 命令向けベクタ化に重きが置かれていると考えられる．

例えば，図 5.11 の定義 1 から定義 2 へのコードへの変形は，ベクタ化が担当する．その際に補助変数 `t[]` が導入されているが，これはベクタ化の「スカラ拡張」変換の一例である．一般には `t[0:7]` への集積をループを回りきるまで続けるが，ここでは 8 ビットの変数を集積に使っているので，ループを 255 周する毎に `t[]` の寄せ集めをしている．これは，サブレジスタの表現域を考慮した，`t[]` の寄せ集めの削減策で，SIMD 命令に特化している．

このように，SIMD 命令向けベクタ化の多くは，ソースコードレベルで可能な変形であるので，ソースコードが有するデータ構造や制御構造の情報を用いるのが適切である．本研究で実装に用いた COINS コンパイラインフラストラクチャ（5.4 節で詳述する.）では，言語寄りの中間表現における変換としてベクタ化の実装を検討しているが，現在は手作業でソースプログラムに対して行っている．この最適化を経たプログラムに対して，次に説明する SIMD 命令向け並列化を実施する．

5.2.3 SIMD 命令向け並列化

本研究では，Leupers の研究 [47][48] と同様に，主に SIMD 命令と中間言語の命令列との照合に焦点をあてた．換言すれば，SIMD 命令を意識して記述したプログラムに対して，プログラマが想定する SIMD 命令を適用することに集中した．これを「SIMD 命令向け並列化」，あるいは,「SIMD 並列化」と呼ぶことにする．本研究で用いた COINS では，SIMD 並列化を機械寄りの中間表現における変換として実装している．

図 5.12 の例の場合，SIMD 命令最適化系がベクタ化を備えている場合は `(B)` が SIMD 命令適用の対象となり，静的な解析や動的なポインタの値の解析を行えば `(A)` も対象となる．SIMD 命令最適化系が SIMD 命令向け並列化のみである場合は，`(A)` や `(B)` はそのままでは SIMD 命令適用の対象とはならず，予めソースレベルあるいは中間言語レベルで `(C)` や `(D)` のように展開しておく必要がある．しかし，コンパイラ内の中間言語の設計に依存することではあるが，例えば COINS の低水準中間言語では，`(D)` と `(E)` は同じ表現になるので，`(E)` のようなプログラムに対しても SIMD 命令を適用できる．

```
  unsigned char a[], b[];
  int c = 0, i = 0;
.....
/* 定義 1：元のループ */
  for (;i < M; i++)
    if (a[i] > b[i]) c++;
----------------------------------------
/* 定義 2：SIMD 命令向けに展開
   スカラ拡張を施し，c と t の更新の頻度を低減 */
  int lc = 0;
  for (c = 0, i = 0; i < M - 7; i+=8) {
    /* 8 展開 */
    t[0] += (a[i+0] > b[i+0]);
    ....
    t[7] += (a[i+7] > b[i+7]);
    if (lc == 255) {
      /* unsigned char :0..255 */
      c += t[0] + t[1] ... + t[7];
      t[0:7] = 0; lc = 0;
    } else lc++;
  }
  c += t[0] + t[1] ... + t[7];
  /* SIMD レジスタに満たないデータの処理
     定義 1 そのまま */
  for (;i < M; i++)
    if (a[i] > b[i]) c++;
```

図 5.11: 条件に合う件数を数える例題

5.2.4　SIMD 最適化の関連研究

Larsen ら [44] では，SLP（Superword Level Paralelism）という概念を提起している．これは，SIMD 命令程度の規模と内容の並列性を意味し，ベクタ並列性やループ並列性,（超並列機レベルの）SIMD 並列性，そして命令レベル並列性と並ぶものだとしている．並列性を取り出す糸口として，図 5.12 の (E) の例のような同型な演算の発見と，メモリ参照の連続性を挙げ，プログラムからの SIMD 命令向けの並列性抽出方式について議論している．SUIF コンパイラ・インフラストラクチャ [30] を使った実装の結果を示しており，ベクタ並列に基づく最適化よりも良い結果を得ている．しかし，第 7 章で述べる最適処理データサイズの解析については言及していない．

Sreraman ら [65] は，SUIF を使って Intel MMX 命令セット向けのベクタ化コンパイラを実装し，評価している．SWIF に元々備わっているファシリティ以外に，ループ切断（strip mining）やスカ

```
#define AVE(x,y) (((x)>>1)+((y)>>1)+(((x)|(y))&1))
struct {
  short r, g, b, a;
} *u1, *u2, *u3;

  short *v1, *v2, *v3;

  for (i = 0; i < M; i++)            // (A)
    *v1++ = AVE(*v2++, *v3++);
  for (i = 0; i < M; i++)            // (B)
    v1[i] = AVE(v2[i], v3[i]);
  for (i = 0; i < M; i += 4) {       // (C)
    v1[i] = AVE(v2[i], v3[i]);       v1[i+1] = AVE(v2[i+1], v3[i+1]);
    v1[i+2] = AVE(v2[i+2], v3[i+2]); v1[i+3] = AVE(v2[i+3], v3[i+3]); }
  for (i = 0; i < M - 3; i += 4) {   // (D)
    v1[0] = AVE(v2[0], v3[0]); v1[1] = AVE(v2[1], v3[1]);
    v1[2] = AVE(v2[2], v3[2]); v1[3] = AVE(v2[3], v3[3]);
    v1 += 4; v2 += 4; v3 += 4; }
  for (i = 0; i < M; i++) {          // (E)
    u1[i].r = AVE(u2[i].r, u3[i].r); u1[i].g = AVE(u2[i].g, u3[i].g);
    u1[i].b = AVE(u2[i].b, u3[i].b); u1[i].a = AVE(u2[i].a, u3[i].a); }
```

図 5.12: SIMD 最適化可能なループの例

ラ拡張（scalar expansion）等を実装し，手書きのコードの 85%程度の性能にまで迫っている．この論文では，処理データサイズについて議論しているが，最適処理データサイズの解析については言及していない．

Leupers ら [47][48] は，演算パターンと SIMD 命令とのマッチング方式を提案している．CLP（Constraint Logic Programming）あるいは ILP（Integer Linear Programming）を用いるコード選択法を提案し実装している．この方法を使えば，汎用レジスタと SIMD 命令用レジスタ間でデータが行き来する場合も想定した最適コードの選択が可能である．しかし，通常はそのようにして無理に SIMD 命令を生成すると，通常命令で処理する場合に比べて，かえって遅くなってしまう．また，演算の最適サイズの解析や選択には言及していない．

Bik ら [10] は，例えば Intel Compiler（icc）等の商用のコンパイラにおける SIMD 命令向け最適化について紹介している．ベクタ化を構成する基本技法の各々を施された結果として，どのようなコードに変換されるのかを詳細に説明しているが，それを得るための方法については触れていない．またその中で，紹介されているコンパイル例の中には，演算に右シフトを含むような 最適サイズの解析を必要とする例が紹介されているが，同じサイズのデータ同士の限られた演算の場合にとどまっている．

Fisher ら [21] は，いくつかの SIMD 命令セットの内容を検討し，不足する演算を補う方法を議論している．演算の最適サイズの調査が必要であることは指摘しているが，その具体的な解決法については言及していない．

Stephenson ら [69] は，第 7 章で述べる解析法と類似の手法でハードウエア設計における演算器の最適サイズの解析を提案しているが，解析の推論規則の詳細には言及しておらず，また値域の表現方式が第 7 章で述べる方式で `size` が無限大である場合だけであるので，キャストがかけられた式の値域を正確に追跡することができない．また，SIMD 命令のコード生成にも応用可能であるとしている

が，具体的な適用法については言及していない．

5.3　SIMD 命令向けベンチマーク

この節では，第 8 章で議論する SIMD 命令向けベンチマークプログラム集の概要を述べ，関連研究を紹介する．以下，これを単に「SIMD ベンチマーク」と呼ぶことにする．

5.3.1　SIMD ベンチマークの設計要件

多くのアプリケーションやベンチマークのプログラムにおいて，そのままで SIMD 命令の活用による高速化可能な部位は比較的限定的で，SIMD 命令を適用可能な部位をプログラム毎に解析しなければならない．従来のベンチマークでは，SIMD 最適化や SIMD 命令の性能が，それら以外の最適化の効果に埋もれてしまって，正当な判断を行うことができない．そこで，SIMD 命令向けのベンチマークプログラム集を提案する．

SIMD ベンチマークの設計要件を簡単にまとめると，以下の通りである．

- コンパイラの設計者に対して，最適化の設計目標を示す．
- コンパイラのユーザに対して，SIMD 命令向けのコーディングパターンを示す．
- コンパイラの SIMD 命令の誤用を検出できる．

5.3.2　SIMD ベンチマークの関連研究

SPEC ベンチマーク [66] は，コンパイラやハードウエアの性能評価の標準として広く用いられ，調査項目も整数演算や浮動小数点演算から Web サーバに至るまでの多岐に渡っている．しかし，本論文で話題にしている SIMD 命令セットに関連したベンチマークは特には用意されておらず，最も関連がある SPEC CPU でも，実働のアプリケーションをそのまま例題にしているため，上記設計要件に挙げた情報をユーザに与えない．

MediaBench[46] は，例題の出典をメディア処理やファイル圧縮等のプログラムに絞ったベンチマークである．例題採取の方向性から SPEC に比べると SIMD 命令を活用できる可能性は高いが，SPEC と同様の構成方法であるため，そのままでは設計要件を満足する情報を与えない．

DSPStone[81] は，DSP による処理対象に的を絞ったベンチマークで，本論文で挙げている設計要件のいくつかを満たしているが，対象命令セットが SIMD 命令と少し異なる DSP 命令である点と，例題の変形やループの展開法に多様性がなく，誤った最適化やコード生成の検出のための考慮がない．

NpBench[45] や NetBench[52] 等のネットワーク処理を対象としたベンチマークは，命令レベル並列性やスレッドレベル並列性が要求される例題から成っており，SIMD 命令が対象とするデータ並列性を有する例題には主眼が置かれていない．

Dhrystone[78] は，コンパイラと計算機の性能の総合的な評価を目的としたベンチマークで，さまざまな言語で実現されている．その構成方法は，本論文のベンチマークと似ているが，対象が通常命令セットやオペレーティングシステムの入出力処理の性能であり，SIMD 命令向け最適化のテストにはならない．

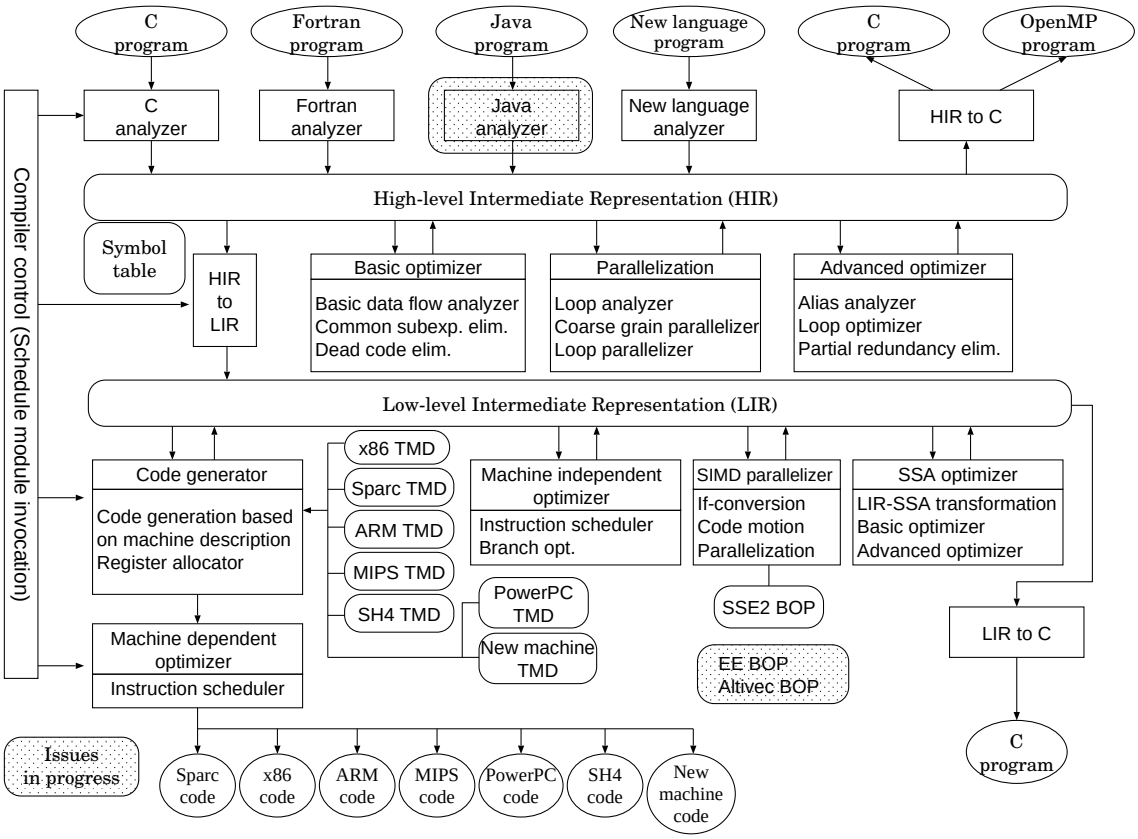


図 5.13: COINS 共通インフラストラクチャの構成

Windows 系オペレーティングシステムについては，SIMD 命令の性能評価も含む多くのベンチマークが公表されているが，それらはソースレベルで提供されていないので，コンパイラの調整やユーザのコーディングパターンの選択には利用できない．

5.4 COINS コンパイラ・インフラストラクチャ

第 6 章と第 7 章の実現に使用した COINS （a COmpiler INfraStructure）コンパイラインフラストラクチャ[15][60] は，文部科学省科学技術振興調整費に基づいて，2000〜2004 年の 5 年間に渡って実施された課題「並列化コンパイラ向け共通インフラストラクチャの研究」で作られたコンパイラの共通インフラストラクチャである．COINS は Java で記述され，オブジェクト指向を駆使して図 5.13 のようにモジュール化されており，機能や最適化の仕組みの新規追加や改良を容易に行うことができる．また，入力言語の変更が容易で，現在は入力言語として C と Fortran を受け付けるようになっており，さらに電気通信大学の情報工学実験第一の実験課題向けに抽象構文木生成系 ¬ <>< ∪ ∪ [42] との組み合わせて TinyC[77] 用が作成されたといった実績がある．さらに，TMD（Target Machine Description）の記述によって，ターゲットマシンの変更も容易になっており，現在は Sparc，IA-32（x86），ARM，MIPS，PowerPC，SH4，Alpha の各々の命令セットアーキテクチャに向けたリターゲッティングが完成している．

```
int a;
void func() {
  int b;
  a = b + a; /* This statement */
}
```

(a) C のソースコード

```
(assign   9 int line 4
 <var      10 int a>
 (add      11 int
  <var      12 int b>
  <var      13 int a>))
```

(b) HIR による表現

```
(SET I32 (MEM I32 (STATIC I32 "a"))
 (ADD I32 (MEM I32 (FRAME I32 "b.1"))
          (MEM I32 (STATIC I32 "a"))))
```

(c) LIR による表現

図 5.14: HIR と LIR における変数参照の表現

第 6 章で議論する方式の説明では，5.4.2 節で説明する COINS の低水準中間表現である LIR と，5.4.3 節で説明するコード生成部のマシン記述である TMD を用いている．また第 7 章でも，LIR の図表現を用いて，アルゴリズムを説明している．そこで，この節では，COINS の高水準中間表現である HIR も含めた 2 つの中間表現と，コード生成部について説明する．

5.4.1　高水準中間表現—HIR

フロントエンドでは，入力されたソースコードの字句解析と構文解析を行ない，高水準中間表現 HIR（High-level Intermediate Representation）に変換する．HIR は多くの手続き型プログラミング言語の抽象構文木（Abstract Syntax Tree，AST）に容易に対応付けられる表現である [59].

HIR に変換されたプログラムは，ループやスイッチといった文や，配列や構造体，共用体，COMMON といったデータ構造についての，ソースコードレベルの情報を保っている．HIR レベルでの変数への参照は抽象的に扱われており，シンボル（名前）に対応する記憶空間上のレイアウトは，シンボルに対応する属性として扱われている．例えば，参照している変数が，動的変数（あるいはローカル変数）であっても静的変数（あるいはグローバル変数）であっても，HIR では図 5.14 の `(a)` と `(b)` のように同じ形で表される．どの記憶クラスに属するか等の情報は，記号表に記録されている．

HIR レベルの解析系や最適化系は，このようなソースレベルの情報を利用することができる．ベクタ化を含むループ解析・変換・最適化は，このレベルに適した仕事の例である．これらを終えると，

通常は HIR は LIR に変換されるが，HIR 部のデバッグのために C 言語に変換されたり，並列化後に OpenMP のコードに変換される場合もある．

ループ並列化系と SMP 並列化系は，HIR レベルで実装されており，現在は，OpenMP のディレクティブを埋め込まれた C のソースコードを出力する．

5.4.2 低水準中間表現—LIR

COINS の低水準中間表現 LIR（Low-level Intermediate Representation）は抽象マシンを表現していて，表示的意味論が厳格に定義されている [1][54]．LIR のデータ型は，その型（整数か浮動小数点数）でのデータサイズを表しているに過ぎない．つまり，LIR を独立したプログラミング言語として取り扱うことが可能である．LIR では，低水準（あるいはマシンに近いレベル）での解析と最適化が行われる．静的単一代入（SSA，Static Single Assignment）形式の最適化や，分岐最適化がその例である．

LIR は L 式から構成される．第 6 章と第 7 章では，S 式風のテキストを L 式の表現に用いる．L 式のプリミティブの抜粋を表 5.3 に示す．オリジナルの LIR では，論理値の「真」を 1,「偽」を 0 として表しているが，第 6 章では –1（全てのビットが 1）と 0 を使用する．

LIR に翻訳されたプログラムには，ソースコードレベルの情報は，行番号程度しか残っていない．シンボルに対応する情報は，LIR それ自身が表している．例えば図 5.14 の例のように，変数からの参照が動的変数か静的変数かによって，LIR の形は異なる．

5.4.3 TMD によるコード生成部

最後にコード生成部で，ターゲットマシンの命令が iburg[23] 風の木文法生成系によって生成される [55]．COINS では，機種依存の TMD（Target Machine Description）に LIR のパターンと機械語のパターンの対応付けを記述することで，コード生成系を作ることができる．現在は，Sparc，IA-32（x86），ARM，MIPS，PowerPC，それに SH4 の TMD が用意されている．

図 5.15 に，IA-32 向けの TMD 記述からの抜粋を示す．TMD コンパイラが，TMD を Java のソースコードに変換する．これには，最小コストのコードを選択するために使われる動的計画法（Dynamic Programming，DP）マッチングの表が含まれる．複数のターゲットマシン中から一つを，COINS のコンパイラドライバのオプションで指定することができる．

5.5 第 6 章と第 7 章と第 8 章で解決する問題

第 6 章では，通常の言語仕様の範囲内で SIMD 命令の適用向けに変形されたプログラムに対して，適切な SIMD 命令を生成する方式を示し，実装に基づいた実験結果に基づく評価を行う．

第 7 章では，高級言語が必ず備える汎整数拡張を行った場合と同じ実行結果を得られ，しかも SIMD 命令を有効に活用するための，処理データサイズの解析方式を示し，実験結果を示し，評価を行う．

第 8 章では，既存のベンチマークではカバーできていない，コンパイラによる SIMD 命令の有効活用や，SIMD 命令の実行性能をに焦点を定めたベンチマーク集を提案し，実装結果とそれによる実験結果を示し，評価を行う．

表 5.3: L 式の意味 （抜粋）

プリミティブ	意味
$(INTCONST\ t\ z)$	型 t の整数値 z
$(FLOATCONST\ f\ r)$	型 f の浮動小数点値 r
$(STATIC\ t\ s)$	ラベル s で指定される記憶番地
$(FRAME\ t\ s)$	フレームポインタとラベル s で指定される記憶番地
$(LABEL\ s)$	s で指定される記憶番地
$(REG\ t\ s)$	型 t としてのレジスタ s の値
$(SUBREG\ t\ x\ n)$	型 t でレジスタ s を分割した時の n 番目のサブレジスタの値
$(NEG\ t\ x)$	$-x$
$(ADD\ t\ x\ y)$	$x+y$
$(SUB\ t\ x\ y)$	$x-y$
$(MUL\ t\ x\ y)$	$x \times y$
$(CONVSX\ t\ x)$	x を符号拡張した値
$(CONVZX\ t\ x)$	x をゼロ拡張した値
$(CONVIT\ t\ x)$	x を型 t に縮退した値
$(BAND\ t\ x\ y)$	x と y の論理積
$(BOR\ t\ x\ y)$	x と y の論理和
$(BNOT\ t\ x)$	x の 1 の補数 (ビット毎の反転)
$(TSTxxS\ t\ x\ y)$	符号つき整数として比較，xx: $NE \rightarrow x \neq y$, $LE \rightarrow x \leq y, ...$
$(TSTxxU\ t\ x\ y)$	符号なし整数として比較，xx: $GE \rightarrow x >= y$, $LT \rightarrow x < y, ...$
$(MEM\ t\ x)$	型 t としてみた番地 x で指定される記憶の値
$(SET\ x\ y)$	y の値を x に代入する．x は MEM 式，REG 式，もしくは $SUBREG$ 式のいずれかでなければならない．
$(JUMPC\ c\ l_1\ l_2)$	もし c が真なら l_1 に飛ぶ，さもなくば l_2 に飛ぶ．
$(PARALLEL\ x_1 \cdots x_n)$	x_1, x_2,...x_n を並列に実行する．x_i の各々は SET 式でなければならない．
$(DEFLABEL\ s)$	記憶番地にラベル s を対応付ける

t: 整数データの型 （$I8$, $I16$, $I32$, $I64$, $I128$ のうちのいずれか）

f: 浮動小数点データの型 （$F32$, $F64$, $F128$ のうちのいずれか）

s: 文字列

c: 真偽値

5.6　この章のまとめ

この章では，先ず SIMD 命令の説明を行ない，SIMD 命令向けコンパイラ最適化が実施すべき事項を挙げた．次に，SIMD 命令向けコンパイラ最適化という問題を，SIMD 命令向けベクタ化と SIMD 命令向け並列化の 2 つの問題に切り分け，第 6 章で説明する最適化の内容のあらましを述べ，関連する従来研究を紹介した．そして，従来のベンチマークプログラムが，SIMD 命令活用という観点から

```
; Real-registers
(def *real-reg-symtab*
  (SYMTAB
    ;; general registers
    (foreach @g (edxeax ebxecx ediesi)
      ("%@g" REG I64 4 0))
    (foreach @g (eax ecx edx ebx esi edi)
      ("%@g" REG I32 4 0))
    (foreach @g (al ah cl ch dl dh bl bh)
      ("%@g" REG I8 4 0))
...
    ("%ebp" REG I32 4 0) ; frame pointer
    ("%esp" REG I32 4 0) ; stack pointer
))

; TMD can describe dependencies between
; registers.
(def (REG I32 "%eax")
  (SUBREG I32 (REG I64 "%edxeax") 0))

(def (REG I16 "%ax")
  (SUBREG I16 (REG I32 "%eax") 0))

(def (REG I8 "%al")
  (SUBREG I8 (REG I16 "%ax") 0))
...
```

```
|; Same operations/different data-sizes
|(foreach (@op @code)((NEG negl)(BNOT notl))
|  (defrule regl (@op I32 regl)
|    (eqreg $1 $0)
|    (code (@code $0))
|    (cost 2)))
|
|(defrule regq (NEG I64 regq)
|  (eqreg $1 $0)
|  (code (negl (qlow $0))
|        (adcl (imm 0)(qhigh $0))
|        (negl (qhigh $0)))
|  (cost 2))
|
|; Same operations/different shift amount
|(foreach (@op @code)
|  ((LSHS sall)(RSHS sarl)(RSHU shrl))
|  (defrule regl (@op I32 regl con)
|    (eqreg $1 $0)
|    (code (@code (imm $2) $0))
|    (cost 2)))
|
|(foreach (@op @code)
|  ((LSHS sall)(RSHS sarl)(RSHU shrl))
|  (defrule regl (@op I32 regl shfct)
|    (eqreg $1 $0)
|    (code (@code "%cl" $0))
|    (cost 1)))
```

図 5.15: IA-32 向け TMD の例 （抜粋）

は適しておらず，それに的を絞ったベンチマークプログラムが必要であることを指摘し，既存のベンチマークプログラムを紹介し，それらが SIMD 命令活用には適さないことを示した．さらに，本研究で SIMD 最適化を実装した COINS コンパイラ・インフラストラクチャについて説明した．

第6章　SIMD並列化

本章では，本研究で提案し実装したSIMD並列化の方式について議論する．

6.1　最適化の基本方針

本研究は，SIMD最適化を段階的に進めていくという方針に基づいている．具体的には，以下の方針に基づいて研究を進めていった．

- ベクタ型の宣言や演算子といった特別な言語仕様を導入しない．SIMD並列化の対象は，通常の言語規格の範囲内で記述されたものとする．
- ループ展開や複雑なif文の引き剥がしといったベクトル化関連の最適化を行わない．
- SIMD並列化系を，LIRからLIRへの変換系として実装する．

ベクトル化や複雑なif文の引き剥がしは，現実のコードに対してSIMD最適化を行う場合には重要な事項ではあるが，これらの多くはソースコードレベルで実施できる．このような変換や最適化はCOINSのポリシに従えば，HIRレベルで行うべき事項である．これらは，本研究の対象外とした．

SIMD並列化系は，SIMD並列化を行うか否かを，副プログラム（つまり関数）単位で，自動的に決定する．

例えば，現在のCOINSのSIMD並列化系は，図5.12の（A）と（B）に対しては，SIMD命令を生成できないが，（C）と（D）と（E）に対しては生成できる．SIMD並列化をベクタ並列化の結果として行うコンパイラでは，（E）に対してはSIMD命令を生成できないかもしれない．

先にも触れたようにここでは，SIMD命令セットの比較命令では，比較の種類（<，<=，＝等）が命令で決まり，全ビットが1の値を真，0の値を偽として生成するということを仮定する．つまり，図5.2の2番目の例のように，ビットマスクを生成すると仮定する．ほとんどのSIMD命令セットでは，このスタイルが採用されている．この2つの値は，−1と0という数値としても使うことができるが，本研究で実装したif変換アルゴリズムでは，これを活用する．この論理値に対する意味論は，オリジナルのLIRのそれとは異なるので，SIMD並列化フェーズでのみ用いている．

6.2　SIMD並列化の目標

ここではSIMD並列化がどのようなコード最適化を行うかを示す．

図6.1に示すコードをSIMD並列化の例として用いる．このコードは，Gouraud shading [27]の最内側ループと等価な働きをしている．図6.1の例題のループカーネルがLIRに変換されると，図6.2のようになる．

```
static unsigned char sa,sr,sg,sb;
static short da,dr,dg,db;
static short k;

static void
hLineRight(unsigned char *p,int n,
           unsigned char a, unsigned char r, unsigned char g, unsigned char b,
           short ea, short er, short eg, short eb) {
  while(n!=0) {
    p[0]=b; p[1]=g; p[2]=r; p[3]=a;
    a+=sa; r+=sr; g+=sg; b+=sb;
    if((ea+=da)>=0) { a++; ea-=k; } if((er+=dr)>=0) { r++; er-=k; }
    if((eg+=dg)>=0) { g++; eg-=k; } if((eb+=db)>=0) { b++; eb-=k; }
    --n; p+=4; }}
```

図 6.1: コンピュータグラフィクスからの例題

最適化系が図 6.3 のようなコードを生成するためには，例えば，コンパイラがベクタ化以外の方法で，オペランドが記憶空間上に連続して置かれていることを発見しなければならず，あるいは，“`a++; r++; g++; b++;`” に対しては，比較結果の値を $-1/0$ の数値としてうまく使えることが必要である．

6.3　SIMD 並列化アルゴリズム

SIMD 並列化は，次の段階から成る．

1. if 変換
2. DAG（Directed Acyclic Graph）への変換
3. データサイズ推論（第 7 章で詳説する.）
4. DAG から SIMD 命令へのマッチング
5. 並列化
6. SIMD レジスタの割り当て

以下，データサイズ推論を除くそれぞれの段階について，説明していく．

6.3.1　if 変換

LIR は if 変換されて，図 6.4 の最初の部分に示すような分岐不要な LIR に変換される．行末に `M`, `N`, ⋯ といった文字を，変換の段階間でどの部位に該当するかを表すために付け加えてある．実装した if 変換では，比較演算の結果を単にマスクとしてだけではなく，$-1/0$ の数値としても活用する．

```
(SET I8 (MEM I8 (ADD I32 (REG I32 "p.1%_1")(INTCONST I32 0)))(REG I8 t0)) M
(SET I8 (MEM I8 (ADD I32 (REG I32 "p.1%_1")(INTCONST I32 1)))(REG I8 t1)) N
(SET I8 (MEM I8 (ADD I32 (REG I32 "p.1%_1")(INTCONST I32 2)))(REG I8 t2)) O
(SET I8 (MEM I8 (ADD I32 (REG I32 "p.1%_1")(INTCONST I32 3)))(REG I8 t3)) P
(SET (REG I8 t0)(ADD I8 (REG I8 t0)(REG I8 t4)))         ; a+=sa          M
(SET (REG I8 t1)(ADD I8 (REG I8 t1)(REG I8 t5)))         ; r+=sr          N
(SET (REG I8 t2)(ADD I8 (REG I8 t2)(REG I8 t6)))         ; g+=sg          O
(SET (REG I8 t3)(ADD I8 (REG I8 t3)(REG I8 t7)))         ; b+=sb          P
(SET (REG I16 t8)(ADD I16 (REG I16 t8)(REG I16 t12)))    ; ea+=da         M
(JUMPC (TSTGE I1 (REG I16 t8)(INTCONST I16 0))(LABEL L0)(LABEL L4))       M
(DEFLABEL L0)                                                             M
  (SET (REG I8 t0)(ADD I8 (REG I8 t0)(INTCONST I8 1)))   ; a++            M
  (SET (REG I16 t8)(ADD I16 (REG I16 t8)(REG I16 t28)))  ; ea-=k          M
(DEFLABEL L4)                                                             M
(SET (REG I16 t9)(ADD I16 (REG I16 t9)(REG I16 t13)))    ; er+=dr         N
(JUMPC (TSTGE I1 (REG I16 t9)(INTCONST I16 0))(LABEL L1)(LABEL L5))       N
(DEFLABEL L1)                                                             N
  (SET (REG I8 t1)(ADD I8 (REG I8 t1)(INTCONST I8 1)))   ; r++            N
  (SET (REG I16 t9)(ADD I16 (REG I16 t9)(REG I16 t28)))  ; er-=k          N
(DEFLABEL L5)                                                             N
(SET (REG I16 t10)(ADD I16 (REG I16 t10)(REG I16 t14)))  ; eg+=dg         O
(JUMPC (TSTGE I1 (REG I16 t10)(INTCONST I16 0))(LABEL L2)(LABEL L6))      O
(DEFLABEL L2)                                                             O
  (SET (REG I8 t2)(ADD I8 (REG I8 t2)(INTCONST I8 1)))   ; g++            O
  (SET (REG I16 t10)(ADD I16 (REG I16 t10)(REG I16 t28))); rg-=k          O
(DEFLABEL L6)                                                             O
(SET (REG I16 t11)(ADD I16 (REG I16 t11)(REG I16 t15)))  ; eb+=eb         P
(JUMPC (TSTGE I1 (REG I16 t11)(INTCONST I16 0))(LABEL L3)(LABEL L7))      P
(DEFLABEL L3)                                                             P
  (SET (REG I8 t3)(ADD I8 (REG I8 t3)(INTCONST I8 1)))   ; b++            P
  (SET (REG I16 t11)(ADD I16 (REG I16 t11)(REG I16 t28))); eb-=k          P
(DEFLABEL L7)                                                             P
```

図 6.2: 図 6.1 のループカーネルに対する LIR

if 変換の候補となるためには，if 文の then 部と else 部について，図 6.5 の (a) のように，単一の基本ブロックでなければならない．(b) の形も，点線で囲った部分が if 変換可能であれば，if 変換可能である．SIMD 並列化系では，if 変換を行う入れ子のレベルの最大値を指定する定数（= 2）を指定するようになっている．

6.3.2 DAG への変換

ここでの DAG 化では，式を構成している木を演算毎に切断し，ノード間に中間変数（LIR では仮想レジスタ）への代入と参照の形に変換している．これはコンパイラ最適化の一般的技法で，共通部分式の発見や，オペレーションのマッチングを容易にしている．

LIR の表現上はバラバラにされているが，仮想レジスタの定義ー参照によって縫い合わされている．この段階では，図 6.4 の 3 番目の部分の LIR に示すような大きな式が，図 6.6 の複雑な SIMD 命令を

```
_hLineRight:                                |
    pushl   %ebp                            |    movw    _k,%ax
    movl    %esp,%ebp                       |    movd    %eax,%mm4
    subl    $24,%esp                        |    pshufw  $0,%mm4,%mm4   #mm4:{k,k,k,k}
    pushl   %ebx                            |    movq    -20(%ebp),%mm2 #mm2:{ea,er,eg,eb}
    pushl   %esi                            |    cmpl    $0,%ecx
    pushl   %edi                            |    je      .L37
    movl    8(%ebp),%esi  #p                |.L22:
    movl    12(%ebp),%ecx #n                |    movq    %mm1,%mm7
    movb    16(%ebp),%al                    |    packuswb %mm7,%mm7
    movb    %al,-4(%ebp)  #a                |    movd    %mm7,(%esi)   #p[0:3]={a,r,g,b}
    movb    20(%ebp),%al                    |    paddw   %mm3,%mm1     #{a,r,g,b}+={sa,sr,sg,sb}
    movb    %al,-3(%ebp)  #r                |    paddw   _da,%mm2   #{ea,er,eg,eb}+={da,dr,dg,db}
    movb    24(%ebp),%al                    |    movq    %mm2,%mm7
    movb    %al,-2(%ebp)  #g                |    movq    %mm2,%mm6
    movb    28(%ebp),%al                    |    pcmpgtw %mm0,%mm7  #mm7={ea,er,eg,eb}>{0,0,0,0}
    movb    %al,-1(%ebp)  #b                |    pcmpeqw %mm0,%mm6  #mm6={ea,er,eg,eb}=={0,0,0,0}
    movw    32(%ebp),%di                    |    por     %mm6,%mm7  #mm7={ea,er,eg,eb}>={0,0,0,0}
    movw    %di,-20(%ebp) #sa               |    psubw   %mm7,%mm1  #{a,r,g,b}++ if TRUE
    movw    36(%ebp),%di                    |    movq    %mm4,%mm6  #{k,k,k,k}
    movw    %di,-18(%ebp) #sr               |    pand    %mm7,%mm6  #mm6=k if TRUE, 0 if FALSE
    movw    40(%ebp),%di                    |    psubw   %mm6,%mm2  #{ea,er,eg,eb}-={k,k,k,k}
    movw    %di,-16(%ebp) #sg               |    leal    4(%esi),%esi
    movw    44(%ebp),%di                    |    decl    %ecx
    movw    %di,-14(%ebp) #sb               |    cmp     $0,%ecx
    pxor    %mm0,%mm0     #mm0:{0,0,0,0}    |    jne     .L22
    movd    _sa,%mm3                        |.L37:
    punpcklbw %mm0,%mm3   #mm3:{sa,sr,sg,sb}|    popl    %edi
    movd    -4(%ebp),%mm1                   |    popl    %esi
    punpcklbw %mm0,%mm1   #mm1:{a,r,g,b}    |    popl    %ebx
                                            |    leave
```

図 6.3: 図 6.1 の例題に対して目標とするコード生成（命令セット：IA-32/MMX）

```
(SET (REG I16 t8)(ADD I16 (REG I16 t8)(REG I16 t12)))     M
(SET (REG I16 t16)                                        M
     (TSTGE I16 (REG I16 t8)(INTCONST I16 0)))            M
(SET (REG I8 t0)                                          M
     (ADD I8 (REG I8 t0)                                  M
             (NEG I8 (CONVIT I8 (REG I16 t16)))))         M
              ↑ TSTGE 命令の結果を-1/0 として利用
(SET (REG I16 t8)                                         M
     (ADD I16 (REG I16 t8)                                M
              (BAND I16 (REG I16 t28)(REG I16 t16))))     M
```

図 6.4: SIMD 並列化の第 1 段階 (if 変換) 後

マッチできるようになる．例えば，3 番目の式の値は，レジスタ t32 を介して 4 番目の SET 式に運ばれる．

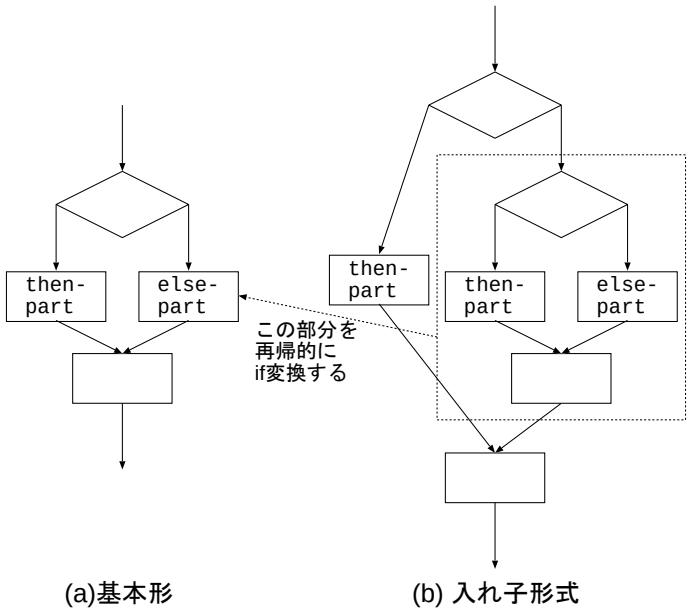


図 6.5: if 変換可能な if 文の形

```
(SET (REG I16 t8)(ADD I16 (REG I16 t8)(REG I16 t12)))     M
(SET (REG I16 t16)                                        M
     (TSTGE I16 (REG I16 t8)(INTCONST I16 0)))            M
(SET (REG I8 t32)(CONVIT I8 (REG I16 t16)))               M
(SET (REG I8 t0)   ;matched to SUB                        M
     (ADD I8 (REG I8 t0)(NEG I8 (REG I8 t32))))           M
(SET (REG I16 t31)(BAND I16 (REG I16 t28)(REG I16 t16)))  M
(SET (REG I16 t8)(ADD I16 (REG I16 t8)(REG I16 t31)))     M
```

図 6.6: SIMD 並列化の第 2 段階 (DAG 化) 後

6.3.3 DAG から SIMD 命令の演算内容へのマッチング

BOP（Basic OPerator）と呼ばれるテンプレートの表を用意し，これを用いて LIR のパターンとターゲットマシンの SIMD 命令演算内容とのマッチングを行う．

図 6.7 に BOP の例を示す．SIMD 並列化系が，BOP の表のエントリの順に従って，DAG とのマッチングを行っていく．表には SIMD 命令のサブレジスタ間の演算内容が登録されている．これには，データサイズ推論による解析結果も関係するが，処理データサイズ関連のマッチングの詳細は，7.2.4 節で述べる．

図 5.2 のような単純な演算から，図 5.9 の右側や図 5.10 のような複雑な演算まであるが，適用された場合の性能向上が期待される複雑な演算を先に登録する．複数の BOP にマッチする L 式には，新しいレジスタを用意してそれに値を書くように SET 式を埋め込む．BOP の HOLE 式の数値は，その BOP に渡ってくるオペランドの序数を表す．DAG のマッチしなかった部分は，通常命令に翻訳される．この段階でマッチしない部分が大きい場合は，それを含む副プログラムに対する SIMD 並列化

```
; 2 つの値のうちの最大値を求める演算
(BOR I8
  (BAND I8 (HOLE 1 I8)(TSTGES I8 (HOLE 1 I8)(HOLE 2 I8)))
  (BAND I8 (HOLE 2 I8)
           (BNOT I8 (TSTGES I8 (HOLE 1 I8)(HOLE 2 I8)))))
; 2 つの値の平均値を求める演算 (符号つき)
(RSHS I8
  (ADD I8
    (ADD I8 (HOLE 1 I8)(HOLE 2 I8))
    (INTCONST I8 1))
  (INTCONST I8 1))
```

図 6.7: BOP の例

を諦め，通常命令の生成に切り替える．

6.3.4　並列化

SIMD 並列化系は図 6.8 に示すように，異なるデータに対して，同じ演算を行っている SET 式を発見し，それらを寄せ集めて，PARALLEL 式で括る．PARALLEL 式は，表 5.3 で説明したように，並列に実行してもかまわない，つまりどの順序でも実行できる複数の SET 式を括る L 式である．SIMD 並列化系は，後述する BONE パターンの表を使って，複数の SET 式を寄せ集め，ひとつの PARALLEL 式にまとめる．

連続した記憶位置を参照している SET 式の集合が，この括りの最初の候補である．例えば，図 6.2 の最初の 4 つの SET 式は，ポインタ p に定数 $0, 1, 2, 3$ を加算したアドレスのメモリを参照しているので，PARALLEL 式で一括りにする候補となる．SET 式は，記憶参照の読み込みと書き出しが整序するような順番で，PARALLEL 式の中に入れられる．図 6.2 の例では，BONE でマッチされ，PARALLEL 式で括られた SET 式のうちで，t0，t1，t2，t3 を参照しているものが最初に取り上げられ，SET 式の順番をその順番に並べ替える．この順番は，レジスタの定義/参照情報（DU 鎖，Define/Use Chain）を使って，他の PARALLEL 式の中の SET 式の順番を決めるように波及させていく．

この BONE はひとつの SIMD 命令の振る舞いを丸ごと表している．BONE パターンは，演算の並列度，レジスタ間での可換性，それにレジスタの制約を表す部分（BONE 情報）と，対象 SIMD 命令の演算パターンを表す部分から成る．表 6.1 に BONE 情報の内容を，図 6.9 に BONE パターンの例として図 5.2 の 2 つの SIMD 命令と，図 5.3 演算に対応する BONE パターン（命令セットは IA-32/SSE2）を示す．図 6.7 の 1 番目の例と，図 6.9 の 3 番目の例を比べると判るように，演算パターンの部分は，ほぼ BOP 情報と同じであるが，結果の副作用を指定するために，ターゲットの HOLE を含む SET 式で囲まれている．BONE 情報の Para の値の回数だけ演算パターンの部分が複製され，それを PARALLEL 式で囲んだものが，実際の LIR とのマッチングに使われる．BONE パターンを用いて，ターゲットマシンが 2 アドレスと 3 アドレスのどちらを採用しているのかということや，図 5.5 のようなシャッフル命令の動作や，図 5.10 のような複雑な命令の動作を記述する．例えば，ターゲットマシンが 3 アドレスならば，図の OutH の箇所は，0 になる．この段階では，PARALLEL 式

```
(PARALLRL
 (SET I8 (MEM I8 (ADD I32 (REG I32 "p.1%_1")(INTCONST I32 0)))(REG I8 t0))  M
 (SET I8 (MEM I8 (ADD I32 (REG I32 "p.1%_1")(INTCONST I32 1)))(REG I8 t1))  N
 (SET I8 (MEM I8 (ADD I32 (REG I32 "p.1%_1")(INTCONST I32 2)))(REG I8 t2))  O
 (SET I8 (MEM I8 (ADD I32 (REG I32 "p.1%_1")(INTCONST I32 3)))(REG I8 t3))) P
(PARALLEL
 (SET (REG I8 t0)(ADD I8 (REG I8 t0)(REG I8 t4)))                           M
 (SET (REG I8 t1)(ADD I8 (REG I8 t1)(REG I8 t5)))                           N
 (SET (REG I8 t2)(ADD I8 (REG I8 t2)(REG I8 t6)))                           O
 (SET (REG I8 t3)(ADD I8 (REG I8 t3)(REG I8 t7))))                          P
(PARALLEL
 (SET (REG I16 t8)(ADD I16 (REG I16 t8)(REG I16 t12)))                      M
 (SET (REG I16 t9)(ADD I16 (REG I16 t9)(REG I16 t13)))                      N
 (SET (REG I16 t10)(ADD I16 (REG I16 t10)(REG I16 t14)))                    O
 (SET (REG I16 t11)(ADD I16 (REG I16 t11)(REG I16 t15))))                   P
(PARALLEL
 (SET (REG I16 t16)(TSTGE I16 (REG I16 t8)(INTCONST I16 0)))                M
 (SET (REG I16 t17)(TSTGE I16 (REG I16 t9)(INTCONST I16 0)))                N
 (SET (REG I16 t18)(TSTGE I16 (REG I16 t10)(INTCONST I16 0)))               O
 (SET (REG I16 t19)(TSTGE I16 (REG I16 t11)(INTCONST I16 0))))              P
(PARALLEL
 (SET (REG I8 t32)(CONVIT I8 (REG I16 t16)))                                M
 (SET (REG I8 t34)(CONVIT I8 (REG I16 t17)))                                N
 (SET (REG I8 t36)(CONVIT I8 (REG I16 t18)))                                O
 (SET (REG I8 t38)(CONVIT I8 (REG I16 t19))))                               P
(PARALLEL
 (SET (REG I8 t0)(SUB I8 (REG I8 t0)(REG I8 t32)))                          M
 (SET (REG I8 t1)(SUB I8 (REG I8 t1)(REG I8 t34)))                          N
 (SET (REG I8 t2)(SUB I8 (REG I8 t2)(REG I8 t36)))                          O
 (SET (REG I8 t3)(SUB I8 (REG I8 t3)(REG I8 t38))))                         P
(PARALLEL
 (SET (REG I16 t31)(BAND I16 (REG I16 t28)(REG I16 t16)))                   M
 (SET (REG I16 t33)(BAND I16 (REG I16 t28)(REG I16 t17)))                   N
 (SET (REG I16 t35)(BAND I16 (REG I16 t28)(REG I16 t18)))                   O
 (SET (REG I16 t37)(BAND I16 (REG I16 t28)(REG I16 t19))))                  P
(PARALLEL
 (SET (REG I16 t8)(SUB I16 (REG I16 t8)(REG I16 t31)))                      M
 (SET (REG I16 t9)(SUB I16 (REG I16 t9)(REG I16 t33)))                      N
 (SET (REG I16 t10)(SUB I16 (REG I16 t10)(REG I16 t35)))                    O
 (SET (REG I16 t11)(SUB I16 (REG I16 t11)(REG I16 t37))))                   P
```

図 6.8: 並列化された LIR

で括られた SET 式を移動する時は，制御フロー情報を使ってプログラムの意味が変更されないようにする．

現在は BOP パターンや BONE パターンと LIR とのマッチングには，ハッシュ等の技法を用いていないので，非常に処理時間がかかっている．この点には，改良の余地がある．

表 6.1: BONE 情報の内容

フィールド名	形式	意味
Para	整数のリスト	PARALLEL 式に含まれる L 式の個数，つまり並列度.
OutH	整数	結果を出力する HOLE の番号.
Comm	真偽	HOLE 1 と HOLE 2 は交換可能（可換）である.
Repl	整数	書き換え規則番号
UnUs	–	＜現在未使用＞
ShaH	整数	PARALLEL 式でまとめられたとき，各式に共通なレジスタに対応する HOLE の番号
NoRg	整数	レジスタに置き換えることを禁止する HOLE の番号
Subg	整数のリストのリスト	レジスタのグルーピングを指定する.

注 1：整数を置く箇所に nil を設定すると，該当するものが無いことを意味する.
注 2：「整数のリスト」になっている箇所は，複数の候補を並べられることを意味する.
注 3：各フィールドは表の順番に従って並べられた S 式表現をとる．途中から省略されると，以降のフィールドには規定値 nil が使われる.

```
% IA-32/SSE2 の PADDB 命令用 BONE パターン
((16 8) % Para 16 又は 8 並列
 1      % OutH IA-32 は 2 アドレスマシンなので
 t)     % Comm 加算は可換な演算
(SET I8 % 以下は演算パターン
 (HOLE 0 I8)
 (ADD I8
  (HOLE 1 I8)
  (HOLE 2 I8)))

% IA-32/SSE2 の PCMPGTB 命令用 BONE パターン
((8 4)   % Para 8 又は 4 並列
 1       % OutH IA-32 は 2 アドレスマシンなので
 n)      % Comm 大小比較は非可換な演算
(SET I16 % 以下は演算パターン
 (HOLE 0 I16)
 (TSTGTS I16
  (HOLE 1 I16)
  (HOLE 2 I16)))
```

```
% IA-32/SSE2 の PMAXSW 命令用の BONE パターン
  ((8 4)   % Para
   1       % OutH
   nil)    % Comm 大小比較と同様に非可換な演算
  (SET I16 % 以下は演算パターン
    (HOLE 0 I16)
    (BOR I16
     (BAND I16
      (HOLE 1 I16)
      (TSTGES I16
       (HOLE 1 I16)
        (HOLE 2 I16)))
     (BAND I16
     (HOLE 2 I16)
     (BNOT I16
       (TSTGES I16
        (HOLE 1 I16)
        (HOLE 2 I16))))))
```

図 6.9: BONE パターンの例

6.3.5　SIMD レジスタの割り当て

SIMD 並列化系は，図 6.10 に示すように，BONE パターンの情報からの制約を用いて，ベクタレジスタを各々の PARALLEL 式に割り当てていく.

```
(PARALLRL
 (SET (SUBREG I8 (MEM I32 (REG I32 "p.1%_1")) 0)(SUBREG I8 (REG I32 m0) 0))                          M
 (SET (SUBREG I8 (MEM I32 (REG I32 "p.1%_1")) 1)(SUBREG I8 (REG I32 m0) 1))                          N
 (SET (SUBREG I8 (MEM I32 (REG I32 "p.1%_1")) 2)(SUBREG I8 (REG I32 m0) 2))                          O
 (SET (SUBREG I8 (MEM I32 (REG I32 "p.1%_1")) 3)(SUBREG I8 (REG I32 m0) 3)))                         P
(PARALLEL
 (SET (SUBREG I8 (REG I32 m0) 0)(ADD I8 (SUBREG I8 (REG I32 m0) 0)(SUBREG I8 (REG I32 m1) 0)))      M
 (SET (SUBREG I8 (REG I32 m0) 1)(ADD I8 (SUBREG I8 (REG I32 m0) 1)(SUBREG I8 (REG I32 m1) 1)))      N
 (SET (SUBREG I8 (REG I32 m0) 2)(ADD I8 (SUBREG I8 (REG I32 m0) 2)(SUBREG I8 (REG I32 m1) 2)))      O
 (SET (SUBREG I8 (REG I32 m0) 3)(ADD I8 (SUBREG I8 (REG I32 m0) 3)(SUBREG I8 (REG I32 m1) 3))))     P
(PARALLEL
 (SET (SUBREG I16 (REG I64 m2) 0)(ADD I16 (SUBREG I16 (REG I64 m2) 0)(SUBREG I16 (REG I64 m3) 0)))  M
 (SET (SUBREG I16 (REG I64 m2) 1)(ADD I16 (SUBREG I16 (REG I64 m2) 1)(SUBREG I16 (REG I64 m3) 1)))  N
 (SET (SUBREG I16 (REG I64 m2) 2)(ADD I16 (SUBREG I16 (REG I64 m2) 2)(SUBREG I16 (REG I64 m3) 2)))  O
 (SET (SUBREG I16 (REG I64 m2) 3)(ADD I16 (SUBREG I16 (REG I64 m2) 3)(SUBREG I16 (REG I64 m3) 3)))) P
(PARALLEL
 (SET (SUBREG I16 (REG I64 m4) 0)(TSTGES I16 (SUBREG I16 (REG I64 m2) 0)(INTCONST I16 0)))          M
 (SET (SUBREG I16 (REG I64 m4) 1)(TSTGES I16 (SUBREG I16 (REG I64 m2) 1)(INTCONST I16 0)))          N
 (SET (SUBREG I16 (REG I64 m4) 2)(TSTGES I16 (SUBREG I16 (REG I64 m2) 2)(INTCONST I16 0)))          O
 (SET (SUBREG I16 (REG I64 m4) 3)(TSTGES I16 (SUBREG I16 (REG I64 m2) 3)(INTCONST I16 0))))         P
(PARALLEL
 (SET (SUBREG I8 (REG I32 m5) 0)(CONVIT I8 (SUBREG I16 (REG I64 m4) 0)))                             M
 (SET (SUBREG I8 (REG I32 m5) 1)(CONVIT I8 (SUBREG I16 (REG I64 m4) 1)))                             N
 (SET (SUBREG I8 (REG I32 m5) 2)(CONVIT I8 (SUBREG I16 (REG I64 m4) 2)))                             O
 (SET (SUBREG I8 (REG I32 m5) 3)(CONVIT I8 (SUBREG I16 (REG I64 m4) 3))))                            P
(PARALLEL
 (SET (SUBREG I8 (REG I32 m0) 0)(SUB I8 (SUBREG I8 (REG I32 m0) 0)(SUBREG I8 (REG I32 m5) 0)))      M
 (SET (SUBREG I8 (REG I32 m0) 1)(SUB I8 (SUBREG I8 (REG I32 m0) 1)(SUBREG I8 (REG I32 m5) 1)))      N
 (SET (SUBREG I8 (REG I32 m0) 2)(SUB I8 (SUBREG I8 (REG I32 m0) 2)(SUBREG I8 (REG I32 m5) 2)))      O
 (SET (SUBREG I8 (REG I32 m0) 3)(SUB I8 (SUBREG I8 (REG I32 m0) 3)(SUBREG I8 (REG I32 m5) 3))))     P
(PARALLEL
 (SET (SUBREG I16 (REG I64 m4) 0)(BAND I16 (SUBREG I16 (REG I64 m4) 0)(SUBREG (REG I64 m7) 0)))     M
 (SET (SUBREG I16 (REG I64 m4) 1)(BAND I16 (SUBREG I16 (REG I64 m4) 1)(SUBREG (REG I64 m7) 1)))     N
 (SET (SUBREG I16 (REG I64 m4) 2)(BAND I16 (SUBREG I16 (REG I64 m4) 2)(SUBREG (REG I64 m7) 2)))     O
 (SET (SUBREG I16 (REG I64 m4) 3)(BAND I16 (SUBREG I16 (REG I64 m4) 3)(SUBREG (REG I64 m7) 3))))    P
(PARALLEL
 (SET (SUBREG I16 (REG I64 m2) 0)(SUB I16 (SUBREG I16 (REG I64 m2) 0)(SUBREG I16 (REG I64 m4) 0)))  M
 (SET (SUBREG I16 (REG I64 m2) 1)(SUB I16 (SUBREG I16 (REG I64 m2) 1)(SUBREG I16 (REG I64 m4) 1)))  N
 (SET (SUBREG I16 (REG I64 m2) 2)(SUB I16 (SUBREG I16 (REG I64 m2) 2)(SUBREG I16 (REG I64 m4) 2)))  O
 (SET (SUBREG I16 (REG I64 m2) 3)(SUB I16 (SUBREG I16 (REG I64 m2) 3)(SUBREG I16 (REG I64 m4) 3)))) P
```

図 6.10: 図 6.6 の LIR に対する SIMD レジスタの割り当て

6.4 SIMD コード生成

PARALLRL 式と SIMD 命令の対応付けも，図 6.11 に示すように TMD 記述の中で行う．現状の COINS の実装では，SIMD 並列化以外の最適化は，PARALLRL 式無視するようになっている．

6.5 実験と評価

現在，基本ブロック単位で SIMD 並列化が行えるようになっており，第 8 章で議論するベンチマークプログラムの図 6.12 左側のような簡単な例題に対して，右側のような最適化された SIMD 命令を生成する．表 6.2 にその結果を示す．

命令セットアーキテクチャは IA-32/SSE2 で，PentiumM と Pentium4（Northwood）の機種で測定した．SIMD 並列化の対象となる基本ブロックの実行にかかるクロック数を，`rdtsc`（read time-stamp counter）命令を関数の入り口と出口に埋め込み，その間の所要クロック数を関数の値として返すように計装し，呼び出し側で表示するようにして，測定した．この計装は COINS や gcc の関数の出入

```
;Add/Subtract Packed Byte Integers, and Logical AND/OR/XOR
(foreach (@cd @op)(
 (paddb ADD)(psubb SUB)(pand BAND)(por BOR)(pxor BXOR))
 (defrule void
  (PARALLEL
   (foreach @i (0 1 2 3 4 5 6 7)
    (SET I8
     (SUBREG I8 regm (INTCONST I32 @i))
     (@op I8
      (SUBREG I8 regm (INTCONST I32 @i))
      (SUBREG I8 regm (INTCONST I32 @i))))))
  (code (@cd $3 $2))
  (cost 3)))

;Packed Compare for Equal/Greater-Than Byte
(foreach (@cc @fn)((EQ eq)(GTS gt))
 (defrule void
  (PARALLEL
   (foreach @i (0 1 2 3 4 5 6 7)
    (SET I8 (SUBREG I8 regm (INTCONST I32 @i))
     (BOR I8
       (BAND I8 con
        (TST@cc I8 (SUBREG I8 regm (INTCONST I32 @i))
                   (SUBREG I8 regm (INTCONST I32 @i))))
       (BAND I8 con
        (BNOT I8
         (TST@cc I8
          (SUBREG I8 regm (INTCONST I32 @i))
          (SUBREG I8 regm (INTCONST I32 @i)))))))))
  (cond "(((LirIconst)$2).signedValue() == 255 ||
           ((LirIconst)$2).signedValue() == -1) &&
           ((LirIconst)$5).signedValue() == 0")
  (code (pcmp@fnb $4 $3))
  (cost 3)))
```

図 6.11: SIMD 命令に対する TMD の例 (x86.tmd からの抜粋)

り口のコード生成の仕様に基づき，awk スクリプトで実装した．COINS 向けの計装を行うスクリプトを図 6.13 に，生成したコードへの計装例を図 6.14 に示す．

例題は，すべて 2 つのベクタ間の演算結果を別のベクタに格納する形のものである．SIMD レジスタとして，64 ビットに対応する例題と，128 ビットに対応する例題が用意されており，同じ処理データサイズについて並列度が異なるものがある．

キャッシュミスや分岐予測器の影響を除外するために，2 回目の実行の際にかかったクロック数をサンプルした．比較の対象は COINS で最適化のための指示が無い場合である．

gcc（ver 4.0，`-O2`）に対しても同様の手法で計装を行ない，所要クロック数を測定したところ，COINS の最適化なしの場合とほぼ同じであった．また，gcc で SIMD 並列化を指示しても，これらの例題では SIMD 命令を生成しなかった．このように COINS では，最適化の指示が無くても gcc の `-O2` 程度の最適化を行うので，この比較は妥当であると考える．

Intel の icc に対しては，関数の出入り口のコード生成の仕様が不明であったために，上記の手法

```
#define MAX(x,y)((x>=y)? x: y)
void pcalcsub(short *a,short *b,short *c) {
  c[0]=MAX(a[0], b[0]);
  c[1]=MAX(a[1], b[1]);
  c[2]=MAX(a[2], b[2]);
  c[3]=MAX(a[3], b[3]);                          ⇒
  return;
}
void pcalcsub(short *a,short *b,short *c) {
  short a1,a2,a3,a4;
  short b1,b2,b3,b4;
  short c1,c2,c3,c4;
  a1 = a[0]; a2 = a[1]; a3 = a[2]; a4 = a[3]; ⇒
  b1 = b[0]; b2 = b[1]; b3 = b[2]; b4 = b[3];
  c1=MAX(a1,b1); c4=MAX(a4,b4);
  c3=MAX(a3,b3); c2=MAX(a2,b2);
  c[0]=c1; c[1]=c2; c[2]=c3; c[3]=c4;
  return;
}
```

```
                .section .text
                .align  4
                .global pcalcsub
pcalcsub:
                pushl   %ebp
                movl    %esp,%ebp
                movl    8(%ebp),%edx
                movl    12(%ebp),%ecx
                movl    16(%ebp),%eax
                movq    (%edx),%MM1
                movq    (%ecx),%MM0
                pmaxsw  %MM0,%MM1
                movq    %MM1,(%eax)
.L15:
                leave
                ret
```

図 6.12: コード生成例（IA-32/SSE2）

表 6.2: SIMD 並列化の効果

例題 （処理データサイズ×並列度）	PentiumM			Pentium4		
	S	N	N/S	S	N	N/S
加算（8 × 16）	56	86	1.54	92	140	1.52
加算（8 × 8）	52	72	1.38	84	116	1.33
加算（32 × 4）	56	61	1.09	92	92	1.00
加算（32 × 2）	66	58	0.88	100	92	0.92
平均（8 × 8）	52	89	1.54	84	120	1.43
平均（16 × 4）	52	67	1.29	84	100	1.19
最小値（16 × 4）	52	131	2.52	84	284	3.38
積の下位（16 × 8）	58	117	2.02	92	152	1.65

S：SIMD 並列化した場合の処理クロック数
N：最適化なしの場合の処理クロック数

で rdtsc 命令を計装することができなかった．

演算が同じで SIMD レジスタが同じサイズならば，処理データサイズが小さいほど並列度が上がり SIMD 並列化の効果が大きくなる．32 ビットの演算では，SIMD 並列化を行うと逆に遅くなる場合もある．その理由は，通常命令がスーパスカラにより数命令が同時実行されているのに対し，SIMD 命令にはスーパスカラ技術が適用されていないためであると考えられる．しかし，最小値を求める演算のように，分岐が必要な演算や，積のように結果を得るまでの遅延が大きい演算では，SIMD 並列化の効果が比較的高いことが判る．

6.6　この章の結論

コンパイラのマシン寄り中間表現から並列実行可能な部分を抽出し，SIMD 命令に変換する方式を提案・実装し，基本例題で有効性を確認した．提案方式では，SIMD 命令向きに記述されたプログラムに対して，SIMD 命令を適切に適用することに主眼を置いている．他のコンパイラが行っているようなベクトル化は行っていないが，ループが適切に展開されたプログラムや，構造体のメンバで同型の演算を行うものに対しては，SIMD 命令を生成できる．

実装した SIMD 並列化の適用範囲を広げるには，SIMD 命令向きのベクトル化の実装が必須であるが，これを HIR を用いて実装することは，将来の課題として残された．また，BOP や BONE の SIMD 命令対応の TMD からの自動生成も，解決すべき課題のひとつである．さらに，SIMD 並列化系は，SIMD 並列化を行うか否かを，副プログラム（つまり関数）単位で，自動的に決定するとしたが，将来的には，ループやブロックといったより細かいレベルで行うようにすべきである．

```
$1 == "leave" {
  print "\trdtsc\n\tpopl\t%ecx\n\tsubl\t%ecx,%eax\n";
  print $0;
  next;}
$1 == "movl" && $2 == "%esp,%ebp" {
  t1 = $0; getline; t2 = $0;
  if ($1 == "subl" && $2 ~ /,%esp/) {
    print t1; print t2;
    print "\trdtsc\n\tpushl\t%eax\n";
  } else {
    print t1;
    print "\trdtsc\n\tpushl\t%eax\n";
    print t2; }
  next; }
{ print $0; next; }
```

図 6.13: rdtsc 命令を計装するための awk スクリプト (COINS 用)

```
/*         計装前              */  /*         計装後              */
        .section .text                      .section .text
        .align  4                           .align  4
        .global pcalcsub                    .global pcalcsub
pcalcsub:                           pcalcsub:
        pushl   %ebp                        pushl   %ebp
        movl    %esp,%ebp                   movl    %esp,%ebp
                                            rdtsc               /* ここから */
                                            pushl   %eax
        pushl   %ebx                        pushl   %ebx
        movl    8(%ebp),%ebx                movl    8(%ebp),%ebx
        movl    12(%ebp),%edx               movl    12(%ebp),%edx
        movl    16(%ebp),%eax               movl    16(%ebp),%eax
        movw    $1,%cx                      movw    $1,%cx
        movq    (%edx),%MM1                 movq    (%edx),%MM1
        movq    (%ebx),%MM0                 movq    (%ebx),%MM0
        movd    %ecx,%MM7                   movd    %ecx,%MM7
        psraw   %MM7,%MM0                   psraw   %MM7,%MM0
        movd    %ecx,%MM7                   movd    %ecx,%MM7
        psraw   %MM7,%MM1                   psraw   %MM7,%MM1
        pminsw  %MM1,%MM0                   pminsw  %MM1,%MM0
        movq    %MM0,(%eax)                 movq    %MM0,(%eax)
.L15:                               .L15:
        popl    %ebx                        popl    %ebx       /* ここまでを */
                                            rdtsc              /* 測定する   */
                                            popl    %ecx
                                            subl    %ecx,%eax
        leave                               leave
        ret                                 ret
```

図 6.14: rdtsc 命令の計装例

第7章　SIMD命令適用のための最適な処理データサイズ

この章では，与えられたプログラムに対して，適切な処理データ幅 M の SIMD 命令を適用したコード生成を行うためのプログラム解析法について議論する．

7.1　SIMD命令セットと汎整数拡張

5.1 節でも議論したように，演算のデータサイズ M を小さくまとめることができれば，並列度を高めて実行効率を上げることができる．しかし，C 言語等の標準仕様として定められている「汎整数拡張」の仕様（integral promotion rules）を遵守したときと同じ結果を得るように，M の値をどこまで小さくしても同じ結果を得られるかを調べなければならない．

汎整数拡張とは，処理系が規定している int 型（多くの場合は 32 ビット整数，アーキテクチャによっては 64 ビットの場合もある.）より小さいサイズの整数型変数を参照する際に，符号拡張やゼロ拡張を施して int 型のサイズにし，それを演算で用いることである．そして，演算結果を小さいサイズの整数型変数に書き出す場合は，結果の上位ビットの値は捨てられ，下位ビットから書き出し先のサイズの分だけ切り出して書き出す．汎整数拡張を行えば，int 型より小さいサイズのデータを演算に用いて，結果を小さいサイズの変数に書き戻す場合でも，多くの場合に演算の途中結果がラップアラウンドせずに済む．

しかし，汎整数拡張を SIMD 命令セットのコード生成で素直に実施すると，実際には SIMD 命令で処理可能なコードであるのに，コンパイラは（int 型のデータサイズを処理する SIMD 命令がなくて）適用不能と判断するか，M=32 として処理するので変数の参照時に常にサイズ変換命令を挿入することになり，サイズ変換のオーバヘッドや並列実行性能の低下を招く．また，レジスタプレッシャーも高まり，レジスタの溢れが起き易くなる．

ただし，文献 [10][44][48] の例のように，式を構成する演算の中に（例えば右シフトのように）上位ビットの値が結果の下位ビットに影響を与える演算を含まない場合は，代入先のデータサイズと演算のデータサイズを等しくすることができる．また，コンパイラに内蔵された，汎整数拡張を行う場合との互換性が検討済みである特定のパターンに演算がマッチした場合も，提案する手法を用いる場合と同様にコードを生成できる．しかし，この種の方法は柔軟性や一般性に欠ける．

例えば図 7.1 で `AVE` の定義を SIMD 命令向きの（`a`）とする場合の代入文　`*a=AVE(*b, *c)`　を考える．これに対しては図 7.2 のような L 式が生成される．以降は，テキスト表現の L 式の代わりに，処理データ幅の説明に適切な図 7.3 のような L 式の図表現を用いていく．

このコードは SIMD 命令向きに処理の全てが 16 ビットに収まるように工夫されているが，上のような演算を含むので，通常は図で（`CONVSX I32 ...`）と表されているような汎整数拡張が必要となり，32 ビットで処理するコードが生成される．

```
#define AVE(x,y) (((x)>>1)+((y)>>1)+(((x)|(y))&1)) // (a)
#define AVE(x,y) (((x)+(y)+1)>>1)                  // (b)
....
short *a, *b, *c;
for (...;i+=8) {
  a[0]=AVE(b[0],c[0]);a[1]=AVE(b[1],c[1]);
  ...
  a[6]=AVE(b[6],c[6]);a[7]=AVE(b[7],c[7]);
  a+=8; b+=8; c+=8;
}
```

図 7.1: 整数配列間の平均値を求めるプログラム例

```
(SET I16
 (MEM I16 (MEM I32 (STATIC I32 "a") &id ("a" 9)))
 (CONVIT I16
  (ADD I32
   (ADD I32
    (RSHS I32
      (CONVSX I32 (MEM I16 (MEM I32 (STATIC I32 "b") &id ("b" 1)) &id 2))
      (INTCONST I32 1))
    (RSHS I32
     (CONVSX I32 (MEM I16 (MEM I32 (STATIC I32"c") &id ("c" 3)) &id 4))
     (INTCONST I32 1)))
  (BAND I32
   (BOR I32
    (CONVSX I32 (MEM I16 (MEM I32 (STATIC I32 "b") &id ("b" 5)) &id 6))
    (CONVSX I32 (MEM I16 (MEM I32 (STATIC I32 "c") &id ("c" 7)) &id 8)))
   (INTCONST I32 1)))))
```

図 7.2: 図 7.1 の例題で (a) の場合の L 式

このように，SIMD 命令を活用して効率的なコードを得ようとしても，汎整数拡張がプログラマの意図の反映を阻害しているといえる．本章では，int 型よりも小さいデータサイズで演算を行っても，汎整数拡張を素直に行う場合と同じ結果を得るための解析である「データサイズ推論」についての議論を行う．以下の議論では，SIMD 命令向けに 中間言語に対して if 変換やベクタ化が施された形のものを取り扱い，同型の操作が複数並んだ様子をそのままではなく，1 つを取り出して図示することにする．

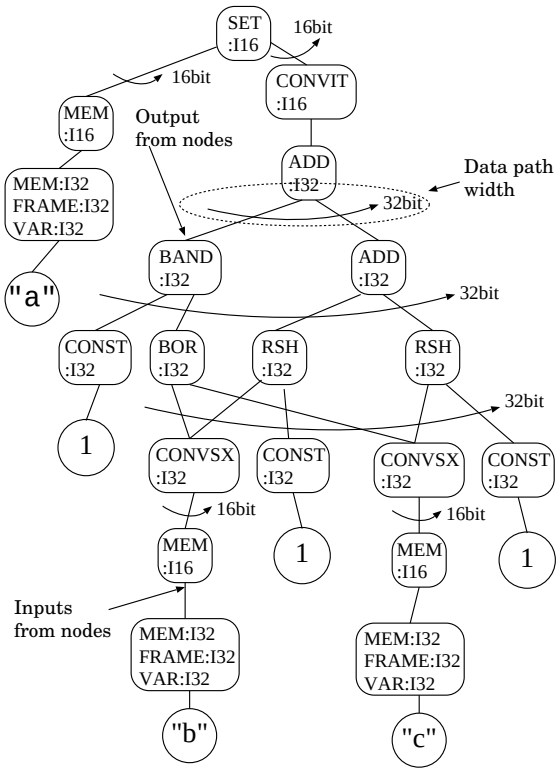


図 7.3: 図 7.1 の例題で（a）の場合の L 式の図表現

7.2 データサイズ推論

データサイズ推論は代入文単位で行い,「上向き解析」と「下向き解析」の 2 つの解析をこの順番で行う.

上向き解析では，参照している変数のサイズも含む型や定数の値から，途中結果のビット数の膨らみ，つまり，右辺値の計算を行うのにどれだけビット数で行えば，途中結果を溢れさせることなく演算を行うことができるかを解析する．各演算子毎に，上向き解析の推論規則が決められていて，それに従って各演算ノードに対応する途中結果のビット数の膨らみを推論する．下向き解析では，左辺値の変数のサイズから，必要とされているビットを知り，それと下向き解析の推論規則を元にして，各演算のノードで結果として必要とされているビットを調べる.

各演算ノードについて，上向き解析の結果として得られた値域に対応するビット列と，下向き解析の結果として得られたビット列の論理積を取り，下から何ビットが 1 であるかを調べることで，各演算ノードの適切な処理データサイズを知ることができる.

解析のために L ノードを表すデータ構造を拡張し，表 7.1 に示すフィールドを付加する.

表 7.1: L ノードのデータ構造の拡張

フィールド	意味
`size`	ノードの型（I32 等）に対応するビット幅
`up`	ノードが取り得る値の上界のビットパターン
`lo`	ノードが取り得る値の下界のビットパターン
`lv`	ノードの出力の有効ビット集合

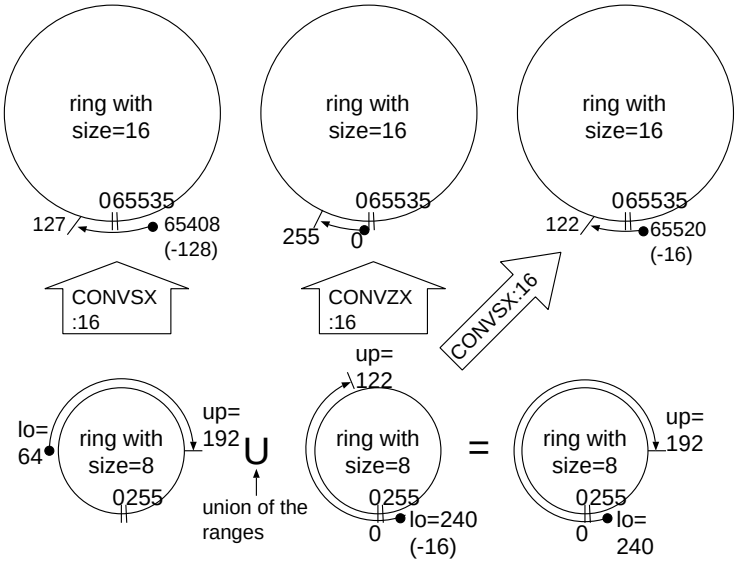


図 7.4: ノードがとり得る値の表現

7.2.1　上向き解析

先ず，右辺値の式を構成している木を深さ優先で辿って行きながら，各ノードでの演算結果の値域（上界と下界）を求める解析を行う．

ノードの取り得る値のビットパターンを $2^{\mathtt{size}}$ を法とする 2 進数と見なし，その値域を図 7.4 に示すように `lo` から始まり `up` に終わる連続した値と定義する．図では `lo` が黒丸，`up` が矢印の先に対応し，$2^{\mathtt{size}}$ の剰余系の上を時計回りで値が大きくなっていくように表現している．図の例のように，必ずしも $\mathtt{up} \geq \mathtt{lo}$（符号無し比較）の関係ではなく，逆の場合もある．図では，サイズ拡張処理に対する値域の設定や，値域の合併の例も示されている．

値域をデータサイズの剰余系として表現することによって，キャストを施された式の値域の追跡をラップアラウンドやシフトのはみ出しも含めて正確に行うことができる．

演算子によって，可換性や上位ビットと下位ビットの影響関係等が異なり，またノードの各入力が定数かどうかによっても推論される値域が異なる．演算子ごとに定めた解析方法を「上向き解析の推論規則」といい，表 7.2 に示す．

さらに IF ノードについては，図 7.5 の破線で囲った部分の構造に対するマッチングを行い飽和処理

表 7.2: 上向き解析の推論規則

L ノード	上向き解析の推論規則
$(NEG : t\ \ x)$	$\mathtt{lo} = -x.\mathtt{up}$, $\mathtt{up} = -x.\mathtt{lo}$
$(ADD : t\ \ x\ \ y)$	x の上下界の差と y の上下界の差が環のサイズより大きければ全域に渡る．そうでなければ，$\mathtt{lo} = x.\mathtt{lo} + y.\mathtt{lo}$, $up = x.up + y.up$.
$(SUB : t\ \ x\ \ y)$	x の上下界の差と y の上下界の差が環のサイズより大きければ全域に渡る．そうでなければ，$\mathtt{lo} = x.\mathtt{lo} - y.\mathtt{up}$, $\mathtt{up} = x.\mathtt{up} - y.\mathtt{lo}$.
$(MUL : t\ \ x\ \ y)$	符号つき整数として扱い，x と y を正と負の領域に分割し，4 通りの場合の値域の合併を求める．
$(CONVSX : t\ \ x)$	符号つきとして $x.\mathtt{lo} > x.\mathtt{up}$ なら，元のビット数の環の全域に渡るとして扱う．そうでなければ，上下界を符号拡張して新しい上下界とする．
$(CONVZX : t\ \ x)$	$x.\mathtt{lo} > x.\mathtt{up}$ のときは，元のビット数の環の全域に渡るとして扱う．そうでなければ，上下界をゼロ拡張して新しい上下界とする．
$(CONVIT : t\ \ x)$	上下界の差が縮退先のビット数の環のサイズより大きければ，全域に渡るとして扱う．
$(BAND : t\ \ x\ \ y)$, $(BXOR : t\ \ x\ \ y)$, $(BOR : t\ \ x\ \ y)$	x，y の値域が 0 を跨いでいる ($\mathtt{lo} > \mathtt{up}$) ときは，それぞれで値域を $0..\mathtt{up}$ と $\mathtt{lo}..2^{\mathtt{size}} - 1$ に分割し，文献 [32]4 章 3 節の方法で値域を求め，それらを合併する．
$(BNOT : t\ \ x)$	$\mathtt{lo} = \overline{x.\mathtt{up}}$, $\mathtt{up} = \overline{x.\mathtt{lo}}$ (ビット毎の反転)
$(LSH : t\ \ x\ \ y)$, $(RSHS : t\ \ x\ \ y)$, $(RSHU : t\ \ x\ \ y)$	x を y の値域のそれぞれの値でシフトしたときの値域の合併を求める．
$(TSTEQ : t\ \ x\ \ y)$	x と y が定数で同じ値なら $\mathtt{lo} = \mathtt{u}p = 2^{\mathtt{size}} - 1$．$x$ と y の値域が疎なら $\mathtt{l}o = \mathtt{up} = 0$．いずれでもなければ $\mathtt{lo} = 2^{\mathtt{size}} - 1$, $\mathtt{up} = 0$.
$(TSTLTS : t\ \ x\ \ y)$, $(TSTLTU : t\ \ x\ \ y)$	それぞれ符号つきとなしで比較して，x の値域が必ず y より小さければ $\mathtt{lo} = \mathtt{up} = 2^{\mathtt{size}} - 1$．$x$ の値域が必ず y 以上なら $\mathtt{lo} = \mathtt{up} = 0$．いずれでもなければ $\mathtt{lo} = 2^{\mathtt{size}} - 1$, $\mathtt{up} = 0$.
$(IF : t\ \ c\ \ x\ \ y)$	図 7.5 の構造マッチングに外れた場合，c が必ず真 (0 以外) なら x の値域，必ず偽 (0) なら y の値域．いずれでもなければ，x の値域と y の値域の合併.

を認識し，表 7.3 の推論規則に従って IF ノードの値域を推定する．表では符号無し比較（TSTLTU）の場合だけを示しているが，符号付き比較（TSTLTS）の場合は，表中の比較演算子「$<_u$」「$\leq_u$」等を符号つきのものに置き換えて適用する．符号付きの場合 $\mathtt{lo} >_s \mathtt{up}$ は値域が $2^{\mathtt{size}-1}$ を跨いでいることを意味する．

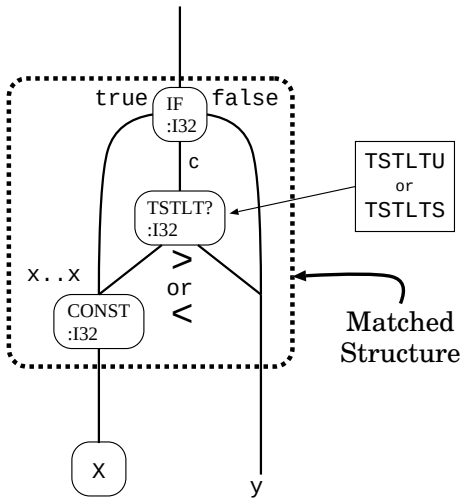


図 7.5: IF ノードの構造マッチング

表 7.3: 飽和処理の推論規則（符号無しの場合）

比較の方向	lo$\leq_u$up の場合			lo$>_u$up の場合
	$x<_u$lo	lo$\leq_u x\leq_u$up	up$<_u x$	（値域が 0 を跨ぐ）
$>$	lo..up	x..up	x	$x..2^{\mathtt{size}}-1$
$<$	x	lo..x	lo..up	0..x

$<_u$，$\leq_u$: 符号無し比較
lo $\equiv y$.lo，up $\equiv y$.up
推論した結果として値域が 2 つに分かれる場合は，2 つを覆う最小の値域をノードの値域としてある．(ex. up$< x$ のとき 0..up $\cup\, x$ を 0..x とする．)

変数を参照している MEM ノードでは size = 変数のサイズ，lo = 0，up = $2^{\mathtt{size}}-1$，つまり環の全域とする．この仮定の改良案については，7.5 節で触れる．
図 7.3 の L 式に対して，上向きの解析を行った結果を図 7.6 に示す.

7.2.2　下向き解析

次に，代入先のデータサイズで決まる左辺値の有効ビットの集合を元にして，木を下向きに辿って行きながら，上向き解析で得た値域が張る有効ビット集合（値域の値の全てを区別して表現できる最小限のビット数だけ 1 を最下位ビットから並べた集合）と，上から与えられた有効ビット集合をつき合わせていく「下向き解析」を行う．値域とそれが張る有効ビット集合の関係を表 7.4 に示す.
上から与えられる有効ビット集合とは，ノードの出力の中で上のノードが必要としているビットの

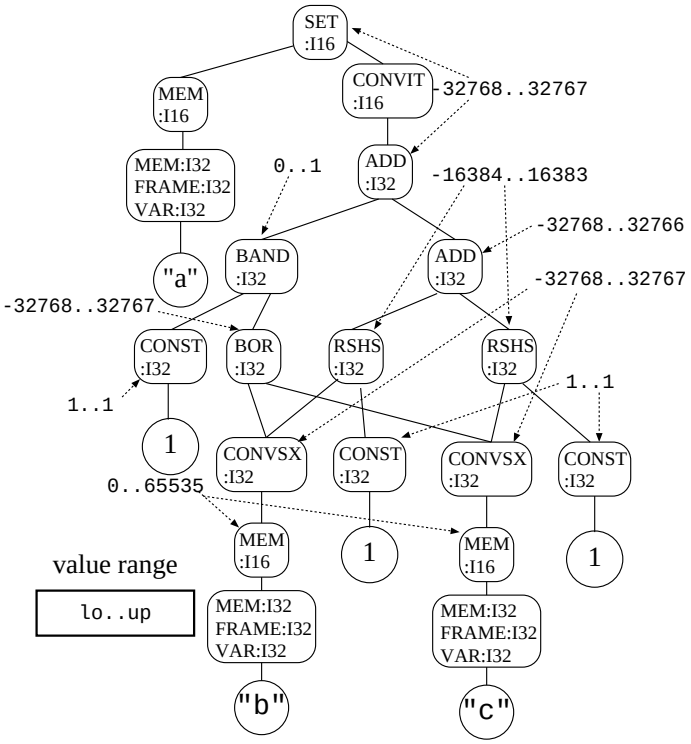


図 7.6: 図 7.3 に対する上向き解析の結果

表 7.4: 値域が張る有効ビット集合

値域	有効ビット集合
$0 \leq \mathtt{lo} \leq \mathtt{up} < 2^{\mathtt{size}-1}$	$cover(\mathtt{up})$
$0 \leq \mathtt{up} < \mathtt{lo} < 2^{\mathtt{size}-1}$	$2^{\mathtt{size}} - 1$（全域）
$2^{\mathtt{size}-1} \leq \mathtt{lo} \leq \mathtt{up} \leq 2^{\mathtt{size}} - 1$	$cover(\overline{\mathtt{lo}})$
$2^{\mathtt{size}-1} \leq \mathtt{up} < \mathtt{lo} \leq 2^{\mathtt{size}} - 1$	$2^{\mathtt{size}} - 1$（全域）
$2^{\mathtt{size}-1} \leq \mathtt{lo} \leq 2^{\mathtt{size}} - 1$ かつ $0 \leq \mathtt{up} < 2^{\mathtt{size}-1}$	$y = cover(\overline{\mathtt{lo}}) \mid cover(\mathtt{up})$ として $y \mid y << 1$
$2^{\mathtt{size}-1} \leq \mathtt{up} \leq 2^{\mathtt{size}} - 1$ かつ $0 \leq \mathtt{lo} < 2^{\mathtt{size}-1}$	$2^{\mathtt{size}} - 1$（全域）

$cover(x)$：x を最上位ビットからみて最初の 1 のビットから下を全て 1 にした値
（ex. $cover(\mathtt{0x0000ff00}) = \mathtt{0x0000ffff}$）
$\overline{x}$：x のビット毎の反転
その他演算子は C 言語の記法に従う．

集合，言い換えれば 上のノードの演算結果に影響を及ぼす可能性のあるビットの集合を意味する．下のノードに与える有効ビット集合は，表 7.5 の下向き解析の推論規則を用いて決定する．例えば，図 7.3 の SET ノードでは，`0x0000ffff` が左辺値の有効ビット集合となる．しかし，有効ビット集合は，必ずしも最下位ビットから始まるわけではない．例えば右シフト演算子のノードが下のノードに有効

表 7.5: 下向き解析の推論規則

L ノード	下向き解析の推論規則
$(NEG : t\ \ x)$	$cover(w)$ を x に伝える．
$(ADD : t\ x\ y)$, $(SUB : t\ \ x\ \ y)$	x も y も定数でなければ $cover(w)$ を x と y に伝える．どちらかが定数であるときは**注 1** の通り．
$(MUL : t\ \ x\ \ y)$	x，y のどちらも定数でないときは $cover(w)$ を x と y に伝える．x，y の片方 (例えば x) が 2 のべき乗の定数なら w を x で割った値を y に伝える．そうでなければ**注 2** の通り．
$(CONVSX : t\ \ x)$	x の最上位 (符号) ビットの位置を i とする．w の i より上のビットに 1 がなければ，w の i より下のビット．そうでなければ，w の i より下のビットの i ビットを 1 にした値を x に伝える．
$(CONVZX : t\ \ x)$	x の最上位ビットの位置を i として，w の i より下のビットを x に伝える．
$(BAND : t\ \ x\ \ y)$	x，y の片方 (例えば x) が定数であるときは $w \wedge x$ を y に，そうでなければ w を x と y に伝える．
$(BXOR : t\ \ x\ \ y)$	w を x と y に伝える．
$(BOR : t\ \ x\ \ y)$	x，y の片方 (例えば x) が定数であるときは $w \wedge \overline{x}$ を y に，そうでなければ w を x と y に伝える．
$(BNOT : t\ \ x)$	w を x に伝える．
$(LSH : t\ \ x\ \ y)$	y が定数の時は w を y ビット右にシフトした値を，そうでなければ $cover(w)$ を x に，全ビットを y に伝える．
$(RSHS : t\ \ x\ \ y)$	y が定数の時は w を y ビット左に符号残しシフトした値を，そうでなければ $over(w)$ を x に，全ビットを y に伝える．
$(RSHU : t\ \ x\ \ y)$	y が定数なら w を y ビット左にシフトした値を，そうでなければ $over(w)$ を x に伝える．
$(TSTcc : t\ \ x\ \ y)$	全ビットを x と y に伝える．$(cc \in EQ, LTS, LTU)$
$(IF : t\ \ c\ \ x\ \ y)$	w を x と y に，全ビットを c に伝える．

w： 上からきた有効ビット集合（図 7.8 の下向き解析のパラメタ）
他の単項演算子は w をそのまま x に伝える．
$over(w)$：w の最下位ビットから見て最初の 1 にビットより上を全て 1 にした値
（ex. $over($`0x0000ff00`$) =$ `0xffffff00`）
その他の記号は表 7.4 に同じ．

ビット集合を渡す場合，集合はシフト量の分だけ左に移動したものになる．このように，上から必要とされているビットは「最下位から何ビット」という形でなく，ビット集合の形で伝える．

図 7.6 で `AVE` の定義を（`a`）とするの場合に対応する L 式に対して，下向きの解析を行った結果を図 7.7 に示す．

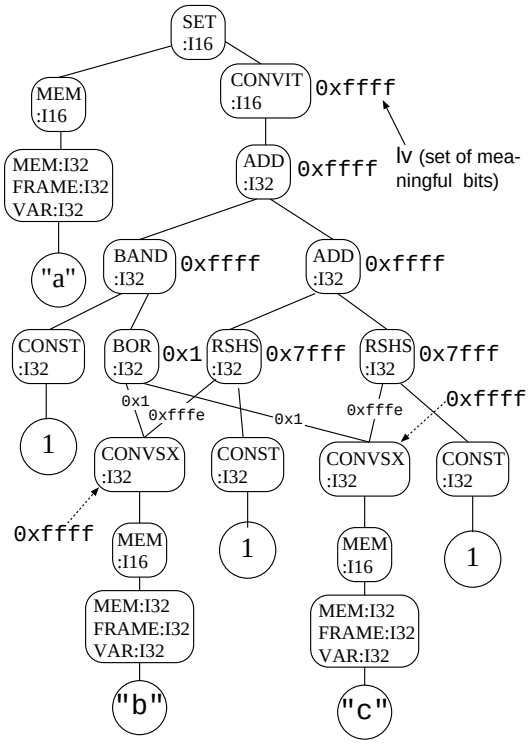


図 7.7: 図 7.6 に対する下向き解析の結果

7.2.3 アルゴリズム

データサイズ推論の概略を図 7.8 に示す．図 7.3 の CONVSX ノードのように，1 つのノードの出力が複数のノードの入力として共有される場合に，パスによって渡される有効ビットの集合の値が異なる場合がある．出力を共有するために，ノードの最終的な有効ビットの集合の値を，それらの合併集合にする（図 7.8 の最後の行の処理）．

7.2.4 推論結果とコード生成

推論結果とコード生成の連携について簡単に述べる．

推論の結果として，最適処理データサイズが SIMD 命令の処理データサイズで正規化しても広くなったり狭くなったりする場合がある．この場合 コード生成には（1）サイズが異なる箇所毎にサイズ拡張/縮退命令を挿入するする方針と，(2）最初から最大のサイズにサイズ拡張を行い，最後に代入先のサイズに合わせてサイズ変換を行う方針が挙げられる．(あるいは可能なサイズの全ての組み合わせを試して，実行時間が最短になるものを探すことも考えられる.）いずれの方針を採用するにせよ，式を構成する L ノードの lv の値によって，ノード間のデータ幅が決定される．

通常のコード生成では汎整数拡張後のデータの演算のみを考えればよいので，演算子の組み合わせと（シフト等の）量に注目してマッチングによって命令を生成する．一方 データサイズ推論の結果を用いるコード生成では，それに加えてノードの入出力のデータサイズと演算命令の処理データサイズ

```
代入式のサイズ推論 (top);
 top: SET ノード;
{
  値域 = 上向き解析 (右辺値のノード);
  下向き解析 (左辺値の有効ビット集合, 右辺値のノード);
  top.lv = 左辺値の有効ビット集合;
}

上向き解析 (n): 値域;
 n: ノード;
{
  n. 値域 =
    上向き解析の推論規則 (
      上向き解析 (n. 入力 1),
      上向き解析 (n. 入力 2),
      ....)
  return n. 値域;
}

下向き解析 (w, n)
 w: 上からの有効ビット集合;
 n: ノード;
{
  foreach (i in n. 入力) {
    下向き解析 (
      下向き解析の推論規則を使って他の入力の値域と w から求めた有効ビット集合,
      n. 入力 i)
  }
  /* n.lv の初期値は 0 */
  n.lv = n.lv | (w & n の値域が張る有効ビット);
}
```

図 7.8: データサイズ推論のアルゴリズム

とのマッチングも必要になる．そこで，6.3.3 節で説明した BOP パターンとのマッチング前に，BOP パターンの I8 や I16 といった L 式中の演算サイズ指定子を処理データサイズを表現する有効ビット集合に展開した形式に変換する．従って，ターゲットの命令の処理データ幅のバリエーションに応じて，同じ演算機能に対して複数の BOP パターンを用意する．

L 式と BOP パターンが演算機能でマッチングすると，処理データサイズが小さいものから順にノードの入出力の有効ビット集合のマッチングを行っていく．この順位付けは，6.3.3 節で述べたように

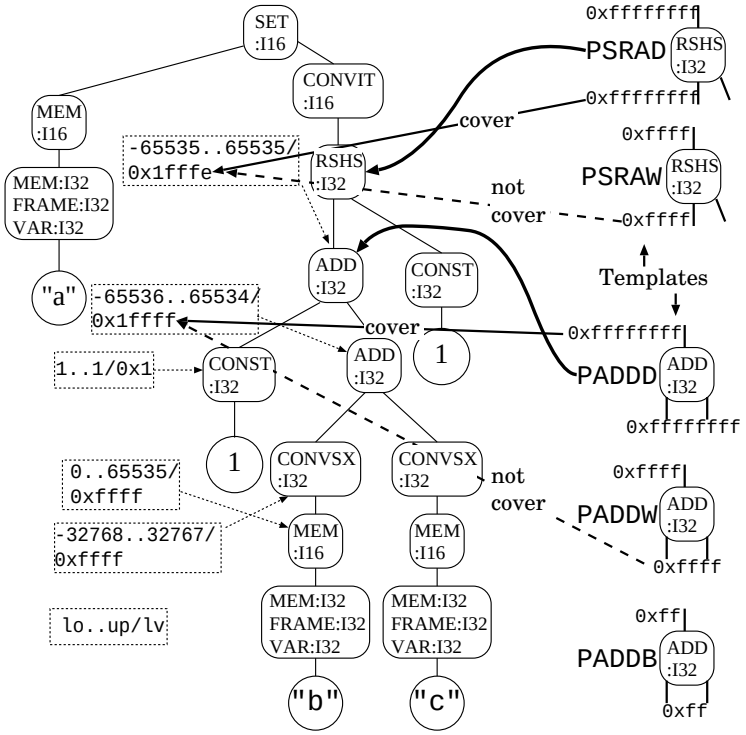


図 7.9: 図 7.1 の（b）の場合の推論結果への有効ビット集合マッチング

BOP の表のエントリ順で実現されている．有効ビット集合のマッチングの成功の条件は，ノード間の有効ビット集合が BOP パターンの有効ビット集合で覆われていること，つまり前者の有効ビット 1 であって後者の有効ビットが 0 であるものがないことである．ただし，ビット毎の演算（BAND，BOR，BNOT，BXOR）については処理データ幅のマッチングは不要である．

図 7.1 で AVE の定義を（b）とする場合の L 式にデータサイズ推論を行った結果に対して，BOP パターンをマッチングする様子を図 7.9 に示す．図の右側にある演算子ノードは，BOP パターンの演算サイズ指定子を有効ビット集合に変換した形式になっている．具体的な命令のニモニックは，IA-32/MMX の標記に従っている．

7.3 実装

上向きの解析の推論規則と下向きの解析の推論規則を，それぞれ値域と有効ビットの集合のクラスとメソッドとして，COINS の記述言語である Java で実装した．その内訳を表 7.6 に示す．

7.4 データサイズ推論の効果

簡単なプログラムから COINS コンパイラで生成した L 式を元にして，人手で IA-32 の MMX や SSE2 命令セットのコードを生成を行い，実行時間を計測した．使用した機材は Celeron 2.0GHz,

表 7.6: データサイズ推論の実装

モジュール名	行数	機能
IntBound	606	L ノードの値域を表す．各 L ノードに対応したメソッドを提供する．
IntLive	446	L ノードの出力として有効なビットを表す．各 L ノードに対応したメソッドを提供する．
IntConst	525	上記 2 つの計算で使われる基本データ型と演算を提供する．

```
        movq ones,%mm0      # 0x0001000100010001
loop:
        movq (%edi),%mm1    # %edi = *b
        movq (%esi),%mm2    # %esi = *c
        movq %mm2,%mm3
        por  %mm1,%mm3
        pand %mm0,%mm3
        psraw $1,%mm1
        psraw $1,%mm2
        paddw %mm1,%mm3
        paddw %mm2,%mm3
        movq %mm3,(%eax)    # %eax = *a
```

図 7.10: 図 7.7 に対して生成されるコード

SiS-655 チップセット，512MB Dual-channel DDR-SDRAM メモリ，OS が Windows2000 の構成のもの（以下 P4 と略）と，Pentium-M 1.3GHz，512MB DDR-SDRAM メモリ，OS が WindowsXP の構成のもの（以下 PM と略）で，cygwin の C コンパイラ（gcc 3.3.1）のアセンブラを使用した．

7.4.1　平均値を求めるプログラム

図 7.1 の（a）の場合に対応する図 7.7 の解析結果から生成したコードを図 7.10 に，図 7.1 の（b）の場合に対応する図 7.9 の解析結果から生成したコードを図 7.11 に示す．

先ず，上向き解析の結果によるコード生成の改善例を説明する．図 7.9 の解析結果によると 32 ビットから 16 ビットへの縮退が必要となるが，図 7.11 では符号付き飽和（符号つき整数の −32768 以下の値を −32768 に，32767 以上の値を 32767 にする）縮退命令 packssdw を用いている．ここでは飽和なし縮退を使うべきだが，命令セットに用意されていない．しかし上向き解析の結果，飽和の影響を受けない値域であることが判るので，packssdw が飽和なしと同じ動作をすることが保証される．このように，解析結果を用いて命令セットの不足を補うことも可能である．

次に，処理時間で比較を行う．図 7.10 や図 7.11 のコード，AVE の定義を（a）として素直に汎整数

```
        movq ones,%mm0        # 0x0000000100000001
loop:
        punpcklwd (%edi),%mm1 # lo half
        punpcklwd (%esi),%mm2 # CONVSX in LIR

        psrad $16,%mm1
        psrad $16,%mm2
        paddd %mm1,%mm2
        paddd %mm0,%mm2
        psrad $1,%mm2         # lo result
        punpckhwd (%edi),%mm4 # higher half
        punpckhwd (%esi),%mm3 # CONVSX in LIR
        psrad $16,%mm4
        psrad $16,%mm3
        paddd %mm4,%mm3
        paddd %mm0,%mm3
        psrad $1,%mm3         # higher result
        packssdw %mm3,%mm2    # saturated packing
        movq %mm2,(%eax)      # write out the result
```

図 7.11: 図 7.9 に対して生成されるコード

拡張して生成したコード，それに gcc でコンパイルしたコードの処理時間を，表 7.7 に示す．

表の各欄の（a）と（b）を比較すると，通常命令による処理の場合や，SIMD 命令を使っても汎整数拡張を行う場合は AVE の定義を（b）とした通常の平均の求め方の方が実行効率が高い．しかし，データサイズ推論を行って不要な汎整数拡張を除去可能ならば，AVE の定義を SIMD 命令向きの（a）とした SIMD 命令向きの求め方が実行効率が最も高い．そして，（b）行の gcc 欄や IP 欄に対する改善比は，SIMD 命令の実行ユニットが改良されている PM の方が高い．このように SIMD 向きのコーディングから最大の実行効率を得るには，提案する方式で最適な処理データサイズを調べることが不可欠であると思われる．

表の（a）行の DSI 欄と IP 欄を比較すると，汎整数拡張や処理の並列度の半減によるのオーバヘッドは P4 の場合で 6〜16%であるが，改良型の PM の場合は 100〜178%にもなっている．これから，機種によっては不要な汎整数拡張が SIMD 命令を使ったコードの実行効率を非常に悪化させることがあると判る．

7.4.2 動画圧縮プログラムからの例題

MPEG-4 Video CODEC の 1 つの実装である XviD（1.0.0 版）中の関数 `interpolate8x8_halfpel_hv()` の実行時間を計測した．この関数は，8x8 の画素集合を受け取り，2 × 2 ドットの輝度の平均を各点について求めて，同じサイズの画素集合に書き出す．平均は，画素集合の要素の 4 つの符号無し 8 ビッ

表 7.7: 配列間の平均を求めるプログラムの処理時間

機種/定義		MMX		SSE2	
	gcc	DSI	IP	DSI	IP
Pentium-4/(a)	4.37	3.67	4.27	3.68	3.94
Pentium-4/(b)	3.80	—	3.90	—	3.82
Pentium-M/(a)	4.53	1.11	3.09	1.49	2.98
Pentium-M/(b)	1.98	—	2.10	—	2.43

10 億回繰り返した時の要素あたりの平均処理時間 (時間の単位はナノ秒)
定義: 図 7.1 の AVE の定義
gcc: gcc の最適化-O6 で生成したコード
MMX: MMX 命令セットで 64 ビット処理
SSE2: SSE2 命令セットで 128 ビット処理
DSI: データサイズ推論の結果から最適なオペランドサイズを選択したコード
IP: 素直に汎整数拡張を実施したコード

表 7.8: XviD の補間関数の実行時間

機種		MMX	
	gcc	DSI	IP
Pentium-4	342	158	372
Pentium-M	344	153	355

1000 万回繰り返した時の 8x8 の画素集合あたりの
平均処理時間（単位はナノ秒）
各項目の意味は表 7.7 に同じ.

トの数と丸めをコントロールする定数の和を求めて，その値を右に 2 ビットシフトして求める．データサイズ推論を用いると 16 ビットに拡張して演算を行えばよいことが判るが，通常は汎整数拡張のために 32 ビットに拡張してから演算するコードになる．

この実行結果を表 7.8 に示す．ここでは画素集合のサイズの都合で，SSE2 命令を使った 128 ビット処理の実験は行っていない．

DSI 欄と IP 欄を比較すると汎整数拡張と並列度の半減のためのオーバヘッドが 130%以上になっている．不要な汎整数拡張を行うと，最適な演算データサイズ（16 ビット）での処理の場合に比べて 2 倍以上の時間がかかり，しかも gcc 欄との比較からスカラ処理よりも遅くなり，SIMD 並列化した意味を失っている．

7.5 この章の結論

言語拡張を行わないで，効率の良いマルチメディア向け SIMD 命令を生成するためのデータサイズ推論の方式を提案し，試験的なコード生成を行い，方式の有効性を確認した．

本章では代入式単位での解析に焦点を絞り，コンパイラが式をまとめ上げることを仮定した．しかし，実装した処理系では第 6 章で述べたように，代入文の右辺式は DAG 化されて処理されるために，変数の代入間を跨る値域の推論が可能であり，7.2.1 節にあるように「参照された変数の値域はサイズで決まる環の全域とする.」とした場合比べて，精度の高い解析が可能となっている．なお，表 7.2 や表 7.5 を求めるために文献 [32] の 2, 3, 4 章を参考にした．

表 7.5 への注

☆注 1

加算では，定数（例えば x）の最下位からみて i ビットの 0 が連続しているとき，加算結果の最下位から i ビットの並びには y の最下位から i ビットの並びの値がそのまま反映する．w の最上位から見て最初の 1 のビットの位置を j とすると，加算結果の i〜j ビット目の間の値は，y の i〜j ビット目の間のビットの値で値で決まる．従って，y に伝える有効ビット集合は，w の 0 ビット目から $\mathrm{i}-1$ ビット目の値と，i〜j ビット目を 1 にした値の合併となる．

例:
`01000010 w`（上から来た有効ビット集合）
`00101000 x`（定数）
`01111010 y` に下げるべき有効ビット集合

減算では，x が定数なら x のビット毎の反転を，y が定数なら $-y$ を用いて加算の推論規則を用いる．

☆注 2

定数（例えば x）の最下位からみて i ビットの 0 が連続していて（最下位からみてビット i ではじめて 1），次の 1 が $\mathrm{i}+\mathrm{j}$ ビット目だとする．すると演算結果のビット 0〜$\mathrm{i}-1$ は常に 0 になり，ビット i〜$\mathrm{i}+\mathrm{j}-1$ までは y の最下位のビット 0〜$\mathrm{j}-1$ までの値がそのまま出力される．ビット $\mathrm{i}+\mathrm{j}+\mathrm{k}$（$\mathrm{k} \geq 0$）では，$y$ のビット 0〜k とビット j〜$\mathrm{j}+k$ までの影響を受ける．よって y に伝える有効ビット集合は，K を w の最上位から見て最初の 1 の位置とすれば，ビット j〜$\mathrm{j}+\mathrm{K}$ を 1 にした値と，ビット 0〜K を 1 にした値と，w を右に i ビットシフトした値の合併になる．

例:

```
   hgfedcba
×  01010010
------------
    fedcba
   dcba
   ba
```

```
-----------
   ****cba0
    |||
    ||a，d の影響を受ける
    |a，b，d，e の影響を受ける
    a，b，c，d，e，f の影響を受ける（3 番目の 1 の影響は 2 番目の 1 の影響に含まれる）
```

第8章　SIMDベンチマーク

この章では，SIMD ベンチマークの意図や設計，そして実装について議論する．

一般的な「ベンチマークプログラム」という語は，ハードウエアやシステムソフトウエアの性能評価を数値化するためのプログラム集という意味を持つが，ここでは，次に議論するようなプログラム集も，ベンチマークプログラム，あるいは単にベンチマークと呼ぶことにする．

8.1　目的と背景

コンパイラによる SIMD 命令の自動適用は，まだ非常に限定的であるので，プログラム作成者は，ソースプログラムにおいてどのようなコーディングを行えば適切な SIMD 命令が生成されるかを知っておく必要がある．それは，アルゴリズムを素直にプログラム化した場合にくらべ，SIMD 命令を適用しやすいように変形したパターンとなる場合が多い．

例えば，ベクタ間の平均を求める図 7.1 のようなコードに対して，表 7.7 のような実行結果が示されると，コンパイラのユーザはこの問題に対して，どのコーディングが実行性能が高いコードを生成するかを知ることができる．また，コンパイラの設計者は，図 7.1 の（a）のようなコードに対しても，ターゲットの命令セットの範囲内で，適切な SIMD 命令を生成することを検討する．例えば，平均値命令が使えるなら，両方のコードバターンに対して平均値命令を生成することを検討し，平均値命令が無いならば，（a）や（b）のコードパターンに対して，それぞれ図 7.10 や図 7.11 のようなコード生成を検討する．この例を組織的に解決するには，コンパイラにデータサイズ推論のような解析機構を組み込むこと等を考えなければならないが，これはコンパイラの設計者に対して有益な目標を与えていると考える．

あるいは，ユーザが 5.1.3 節で紹介したシャッフル命令（図 5.5）や特殊なリダクション命令（図 5.10）をコンパイラに生成させて，プログラムの重要な部位の性能向上を考える場合にも，どのようなコードパターンに対してコンパイラがこの命令を生成するかを知りたいと考えるだろう．反対にコンパイラの設計者は，示されたコードパターンに対して，なるべく多くを統一的にカバーする方法を考える．

一般に，同じ例題に対する変形には，元のアルゴリズムに近いものから，SIMD 命令に近いものまで，いくつもの段階があり得る．SIMD 命令セットを適用できそうな問題を多く収集して，それらを体系的に整理し，多くの段階ごとにパターン化したプログラム集を作成すれば，プログラム作成者は，使用するコンパイラがどんなパターンならば高速処理できる SIMD 命令列を生成するか，つまり SIMD 命令を活用できるかを容易に知ることができる．

このようにして構成された SIMD 命令向けのベンチマークは，コンパイラ設計者にとっても，設計目標を定めやすくなり，テストプログラムとして利用することもできて有用である．

ベンチマークには 5.3 節で紹介したように，すでに多くの種類がある．しかし，先に述べたように，従来のベンチマークは，通常の命令セットやベクタ命令セットに対して，コンパイラによる最適化の

効果や命令セットの実装方法による性能の変化を調べるのに適していても，以下の理由によりコンパイラが SIMD 命令セットを適切に活用したコードを生成できるかどうかの評価には適していない．

- それらのベンチマークは大きい実用プログラムをそのまま使用しているものであって，SIMD 命令セットを活用できる部位の特定が容易でなく，SIMD 命令セットの活用対象となり得るコーディングパターンを体系的に整理したものでもないので，上記の目的には合わない．
- 8.2.2 節の 2 つ目の例のように，アルゴリズム自体はデータ並列性などの点で SIMD 命令セット向きであるが，そのままではプログラム中の定数値やデータサイズなどの選択が不適切なため，現在の最適化技術では効果的な SIMD 命令の適用が不可能か困難であるものが多い．
- 8.2.4 節で詳述するアラインメントへの不整合やベクタの重なりがある場合に，誤った SIMD 命令を生成していないことが判定できない．

SIMD 命令セットという比較的新しい機能について，それを有効活用し得る部位を実際のプログラムの中から見つけ出し，コンパイラのユーザや設計者にその部位の多様な表現を系統的に示すことは，最終的にプログラムの実行性能の向上に大いに寄与するものと考えられる．

そこで本研究では，SIMD 命令セットに特化したベンチマークを作成することを提案し，その 1 つの実装例を示す．これには，SIMD 命令の処理機能を単体でテストする例題のほかに，実際のアプリケーションから抽出した処理パターンから得た例題も含める．現在の実装は整数演算の SIMD 命令セットに特化している．それは以下の設計要件を満たすことを目標とする．

1. コンパイラの設計者に対して，SIMD 命令の適用対象となり得るパターンを体系的に提示することによって、最適化の設計目標を示す．
2. SIMD 最適化コンパイラのユーザに対して，そのコンパイラ向けのコーディングパターンを示す．
3. SIMD 命令の誤った適用の検出を可能にする．
4. SIMD 命令セットの実装の違いによる性能比較を可能にする．

さらにベンチマークの実装にあたっては，2 番目の事項を満たすために，処理系がベクタ化に基づく並列化方式（ループの形）と，第 6 章で述べた並列化方式のいずれの方針を採っていても SIMD 命令の適用が促されるように，ループの形と展開済みの形の両方のコーディングを行う．

8.2　SIMD ベンチマークの設計

ベンチマークは，それ自体の処理系依存性を避けるために，ANSI C 言語の機能の範囲内で記述し，処理系依存性のあるベクタ処理等向けの特別な構文やプラグマは使用しない．8.2.1 節の方法で収集した処理パターンは，8.2.2 節に示すように SIMD 命令向きに変形して複数のバリエーションを構成し，それぞれを 8.2.3 節に示す方法で展開し，同じ処理パターンに対して複数のコーディングを作成する．各々のコーディングは 1 つの関数としてまとめ，それぞれについて実行時間を計測し表示する．実行時間の計測には，Unix 系 OS では関数 `getruseage()` を，Windows 系 OS では関数 `clock()` を使用する．

```
typedef struct array_t{
  unsigned int cdt, col, pos, neg;
} array;
...
  int lsb = (-r) & r;
  a[h+1].cdt = ~(       ~r | lsb);
  a[h+1].col = ~(~a[h].col | lsb);
  a[h+1].pos =  ( a[h].pos | lsb) << 1;
  a[h+1].neg =  ( a[h].neg | lsb) >> 1;
  r = a[h+1].col & ~(a[h+1].pos | a[h+1].neg);
  h++;
```

図 8.1: N-queens 問題の繰り返しによる解

8.2.1 例題の収集

例題としては，加減算のような単体の SIMD 命令に対応する単純なものの他に，平均や最大値のように数ステップの SIMD 命令で構成可能な基本処理パターンによるものと，C 言語のソースプログラムが公開されている実際のプログラムから抽出した処理パターンを元に作成したものを収集した．この節では，実際のプログラムからの処理パターンの抽出について述べる．

実際のプログラムからの処理パターンの抽出は，以下のようにして行う．

1. プログラムを gprof[24] か Intel VTune[35] を用いて，プロファイリングする．
2. ホットスポット（実行時間に占める割合が多い箇所）を，ループや文レベルの実行部位単位で抽出する．
3. それらに対して SIMD 命令の適用可能性を検討し，適用可能なものを残す．
4. 適用可能なものに対してアセンブリ言語による SIMD 命令を活用したコーディングを行い，コンパイラが生成する通常命令（スカラ命令）のコードとの実行時間の比較を行い，高速化できたものを採択する．

これらの作業は，主に IA-32 と SSE2 の計算機の上で行ったが，一部 Power Mac G5 も用いた．gprof では抽出可能なホットスポットは関数単位なので，人手によるソースコードの解析で該当する実行部位を抽出した．

MediaBench[46] を構成するプログラム群も調査の対象としたが，多くのホットスポットを (3) により候補から外した．例えば，暗号化や復号化プログラムの中には，一見したところ SIMD 命令の効果的な適用が可能であるように思われるものがあった．しかし，256 エントリの小規模な表引きの並列実行などのように，現状の SIMD 命令セットでは処理できない演算を含んでおり，やはり候補から外した．ファイル圧縮・伸長のプログラムも同様であった．

プロセッサの実装方法によって (4) で採択の可否が異なる場合があったが，その場合は採択とした．例えば IA-32 の Northwood や Banias，Prescott と呼ばれる実装はいずれも SSE2 命令セットを実装

```
#define ABS(X) (((X)>0)?(X):-(X))
int func0(unsigned char *a, int sum, int sz){
  int i;  unsigned char v;
  for(i=0; i<SIZE; i++, a++){
    v = *a;
    /* 下記 (a)〜(e) のうちの 1 つ */ }
  return sum; }

(a) sum += (v < 128) ? v : 256 - v;
(b) sum += 128 - abs(128-v);
(c) sum += 128 - ABS(128-v);
(d) sum += (v < 128) ? v : (unsigned char)(-v);
(e) sum += (v < 128) ? v : (unsigned char)(~v + 1);
```

図 8.2: 画像フォーマット変換プログラムからの例題（sad）

している．Northwood では SIMD 命令の演算器には倍クロック演算器を用いていないが，後の 2 者はそれを用いており，SIMD 命令を適用したコードの実行性能がピークで 1.8 倍程度まで高まっている．この場合は，プロセッサを変えて通常命令のコードよりも高速化できたものがあれば，例題として採択した．

(4) に関連して，SIMD 命令を適切に活用しても通常命令のコードに比べて高速化できなかった例として，N-queens 問題の繰り返しによる求解 [38] を紹介する．そのループカーネルを SIMD 向きに変形したものを図 8.1 に示す．このプログラムは，フィールドごとに独立にシフト量を決めることができる Altivec で効果的に処理できると思われ，Power Mac G5 で SIMD 命令を使ったコーディングを数通り試したが，最も実行効率が高いものでも 5%程度通常命令のコードより遅くなった．これは，構造体の 4 つのメンバへの代入には並列性があるので，通常命令でも命令レベル並列性が十分に発揮されているためであると思われる．

8.2.2 例題の変形

こうして選ばれた処理パターンに対して，同じ結果をもたらし，SIMD 命令の適用を可能とする方向への変形を行った．それを例によって説明する．

画像フォーマット変換プログラム bmp2png[53] から，図 8.2 に示すような処理パターンを得た．図のコメントの箇所には (a)〜(e) のコードのいずれかが埋め込まれる．いずれも通常の C コンパイラでは同じ結果をもたらす．作業用変数 v に代入しているのは，多くの処理系ではポインタで参照された変数はレジスタへの割り付けを行わないからである．オリジナルでは (a) のコーディングが使われていたが，このコードに対してそのまま SIMD 命令を適用すると，定数の 256 に対する汎整数拡張のために 9 ビット以上で演算を行うことになる．すると，文献 [69] のような解析を行っても，一般的な SIMD 命令の仕様では演算のデータサイズが 16 ビットになってしまい，並列度が低下したりサイズ変換のオーバヘッドが伴うので，SIMD 命令向きのコーディングではない．(b) から (e) のコー

```
/* 割り算を掛け算とシフトで行うための表 */
const unsigned int multipliers[32] ={ 0,32768,16385,10923, ····途中
  省略····,1214,1171,1130,1093,1058 };
unsigned int quant5  (
  int16_t *  coeff, const int16_t *  data, const unsigned int  quant )
{ const unsigned int mult = multipliers[quant];
  const unsigned short quant_m_2 = quant << 1;
  const unsigned short quant_d_2 = quant >> 1;
  int sum = 0;  unsigned int i;
  for (i = 0; i < M; i++) {
    int16_t acLevel = data[i];
    if (acLevel < 0) {
      acLevel = (-acLevel) - quant_d_2;
      if (acLevel < quant_m_2) { coeff[i] = 0; continue; }
      acLevel = (acLevel * mult) >> 16;
      sum += acLevel;    // sum += |acLevel|
      coeff[i] = -acLevel;
    } else {
      acLevel -= quant_d_2;
      if (acLevel < quant_m_2) { coeff[i] = 0; continue; }
      acLevel = (acLevel * mult) >> 16;
      sum += acLevel;
      coeff[i] = acLevel; } }
  return sum; }
```

図 8.3: 動画像圧縮プログラムからの例題 (quant5)

ディングは 8 ビットの演算で済むように改良したものである．(b) では絶対値を求める関数 abs() を呼び出しているが，多くの処理系ではコンパイラが組み込み関数として認識し，インライン展開等を行って関数呼び出しを行わないコードを生成する．ここでは，SIMD 最適化でもこれを期待し，命令セットに依存した最適なコード生成を期待する．(c) はコンパイラがそのような認識を行わない場合に，有効であると思われるコーディングである．(d) と (e) は，256-v の結果がサイズの変数への書き出しの際の 8 ビット目以上の切り落としルールでは，ラップアラウンドにより 0-v と同じ結果になることを利用したコーディングである．

図 8.3 の例は，MPEG4 動画像圧縮プログラムから選んだものの 1 つである．オリジナルでは，multipliers[] が int として宣言されているため，多くの SIMD 命令セットの乗算命令では効率的に処理できない．また，for ループ内の処理が SIMD 最適化の対象になるが，2 重の入れ子になった if 文の平坦化を伴う if 変換，共通処理の括り出しと段階的に SIMD 命令の適用向きに変形し，最終的に図 8.4 や 図 8.5 のような SIMD 命令に容易に対応付けられるコーディングに帰着させる．

このようにして，抽出した例題に対して，複数の変形パターンを作成していく．

```
const unsigned int multipliers[32] ={ 0,32767,.. // 表の値を変更
    int16_t acLevel, acLevel2;
    acLevel = ((data[i] < 0) ? -data[i] : data[i]) - quant_d_2;
    acLevel2 = (acLevel * mult) >> SCALEBITS;
    sum += ((acLevel < quant_m_2) ? 0 : acLevel2);
    coeff[i] = ((acLevel < quant_m_2)
                ? 0
                : ((data[i] < 0) ? -acLevel2 : acLevel2));
```

図 8.4: quant5 のループカーネルに対する変形 1

```
const unsigned int multipliers[32] ={ 0,32767,.. // 表の値を変更
    int16_t acMsk1, acMsk2, acLevel;
    acMsk1 = (data[i] < 0) ? -1 : 0;
    acLevel = ((data[i] & ~acMsk1)|((-data[i]) & acMsk1)) - quant_d_2;
    acMsk2 = (acLevel < quant_m_2) ? -1 : 0;
    acLevel = (acLevel * mult) >> SCALEBITS;
    sum += ~acMsk2 & acLevel;
    coeff[i] = ~acMsk2 & (((-acLevel) & acMsk1) | (acLevel & (~acMsk1)));
```

図 8.5: quant5 のループカーネルに対する変形 2

8.2.3 ループ展開

こうして SIMD 命令向きに何通りかの変形を作られた例題のそれぞれに対して，ループを展開しない形と展開した形を作成する．多くの場合はループの形で記述されているので，幾通りかのループ展開を行ったが，最初から展開された形で記述されていたものもあったので，この場合は逆にループの形に変形した．これによって，処理系が SIMD 並列化をベクタ化に基づいて行っているか，あるいは同型命令の認識に基づいて行っているか，あるいは両者を行っているかを判断できる．展開形では，ベクタレジスタ長は 128 ビットであるとし，第 7 章で議論したように最適な処理データサイズを選択しているとして展開数を決めた．

また，図 8.2 の例の (b) や (c) のコーディングと，5.1.3 節の特殊なリダクション命令に関連して，図 8.6 のように展開したループ内では 128 と v の差の絶対値の総和を求めて，その後に 128 を展開数分足し合わせた値から総和を引くようにコーディングすると，展開したループ内にリダクション命令をそのまま適用可能になる．適用可能ならばこのような変形もループの展開と同時に行う．図 8.6 の変形の場合は，s に差の絶対値を積算していくのに，複数の代入文に分けた記述と，ひとつの式にまとめる記述が可能である．ベンチマークでは，両者について試すようになっている．

```
int func0(unsigned char *a, int sum, int sz){
  int i;  unsigned char v;
  int s;
  for(i=0, s=0; i<SIZE-15; i+=16, a+=16){
    s += abs(128-a[0]); s += abs(128-a[1]);  s += abs(128-a[2]);
    s += abs(128-a[3]); s += abs(128-a[4]);  s += abs(128-a[5]);
    s += abs(128-a[6]); s += abs(128-a[7]);  s += abs(128-a[8]);
    s += abs(128-a[9]);  s += abs(128-a[10]);s += abs(128-a[11]);
    s += abs(128-a[12]);s += abs(128-a[13]); s += abs(128-a[14]);
    s += abs(128-a[15]); }
  sum += 2048 - s;
  for(; i<SIZE; i++, a++){
    v = *a;
    sum += 128 - abs(128-*a); }
  return sum; }
```

図 8.6: 図 8.2 の例題（sad） の特殊なリダクション命令向き変形

8.2.4 誤ったコード生成の検出

SIMD 命令に関連して，コンパイラが誤ったコード生成を行う可能性のあるパターンは次の通りである．

1. ベクタレジスタのサイズのアラインメントに合っていないポインタを使ったベクタレジスタとメモリの間の誤った転送

2. 書き込みのベクタと読み出しのベクタに重なりがある場合の SIMD 命令の誤った適用

（1）は IA-32 のような完全なバイトマシンでは問題にならないが，バイトマシンでない Altivec や EmotionEngine 等では問題となる．実際のアプリケーションプログラムを分析すると，ほとんどの場合で処理データサイズの倍数にポインタが設定されるように記述されているので，この項目の検査が必要である．そこで，SIMD 命令を適用されることが想定される関数の呼び出しで渡すポインタのアラインメントを，ポインタが指すオブジェクトのサイズの倍数でずらし，同じパラメタを渡された通常命令のコードの実行結果と比較することによって，この項目の検査とした．

（2）の例としては，図 5.12 で v2 が v1 の 1 エントリー後を追いかけていき，v2 から読めるデータは v1 を通して書かれた値である場合が挙げられる．この場合に SIMD 命令を適用すると，通常命令でひとつずつ処理する場合と異なる結果になる．これに対するコンパイラによる対策としては，関数の先頭で，全ての書き込みベクタのポインタと読み出しベクタのポインタの組み合わせで，両者の距離がベクタレジスタのサイズよりも大きいことを検査し，接近しているものがあれば，通常命令のコードにスイッチするという仕組みが考えられる．コンパイラがこのような対策を施しているかどうかを検出するために，わざと接近したポインタを関数に渡し，比較参照用のコードの実行結果と比較することによって，この項目の検査とした．

表 8.1: 実験に用いたコンパイラとオプション

コンパイラ	SIMD 命令生成時	基本命令生成時
Intel icc ver. 8.1	`-axW -O3`	`-O3`
gcc ver. 4.0.0	`-O6 -ftree-vectorize -msse2`	`-O6`

8.3 実装と実験

8.3.1 実装

現在のベンチマークの実装で取り上げている例題は，以下の通りである．6 データタイプとは，符号つき/符号なしの `int` と `short` と `char` の組み合わせを意味する．3 データタイプとは，符号つきの `int` と `short` と `char` の組み合わせを意味する．コンパイラが 5.1.1 節で述べた最適な処理データサイズの選択を行うかどうかは，全ての例題に共通の課題である．

ave （6 データタイプ× 4 変形）2 つの配列の対応する要素の平均を求め，別の配列に書き出す問題．通常の足してから 2 で割るアルゴリズム（符号拡張が必要）と，図 5.12 のような 2 で割ってから足すアルゴリズム（符号拡張は不要）を用意した．

add （3 データタイプ× 2 変形）2 つの配列の対応する要素の和を求め，別の配列に書き出す問題．結果がラップアラウンドする加算と，符号つきと符号なしのそれぞれで飽和する加算の 3 つの場合を，要素のサイズを 8，16，32 ビットと変えてテストする．

max （6 データタイプ× 2 変形） 2 つの配列の対応する要素の大きい方の値を求め，別の配列に書き出す問題．要素のサイズ 8，16，32 ビットについてそれぞれ符号つき／符号なしと変えてテストする．

maxc （6 データタイプ× 6 変形）2 つの配列の対応する要素を比較し，片方の配列について大きい値をもつ要素の数を調べる問題．比較命令の結果を数値として使う能力や，特殊なリダクション命令を適用する能力を試す．

sum （6 データタイプ× 3 変形）配列の総和を求める問題．要素のサイズ 8，16，32 ビットについてそれぞれ符号つき／符号なしと変えてテストする．

quant5 （6 変形）図 8.3 の例題．オリジナルと図 8.3 の 2 つの変形をテストする．

sad （37 変形）図 8.2 の例題．これには，特殊なリダクション命令の生成を促す変形が含まれている．

bsad （37 変形）同じ出典に含まれていた sad の派生問題．図 8.9 に示すようにループの途中 `sum` の値が定数値 `BREAKPOINT` よりも大きくなったらループを脱出する．

実験に使用したのは，EPSON EDiCube S160（Pen-tiumM 1.3GHz, 機種 1 とする）と，ASUS P4P800 と Pentium4 HT 2.6GHz の組み合わせの計算機（機種 2 とする）に，それぞれオペレーティングシステムとして Linux （kernel ver. 2.4.27）を搭載したものである．コンパイラと SIMD 命令生成や基本命令生成の指示に用いたオプションの組み合わせは表 8.1 の通りである．

8.3.2 結果

実験で用いた gcc のバージョンにはベクタ化機能が備わっているので，SIMD 命令を活用したコードの生成を期待したが，本ベンチマークのような配列を関数のパラメタで渡す形のコーディングでは，この機能は有効にならなかった．さらにベンチマークを書き換えて，グローバル変数に直接アクセスするように変更してみたが，if 変換が必要な箇所や簡単なシフト演算の翻訳でコンパイラが異常終了した．

```
(1) ループのままで，一時変数を使う
  for (i=0;i<SIZE;i++,a++) {
    v = *a; sum += 128 - abs(128 - v) }
(2) ループのままで，ポインタで直接参照
  for (i = 0; i < SIZE; i++, a++)
    sum += 128 - abs(128 - *a)
(3) ループを展開し，一時変数を使う
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16){
    v = *p;       sum+= 128 - abs(128-v);
    v = *(p+1);   sum+= 128 - abs(128-v);
    ....
    v = *(p+15); sum+= 128 - abs(128-v);}
  (1) のループ /* 端数処理 */
(4) ループを展開し，ポインタで直接参照
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16){
    sum += 128 - abs(128-*p);
    sum += 128 - abs(128-*(p+1));
    ....
    sum += 128 - abs(128-*(p+15));
  (2) のループ /* 端数処理 以下では省略 */
(5) (4) をひとつの式にまとめる
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16){
    sum+= 128 - abs(128-*p) + 128 - abs(128-*(p+1))
    ....
(6) (4) の 128 の 16 回の加算をまとめる
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16){
    sum -= abs(128-*p);
    sum -= abs(128-*(p+1));
    ....
    sum += 2048;}
```

図 8.7: 図 8.2 の例題 (sad) の演算パターン（c）についてのループの変形

```
(7) (5) の 128 をまとめる
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16){
    sum -= abs(128-*p) + abs(128-*(p+1))
    ....
    sum += 2048;}
(8) (6) で別の集積変数を使う
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16){
    tmpsum += abs(128-*p);
    tmpsum += abs(128-*(p+1));
    ....
    sum += 2048-tmpsum; tmpsum = 0;}
(9) (8) の tmpsum の初期化位置を変更
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16,tmpsum=0){
    tmpsum += abs(128-*p);
    tmpsum += abs(128-*(p+1));
    ....
    sum += 2048-tmpsum;}
(10) (7) で別の集積変数を使う
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16){
    tmpsum = abs(128-*p) + abs(128-*(p+1))
    ....
    sum += 2048-tmpsum; tmpsum=0;}
(11) (10) の tmpsum の初期化位置を変更
  for(i=0,p=a;i<(SIZE-(SIZE&15));i+=16,p+=16,tmpsum=0){
    tmpsum = abs(128-*p) + abs(128-*(p+1))
    ....
    sum += 2048-tmpsum;}
```

図 8.8: 図 8.7 の続き

icc は多くの例題の SIMD 向け変形版に対して，SIMD 命令を生成していた．表 8.2 に機種 1 での図 8.2 の例題（sad）について，図 8.7 と図 8.8 に示すようなループ変形（(1)〜(11)）と演算の変形（(a)〜(e)）の組み合わせのそれぞれに対する機種 1 での実行時間を，SIMD 命令を生成した場合（斜線の左側）と基本命令のみの場合（斜線の右側）について示す．また，図 8.9 の例題（bsad）の実行時間を，表 8.3 に示す．この例題のループ展開版では，脱出条件を満たした際の巻き戻し（ループをひとつ前の状態に戻す）処理を含んでいる．

これらの全てについて，SIMD 命令の生成を指示した場合には,「ベクトライズした」という趣旨のメッセージをコンパイラが表示していた．これらの結果についての考察は，8.3.3 節で行う．

表 8.4 に，コンパイラとして icc を用いた場合の，図 8.3 の例題（動画像圧縮）の結果を示す．この問題では，変形 2 をループ展開した場合にのみ SIMD 命令が生成された．また，機種 1（PentiumM,

```
int func0(unsigned char *a, int sum, int sz){
  int i;  unsigned char v;
  for(i=0; i<SIZE; i++, a++){
    v = *a;
    sum += (v < 128) ? v : 256 - v;
    if (sum > BREAKPOINT) break; }
  return sum; }
```

図 8.9: 図 8.2 の例題（sad）の派生版

表 8.2: 図 8.2 の例題（sad）の実行時間（icc を使用）

ループ	演算の変形				
の変形	(a)	(b)	(c)	(d)	(e)
(1)	8600/15230	4380/ 6280	4390/ 6280	20040/14790	10880/15440
(2)	8590/15410	4390/ 6350	4390/ 6350	19990/15260	10880/15140
(3)	5500/13290	4390/ 5710	4390/ 5700	13470/13490	5490/13480
(4)	5490/13260	4400/ 5720	4400/ 5720	13440/13460	5490/13290
(5)	6120/13750	5590/ 6010	5590/ 6010	13750/13750	6180/13900
(6)		5610/ 4880	5610/ 4890		
(7)		5610/ 5740	5610/ 5740		
(8)		5520/ 4990	5510/ 4990		
(9)		5530/ 5000	5530/ 5000		
(10)		5520/ 6100	5510/ 6100		
(11)		5520/ 6100	5520/ 6090		

SIMD 命令の生成を指示時/通常命令の生成を指示時（単位はミリ秒）
ループの変形は図 8.7 中の番号に対応.
空欄の箇所は，有効な該当する変形がないことを意味する.

実装は Banias）では SIMD 命令の適用により高速化しているのに対し，機種 2（Pentium4, 実装は Northwood）では逆に低速になった．これはプロセッサの実装の違いによるものと思われる．

8.2.4 節で示した誤ったコード生成に関するテスト項目について議論する．使用したコンパイラはここに挙げたテスト項目をクリアするコード生成を行っていた．また，IA-32 はバイトマシンなので任意のデータサイズに対して任意のアドレス境界からの参照を許しており，メモリ参照時にアラインメントなしのメモリ参照（movdqu）命令を生成している限りは，(1) のテストを通過する．そこでアセンブリ出力を修正して，アラインメントつき (movdqa) に変更して実行すると，結果の間違いを報告した．また生成されたアセンブリ出力を検討したところ，icc は関数の先頭でベクタの重なりの動的な検査を行っており，(2) のテストも通過した．そこで，アセンブリ出力を修正して，検査のコードを実行しないようにすると，結果の間違いを報告した．よって本ベンチマークは，8.2.4 節に挙げ

表 8.3: 図 8.9 の例題 (bsad) の実行時間 (icc を使用)

ループ	演算の変形				
の変形	(a)	(b)	(c)	(d)	(e)
(1)	8420/14840	8300/ 8320	8290/ 8320	14280/14550	8420/15280
(2)	8090/14840	8270/ 8310	8270/ 8310	14640/14970	8080/15260
(3)	4450/10350	4600/ 4610	4600/ 4610	10450/10530	4470/10390
(4)	4460/10300	4100/ 4630	4100/ 4630	10340/10530	4440/10200
(5)	4950/10820	4540/ 4970	4540/ 4970	11020/10890	5120/11130
(6)		4580/ 3930	4590/ 3930		
(7)		4590/ 4910	4590/ 4910		
(8)		4600/ 4020	4600/ 4010		
(9)		1130/ 4010	1130/ 4020		
(10)		4590/ 4980	4590/ 4980		
(11)		1130/ 4980	1130/ 4980		

詳細は表 8.2 に同じ

表 8.4: 図 8.3 の例題 （quant5） の実行時間（ミリ秒）

	原型		変形 2 をループ展開		高速化比
	SIMD	基本 (a)	SIMD(b)	基本	(a/b)
機種 1(PentiumM)	820	840	410	900	2.05
機種 2(Pentium4)	420	400	460	629	0.87

た検査を想定される範囲内で行う能力を有することが確認できた.

8.3.3 本ベンチマークの利用法

コンパイラユーザの立場で，このベンチマークが示す結果の活用法を，表 8.2 や表 8.3 から得られる情報を例として説明する.

ユーザが記述しようとしている処理パターンがこれらの例題にあれば，原則として，表の中から SIMD 命令を適用して最も処理時間が短いものを採用すればよい．ただし，次に述べるように類似の処理パターンの結果も参考にすべきである.

表 8.3 で，実行時間が 1000 ミリ秒台のパターンについてアセンブリコードを検討したところ，psadbw（バイトデータ間の差の絶対値の，ベクタレジスタ内の総和を求める）命令というリダクション命令を生成していた．ただし，表の前後の対応するパターンを見比べると，この命令へのマッチングは非常に狭く，コンパイラ内では何かの特定処理向けの大きなパターンへのマッチングを行っていると推察される．図 8.2 の例題は，図 8.9 の例題の「脱出がない」という特殊な場合であると考えられるので，脱出の閾値 `BREAKPOINT` を十分大きい値に設定して，図 8.9 のような記述にする方が，psadbw

命令にマッチして，処理を大幅に高速化できることがわかる．このように類似の処理パターンの結果も見ていくと，より高い実行性能を得られるパターンを見出せる場合がある．

目的の処理パターンそのものがベンチマークにない場合は，ベンチマーク中から目的の処理パターンに近い例題を探し，その例題の結果を検討して目的の処理を記述していく．例えば，1 つのベクタから（他への副作用なしで）スカラ値を得る例として，図 8.2 と図 8.9 の例題が選ばれたとする．

ループを展開すべきかどうかという検討課題に対しては，展開したコードの方が総じて実行時間が短くなっているので，展開する．

また，エイリアス解析の困難さからポインタで参照されるメモリのレジスタ割り付けを行わないコンパイラでは,「一時変数の利用」は有効である．しかし，icc の場合はほとんど影響がないように思われるので，このような工夫は記述性を増すものでない限りは行わなくてよいことがわかる．

表 8.2 の (b) 欄や (c) 欄を見ると，途中でループを脱出しない場合には，sum の増分をまとめるのに定数部分を纏め上げてあとで足すような記述よりも，素直なものの方が高速であることがわかる．反対に，ループ脱出がある場合には，表 8.3 の (b) 欄や (c) 欄を見ると，そのような記述をしても遅くならないか，非常に高速化する場合があることがわかる．

また，SIMD 命令の中で負の値を求める演算については，表の (d) 欄と (e) 欄を検討すると，意外なことに「1 の補数に 1 を足す」式の記述の方が高速であることもわかる．

以上はあくまでもベンチマークが与える情報の一部に対する検討に過ぎない．ベンチマークを通して，このような情報を予め参考にできれば，特定コンパイラとプロセッサに対する実行速度の向上のための調整が容易かつ早くなると考えられる．

8.4 この章の結論

コンパイラによるメディア処理向け SIMD 拡張命令セットの活用に主眼を置いたベンチマークプログラムの構成法を示し，それに基づく実装と評価を行ない，提案する方式の有効性を確認した．

なお，本方式のベンチマークは，MediaBench などの既存のベンチマークを置き換えるものではなく，相補的関係にあるものである．

実装したベンチマークは，COINS プロジェクト [15] の配布物の一部として公開している．我々は，ユーザからの意見を採り入れ，利用形態の変化やハードウエア技術，コンパイラ技術の進化に合わせながら，ベンチマークを拡充していく予定である．

この研究では，例題の探索をアプリケーションプログラムのホットスポットに的を絞ったため，SIMD 命令の適用が可能な重要な例題で取りこぼしたものがあるかもしれない．N-queens 問題のようなボーダーライン上にあった問題は，より SIMD 向きの変形を探求していく．そして，この研究では浮動小数点の SIMD 命令を対象にはしなかったが，最近のメディア処理プログラムでは，IIR フィルタを用いるものをはじめとして，SSE2 や Altivec 等の SIMD 命令セットがサポートしている単精度浮動小数点演算命令によって高速化が可能なものが増えている．これらは今後の SIMD ベンチマーク拡充の方向を示している．

暗号やエラー訂正で用いられるガロア体の算術を SIMD 命令で効率良く扱う手法 [9] が公表されている．この成果がプログラムとして公表されれば，8.2.1 節で述べた候補から外された例題を，ベンチマークに加えられるようになると考えられる．

第9章　結論

本研究では，言語処理系の実装における実行性能向上のための 2 つ話題を取り扱い，成果を得た．ひとつは，記憶場所の動的な確保や開放を支援し，ユーザによる記憶場所の明示的な開放や回収を必要としない言語処理系では必須であるごみ集めの実装方式の改良である．もうひとつは，メディア処理向け SIMD 命令をコンパイラが生成するコードで有効活用するための方式である．

第 1 章では，近年の計算機に要求される，ソフトウエアの開発形態や処理内容について考察し，本論文で取り扱っている 2 つの事項が，それぞれに対する解の 1 つであることを示した．

第 2 章では，ごみ集めの問題を定式化し，印掃法と複写法というごみ集めの 2 つの基本手法を説明し，それぞれについての従来の研究成果を紹介した．そして，世代別ごみ集めの問題を定式化し，従来の手法を紹介した．

第 3 章では，印掃法で滑り圧縮を行うごみ集めの手間を，ヒープのサイズではなく，使用中オブジェクトの量に比例するようにするアルゴリズムを提案し，PLisp の処理系に対して実装し，評価した．懸念されるソーティングにかかる時間は，クラスタリングという技術を新規に開発して，ごみ集めにかかる時間を大きく変えるほどにはならないようにした．これにより，使える記憶容量の大小に関わらず，実行性能が高いごみ集め方式を得た．

第 4 章では，印掃法の滑り圧縮ごみ集めが有する生成順序保存という特性を生かした世代別ごみ集めの構成方式を提案し，PLisp に実装し，評価した．これにより，さらにごみ集めの処理時間を短縮することに成功した．

第 5 章では，SIMD 命令の説明を行ない，SIMD 命令向けコンパイラ最適化が実施すべき事項を挙げた．次に，SIMD 命令向けコンパイラ最適化という問題を，ソースレベルで実現可能な SIMD 命令向けベクタ化と，ソースレベルでは実現できない SIMD 命令向け並列化の 2 つの問題に分割統治することを提案した．そして，第 6 章で説明する最適化の内容のあらましを述べ，関連する従来研究を紹介した．また，従来のベンチマークプログラムが，SIMD 命令活用という観点からは適しておらず，それに的を絞ったベンチマークプログラムが必要であることを指摘し，既存のベンチマークプログラムを紹介し，それらが SIMD 命令活用には適さないことを示した．さらに，本研究で SIMD 最適化を実装したコンパイラ・インフラストラクチャについて説明した．

第 6 章では，SIMD 命令向けの言語仕様の拡張を用いずに，標準の言語規格の範囲内で記述されたプログラムで，SIMD 命令の活用を想定したものから，コンパイラが SIMD 命令の振る舞いに相当するオペレーションを自動抽出して，SIMD 命令を自動生成するための方式を提案し，COINS コンパイラ・インフラストラクチャを使った実装を示し，評価した．そして，限定的ではあるが，想定した通りの SIMD 命令の生成ができることを確認した．

第 7 章では，固定長のベクタレジスタを分割して並列演算を実現している SIMD 命令で問題となる事項である，高級言語の汎整数拡張の仕様が効果的な SIMD 命令の生成を阻害するという問題に対して，なるべく小さなデータサイズの演算で，汎整数拡張を行う場合と同じ結果を得るためのプログラム解析法を提案し，COINS に実装し，評価した．これにより，例題によっては並列度が上がり，サ

イズ変換のオーバヘッドが減り，実行性能が高まるようになった．

第 8 章では，従来のベンチマークプログラムでは調査することができなかった SIMD 命令向けコンパイラ最適化の性能や，SIMD 命令の実行性能の測定に特化したベンチマークプログラム集の提案と設計，実装について議論し，評価を行った．これは，ユーザに SIMD 命令の適用向きのコーディングを示しながら，コンパイラによる SIMD 命令の自動適用の度合いを評価することができるという特性を有する．

本研究の成果の一つは，記憶領域の使用効率も実行性能も高いごみ集めを提案したことである．これにより，実行時のごみ集めが必須である近代的な言語によるプログラム開発の恩恵を，主記憶領域が乏しい，例えば携帯電話の中の計算機でも享受できるようになる．もう一つの成果は，SIMD 命令向けの最適化を 2 つの段階に分割し，ソースレベルではなし得ないものについて 2 つの新しい最適化技術を提案したこと，および，SIMD 最適化技術の発展やその活用を促進することを目的とした SIMD 命令向けのベンチマークプログラムを示したことである．これによって，命令セットが異なる計算機間における，同じメディアデータ処理プログラムの共有が促進し，コーディング技術が発展することが期待される．

謝辞

本論文をまとめるにあたり，多くの方々のご指導とご協力を賜りました．ここに厚く御礼を申し上げます．

最初に，本論文を審査していただき，ご指導を賜りました論文審査委員の本学情報工学科の渡邊坦名誉教授，岩田茂樹教授，尾内理紀夫教授，岩崎英哉教授，中山泰一助教授，小林聡助教授，そして野下浩平教授に，深く感謝します．

野下浩平先生には，私の博士後期課程在学中の主任指導教官，本論文の紹介教官，それに論文審査委員会の主査をお引き受けいただきました．在学中には，技術的なことはもとより，研究者としての心構えや論文を書くという行為の厳しさと楽しさを教えていただきました．また，職を得て研究室を離れてからも，公私に渡り常に暖かく見守って下さいました．10 年前の先生との世間話で Pentium-MMX プロセッサのことが話題にならなかったら，SIMD 命令に注目していなかったかも知れません．心から感謝します．

渡邊坦先生には，私が助手として先生の研究室に所属していた間，公私に渡り本当にお世話になりました．そして，先生からお誘いいただいた COINS プロジェクトへの参加が，SIMD 最適化問題に取り組むきっかけになりました．また，本論文の草稿段階から丁寧なアドバイスをいただきました．心から感謝します．

岩崎英哉先生には，研究計画についての親身なアドバイスや，関連論文の執筆についての多くの有益なご意見をいただきました．そして，草稿段階から詳細で親切なアドバイスをいただいたおかげで，私の能力を超えて本論文を良くすることができました．心から感謝します．

中山泰一先生には，研究計画についての有益なご意見，最終論文提出までスケジュールや対応についての的確なアドバイス，それに，公私に渡り適切で親身なアドバイスをいただきました．深く感謝します．

法政大学名誉教授の中田育男先生に，深く感謝します．先生は COINS プロジェクトのリーダーでいらっしゃいました．そして関連論文執筆の際に，コンパイラ最適化の表現が難しい事項の説明について多くの有益なご意見をいただきました．

SIMD 最適化の共同研究者である，株式会社ソニーコンピュータエンタテイメントの藤波順久氏と福岡岳穂氏に，深く感謝します．SIMD 並列化は藤波氏による試験実装を元にしており，データサイズ推論の推論規則の詳細も藤波氏との議論から導き出されました．また，福岡氏には SIMD 並列化の実装で多大なご協力を頂きました．

渡邊研究室に所属されていた東京大学大学院の室田朋樹氏とセイコーエプソン株式会社の小川大介氏に感謝します．両氏には SIMD ベンチマークの研究において例題の収集や調査と例題の変形作業にご協力いただき，そのおかげでベンチマークの内容を充実させることができました．

私と同じ時期に野下研究室に所属されていた，九州工業大学の小出洋助教授に感謝します．氏との白板を使ったディスカッションが，生成順序保存と世代別管理の関係の明確化の一助となりました．

最後に，本学情報システム学研究科の故寺島元章助教授は，私の卒業研究と修士研究をご指導下さ

いました．そして，関連論文におけるごみ集めの研究のご指導をいただきました．ご冥福をお祈りすると共に，本論文をご霊前に捧げ，深い感謝の意を表します．

参考文献

[1] Abe, S., Hagiya, M. and Nakata, I.: A Retargetable Code Generator for the Generic Intermediate Language in COINS, 情報処理学会論文誌:プログラミング, Vol. 46, No. SIG14(PRO27), pp. 12–29 (2005).

[2] Allen, J., Kennedy, K., Porterfield, C. and Warren, J.: Conversion of Control Dependence to Data Dependence, *Proc. SIGPLAN Conf. on Program Language Design and Implementarion (PLDI '83)*, pp. 177–189 (1983).

[3] Allen, R. and Johnson, S.: Compiling C for Vectorization, Parallelization, and Inline Expansion, *Proc. SIGPLAN Conf. on Program Language Design and Implementarion (PLDI '88)*, pp. 241–249 (1988).

[4] Allen, R. and Kennedy, K.: *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann Publishers (2002).

[5] Appel, A.: Simple Generational Garbage Collection and Fast Allocation, *Softw.Pract.Exper.*, Vol. 19, No. 2, pp. 171–183 (1989).

[6] Appleby, K., Carlsson, M., Haridi, S. and Sahlin, D.: Garbage Collection for Prolog Based on WAM, *Comm.ACM*, Vol. 31, No. 6, pp. 719–741 (1988).

[7] Arnold, K. and Gosling, J.: *The Java Programming Language*, Addison-Wesley (1996).

[8] Bacon, D., Graham, S. and Sharp, O.: Compiler Transformations for High-Performance Computing, *ACM Computing Serveys*, Vol. 26, No. 4, pp. 345–420 (1994).

[9] Bhaskar, R., Dubey, P., Kumar, V. and Rudra, A.: Efficient Galois Field Arithmetic on SIMD Architectures, *Proc. 5th annual ACM symposium on Parallel Algorithms and Architectures (SPAA '03)*, pp. 256–257 (2003).

[10] Bik, A., Girkar, M., Grey, P. and Tian, X.: Automatic Intra-Register Vectorization for the Intel®Architecture , *Int. J. of Parallel Programming*, Vol. 30, No. 2, pp. 65–98 (2002).

[11] Boehm, H. and Weiser, M.: Garbage Collection in an Uncooperative Environment, *Softw.Pract.Exper.*, Vol. 18, No. 9, pp. 807–820 (1988).

[12] Carlsson, S., Mattsson, C. and Bengtsson, M.: A Fast Expected-Time Compacting Garbage-Collection Algorithm, *Workshop on Garbage Collection in Object–Oriented Systems (ECOOP/OOPSLA '90)* (1990). available from ftp.diku.dk:/pub/GC90/Mattson.ps.

[13] Chang, C.: The Unit Proof and the Input Proof in Theorem Proving, *J. of ACM*, Vol. 17, No. 4, pp. 698–707 (1970).

[14] Cheney, C.: A Nonrecursive List Compacting Algorithm, *Comm.ACM*, Vol. 13, No. 11, pp. 677–678 (1970).

[15] COINS コンパイラ・インフラストラクチャ協会: 並列化コンパイラ向け共通インフラストラクチャの研究,
`http://www.coins-project.org/`.

[16] Dahl, O. and Nygaard, K.: SIMULA – an ALGOL-Based Simulation Language, *Comm.ACM*, Vol. 9, No. 9, pp. 671–678 (1966).

[17] Deutsch, L. and Bobrow, D.: An Efficient, Incremental, Automatic Garbage Collector, *Comm.ACM*, Vol. 19, No. 9, pp. 522–526 (1976).

[18] Dijkstra, E.: Go To Statement Considered Harmful, *Comm.ACM*, Vol. 11, No. 3, pp. 147–148 (1968).

[19] Dijkstra, E., Lamport, L., Martin, A., Scholten, C. and Steffens, E.: On-the-fly Garbage Collection: An Exercise in Cooperation, *Comm.ACM*, Vol. 21, No. 11, pp. 966–975 (1978).

[20] Fenichel, R. and Yochelson, J.: A LISP Garbage-Collector for Virtual-Memory Computer Systems, *Comm.ACM*, Vol. 12, No. 11, pp. 611–612 (1969).

[21] Fisher, R. and Dietz, H.: Compiling for SIMD Within a Register, *LCPC '98 (Lecture Notes in Computer Science 1656)*, Springer-Verlag, pp. 290–304 (1998).

[22] Franke, B. and O'Boyle, M.: An Empirical Evaluation of High Level Transformations for Embedded Processors, *Proc. int. conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 59–66 (2001).

[23] Fraser, C., Hanson, D. and Proebsting, T.: Engineering a Simple, Efficient Code Generator Generator, *ACM Lett. Programming Languages and Systems*, Vol. 1, No. 3, pp. 213–226 (1992).

[24] Free Software Foundation: GNU gprof,
`http://www.gnu.org/software/binutils/ manual/gprof-2.9.1/gprof.html`.

[25] 藤波順久, 阿部正佳: SIMD 型拡張命令をもっと使った最適化への道のり, 第 43 回プログラミング・シンポジウム報告集, pp. 185–196 (2002).

[26] Goldberg, A. and Robson, D.: *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley (1983).

[27] Gouraud, H.: Continuous Shading of Curved Surfaces, *IEEE Trans. Compt.*, Vol. 20, No. 6, pp. 623–628 (1971).

[28] Griswold, R. and Griswold, M.: *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1983).

[29] Griswold, R., Poage, J. and Polonsky, I.: *The SNOBOL4 Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1971).

[30] Hall, M.W., Anderson, J.M., Amarasinghe, S.P., Murphy, B.R., Liao, S.-W., Bugnion, E., and Lam, M.S.: Maximizing Multiprocessor Performance with the SUIF Compiler, *IEEE Comp.*, Vol. 29, No. 12, pp. 84–89 (1996).

[31] Hejlsberg, A., Wiltamuth, S. and Golde, P.: *The C# Programming Language(2nd Ed.)*, Addison-Wesley (2006).

[32] Warren, H.S., Jr.: *Hacker's Delight*, Addison-Wesley (2003).

[33] Hwang, K.: *Advanced Computer Architecture*, McGraw-Hill, Inc. (1993).

[34] Intel Corporation: IA-32 Intel(R) Architecture Software Developer's Manual, `http://developer.intel.com/design/pentium4/manuals/index_new.htm`.

[35] Intel Corporation: Intel(R) VTune(tm) Performance Analyzer, `http://www.intel.com/software/products/vtune/`.

[36] Jonkers, H.: A Fast Garbage Compaction Algorithm, *Inf.Process.Lett.*, Vol. 9, No. 1, pp. 26–30 (1979).

[37] Jonson, M.: *Superscalar Microprocessor Design*, Prentice-Hall (1991).

[38] 吉瀬謙二, 片桐孝洋, 本多弘樹, 弓場敏嗣: PC クラスタを用いた N-queens 問題の求解, 電子情報通信学会論文誌レター, Vol. J87-D-I, No. 12, pp. 1145–1148 (2004).

[39] Knuth, D.: *The Art of Computer Programming 3rd edition vol.1* , Addison-Wesley (1997).

[40] 小出洋, 鈴木貢, 野下浩平: 生成順序の保存に基づくコピー方式世代管理の一方法, 情報処理学会論文誌, Vol. 35, No. 11, pp. 2529–2532 (1994).

[41] 小出洋, 野下浩平: 生成順序を保存するコピー方式ガーベジコレクションについて, 情報処理学会論文誌, Vol. 34, No. 11, pp. 2395–2400 (1993).

[42] 小藤哲彦: パーザ・ジェネレータにおける構文木生成法の研究, 博士論文, 電気通信大学大学院電気通信学研究科 (2005).

[43] Kurokawa, T.: A New Fast and Safe Marking Algorithm, *Softw.Pract.Exper.*, Vol. 11, No. 7, pp. 671–682 (1981).

[44] Larsen, S. and Amarasinghe, S.: Exploiting Superword Level Parallelism with Multimedia Instruction Sets, *Proc. ACM SIGPLAN 2000 Conf. on Programming Language Design and Implementation(PLDI '00)*, New York, NY, USA, ACM Press, pp. 145–156 (2000).

[45] Lee, B. and John, L.: NpBench: A Benchmark Suite for Control plane and Data plane Applications for Network Processors, *Proc. Int. Conf. on Computer Degisn (ICCD '03)* (2003). `http://www.iccd-conference.org/proceedings/2003/20250226.pdf`.

[46] Lee, C., Potkonjak, M. and Mangione-Smith, W.: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems, *Proc, 30th annual ACM/IEEE Int. Symposium on Microarchitecture (MICRO 30)*, IEEE Computer Society, pp. 330–335 (1997).

[47] Leupers, R.: Code Selection for Media Processors with SIMD Instructions, *Proc. Design, Automation and Test in Europe (DATE'00)*, pp. 4–8 (2000).

[48] Leupers, R. and Bashford, S.: Graph-Based Code Selection Techniques for Embedded Processors, *ACM Trans. Design Automation of Electronic Systems*, Vol. 5, No. 4, pp. 794–814 (2000).

[49] Liberman, H. and Hewitt, C.: A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Comm.ACM*, Vol. 26, No. 6, pp. 419–429 (1983).

[50] Liskov, B. and Guttag, J.: *Abstraction and Specification in Program Development*, MIT Press (1986).

[51] McCarthy, J. et al: *LISP 1.5 Programmer's Manual*, MIT Press (1962).

[52] Memik, G., Mangione-Smith, W. and Hu, W.: NetBench: a Benchmarking Suite for Network Processors, *Proc. 2001 IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD '01)*, Piscataway, NJ, USA, IEEE Press, pp. 39–42 (2001).

[53] Miyakawa, M.: bmp2png/png2bmp home page, `http://cetus.sakura.ne.jp/softlab/b2p-home/`.

[54] 森公一郎, 阿部正佳, 中田育男: 21 世紀のコンパイラ道しるべ（LIR の説明とバックエンドの概要説明）, 情報処理学会学会誌, Vol. 47, No. 6, pp. 662–669 (2006).

[55] 森公一郎, 阿部正佳, 中田育男, 鈴木貢: 21 世紀のコンパイラ道しるべ（TMD によるコード生成 — SPARC0 を例題として）, 情報処理学会学会誌, Vol. 47, No. 7, pp. 776–789 (2006).

[56] Morris, F.: Time- and Space-Efficient Garbage Collection Algorithm, *Comm.ACM*, Vol. 21, No. 8, pp. 662–665 (1978).

[57] 中島浩, 近山隆: スタック領域が不要な深さ優先順コピー型ゴミ集め方式, 情報処理学会論文誌, Vol. 36, No. 3, pp. 687–713 (1995).

[58] 中田育男: コンパイラの構成と最適化, 朝倉書店 (1999).

[59] 中田育男, 渡邊坦: 21 世紀のコンパイラ道しるべ（HIR の説明と簡単な言語のフロントエンド）, 情報処理学会学会誌, Vol. 47, No. 5, pp. 526–539 (2006).

[60] 中田育男, 渡邊坦: 21 世紀のコンパイラ道しるべ（概要）, 情報処理学会学会誌, Vol. 47, No. 4, pp. 425–436 (2006).

[61] 朴少林, 鈴木貢, 渡邊坦: 述語付き命令を持つ計算機における条件変換の静的最適化方式, 「ハイパフォーマンスコンピューティングとアーキテクチャの評価」に関する北海道ワークショップ (HOKKE-2002), pp. 103–108 (2002).

[62] Sahlin, D.: Making Garbage Collection Independent of the Amount of Garbage, Technical report, SICS, Box 1263 S-163 13 SPÅNGA SWEDEN (1987). Research Report R86008.

[63] Schorr, H. and Waite, W.: An Efficient Machine-Independent Procedure for Garbage Collection in Various Lust Structures, *Comm.ACM*, Vol. 10, No. 8, pp. 501–506 (1967).

[64] Shaw, R.: *Empirical Analysis of a LISP System*, PhD Thesis, Stanford University, Stanford, CA, USA (1988).

[65] Sreraman, N. and Govindarajan, R.: A Vectorizing Compiler for Multimedia Extensions, *Int. J. of Parallel Programming*, Vol. 28, No. 4, pp. 363–400 (2000).

[66] Standard Performance Evaluation Corp.: SPEC CPU2000, `http://www.spec.org`.

[67] Steele Jr., G. L.: Multiprocessing Compactifying Garbage Collection, *Comm.ACM*, Vol. 18, No. 9, pp. 495–508 (1975).

[68] Steele, Jr., G.: *Common LISP, 2nd ed.*, Digital Press (1990).

[69] Stephenson, M., Babb, J. and Amarasinghe, S.: Bitwidth Analysis with Application to Silicon Compilation, *Proc. the SIGPLAN Conf. on Program Language Design and Implementarion (PLDI '00)*, pp. 108–120 (2000).

[70] Sun microsystems: VIS Instruction Set, `http://www.sun.com/processors/vis/`.

[71] 鈴木貢, 寺島元章: 可変長セル用並列・実時間ごみ集め, 情報処理学会プログラミング–言語・基礎・実践研究会 3-20 (1991).

[72] 竹内郁雄: Lisp 処理系コンテストの結果, 情報処理学会記号処理研究会報告 5-3 (1978).

[73] Terashima, M. and Goto, E.: Genetic Order and Compactifying Garbage Collectors, *Inf.Process.Lett.*, Vol. 7, No. 1, pp. 27–32 (1978).

[74] 寺島元章: PHL の新インタプリタ, 情報処理学会研究会報告 SYM 73-5, pp. 33–40 (1994).

[75] 寺島元章, 佐藤和美: 可変容量セルの効率的なくず集めについて, 情報処理学会論文誌, Vol. 30, No. 9, pp. 1189–1199 (1989).

[76] Ungar, D.: Generation Scavenging: A non-Disruptive High Performance Storage Reclamation Algorithm, *Proc. 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1)*, New York, NY, USA, ACM Press, pp. 157–167 (1984).

[77] 渡邊坦: コンパイラの仕組み, 朝倉書店 (1998).

[78] Weicker, R.: Dhrystone: a Synthetic Systems Programming Benchmark, *Comm.ACM*, Vol. 27, No. 10, pp. 1013–1030 (1984).

[79] Wilson, P. R. and Moher, T. G.: Design of the Opportunistic Garbage Collector, *Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA '89)*, New York, NY, USA, ACM Press, pp. 23–35 (1989).

[80] Zima, H. and Chapman, B.: *Supercompilers for Parallel and Vector Computers*, Addison-Wesley (1991).

[81] Zivojnovic, V., Martinez, J., Schlager, C. and Meyr, H.: DSPstone: A DSP-Oriented Benchmarking Methodology, *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT '94)* (1994).
`http://www.iss.rwth-aachen.de/Projekte/Tools/PAPERS/cad_Zivojnovic94icspat.ps.gz`.

関連論文の印刷公表の方法及び時期

1. Mitsugu Suzuki and Motoaki Terashima:
   “Time- and Space- Efficient Garbage Collection Based on Sliding Compaction,”
   情報処理学会論文誌, 36 巻, 4 号, pp.925-931 (1995).
   （第 3 章の内容）

2. Mitsugu Suzuki, Hiroshi Koide and Motoaki Terashima:
   “MOA - A Fast Sliding Compaction Scheme for a Large Storage Space,”
   Proceedings of Memory Management: International Workshop (IWMM95), Lecture Notes in Computer Science, Vol. 986, Springer-Verlag, pp.197-210 (1995).
   （第 4 章の内容)

3. Mitsugu Suzuki, Nobuhisa Fujinami, Takeaki Fukuoka, Tan Watanabe and Ikuo Nakata:
   “SIMD Optimization in COINS Compiler Infrastructure,”
   Proceedings of International Workshop on Innovative Architecture for Future Generation High Performance Processors and Systems (IWIA 2005), pp.131-140 (2005).
   （第 6 章, 第 7 章, 第 8 章の内容）

4. 鈴木 貢, 藤波 順久:
   “コンパイラの中間表現から SIMD 命令への変換の一手法について,”
   電子情報通信学会論文誌 (D), J90-D 巻, 3 号, (2007).（印刷中）
   （第 6 章の内容）

5. 鈴木 貢, 藤波 順久, 福岡 岳穂, 渡邊 坦, 中田 育男:
   “マルチメディア SIMD 命令活用のためのデータサイズ推論,”
   情報処理学会論文誌:プログラミング, 45 巻, SIG 5(PRO 21) 号, pp.1-11 (2004).
   （第 7 章の内容)

6. 鈴木 貢, 小川 大介, 室田 朋樹, 渡邊 坦:
   “SIMD ベンチマークの設計と実装,”
   情報処理学会論文誌:コンピューティングシステム, 46 巻, SIG16(ACS12) 号, pp.95-107 (2005).
   （第 8 章の内容）

著者略歴

鈴木　貢 （すずき みつぐ）

1964 年　　東京都に生まれる

学歴

1989 年 3 月　電気通信大学電気通信学部 卒業
1989 年 4 月　電気通信大学電気通信学研究科博士前期課程 (情報工学専攻) 入学
1991 年 3 月　電気通信大学電気通信学研究科博士前期課程 修了 (工学修士)
1991 年 4 月　電気通信大学電気通信学研究科博士後期課程 (情報工学専攻) 入学
1995 年 3 月　電気通信大学電気通信学研究科博士後期課程 単位取得満期退学

職歴

1989 年 4 月　国立予防衛生研究所 (現 国立感染症研究所) 臨時職員
1995 年 3 月　同所退職
1995 年 4 月　電気通信大学電気通信学部情報工学科 助手
　　現在に至る