記　号　処　理　61－1
（1991.　9.　13）

新しい記号処理カーネル TAO/SILENT の設計

竹内郁雄　吉田雅治　天海良治　山崎憲一

NTT 基礎研究所, NTT ヒューマン・インタフェース研究所, NTT ソフトウェア研究所

現在設計中の記号処理カーネル TAO/SILENT の実時間性について述べる．TAO/SILENT はヘテロ結合した AI システムの記号処理を担当する部分として位置付けられる．実際，ロボットを制御するための脳は重要な応用の一つである．このようなシステムでは外界との相互作用を行なうため，実時間性が重要となる．TAO/SILENT は実時間性を提供することを目標の一つとしてその全体が設計されている．

本稿では，まず TAO/SILENT の概要について述べた後，コンカレンシ機能および実時間プリミティブについて述べる．これらは実時間アプリケーションの記述に必須の機能である．最後に，実時間ゴミ集めのアルゴリズムとそれに関連した話題について述べる．実時間ゴミ集めを行ないながら，どの程度のハード実時間処理を保証するかがキーポイントであるが，現在のところ，割り込み禁止時間100 μ秒以下を目標として設計が進められている．

Design of a new symbolic processing kernel TAO/SILENT

Ikuo Takeuchi　Masaharu Yoshida　Yoshiji Amagai　Kenichi Yamazaki

NTT Basic Research Laboratories, NTT Human Interface Laboratories,
NTT Software Laboratories

This paper describes the real-time facilities of a symbolic processing kernel TAO/SILENT which is being designed. TAO/SILENT is expected to be a symbolic processor in a heterogeneous AI system such as the brain of a robot. For such a system that interacts with the external environment, real-timeness is a very important facility.

In this paper, an overview of TAO/SILENT is briefly described at first. Then its concurrent facilities and real-time primitives are described. Finally, the real-time garbage collection and the related topics are discussed. It is a key point for the real-timeness of a symbolic processing system to what extent to guarantee the hard real-timeness along with garbage collection. At the current stage of the TAO/SILENT design, interrupt disabled interval is estimated less than 100 microseconds if the system is not so heavily loaded.

1. はじめに

TAO/SILENT は，タグ部 8 ビット，ポインタ部 32 ビットの 40 ビット Lisp マシン SILENT 上に，豊富な記号処理プリミティブとコンカレンシ・プリミティブの集成体である機械語 TAO が載ったシステムであり，現在設計・開発を進めているものである [1,2,3,4,5,6]．TAO/SILENT は我々が以前開発した TAO/ELIS[7,8] の延長線上にあるシステムであるが，マシンアーキテクチャも言語も互換性はない．言語 TAO の名称は引き継いでいるが別物なので，本稿では ELIS 上の TAO のことを DAO と呼ぶ 1)．

我々は TAO/SILENT を記号処理カーネルと呼んでいるが，これは人間の脳のようにヘテロ結合した知能的機能複合体における記号処理担当部分といった意味である．言い換えれば，ヘテロマルチプロセッサとして構成された知能システムにおける「記号処理エンジン」という意味を込めている．実際，知能ロボットの「脳」の中枢部分への応用は TAO/SILENT の重要目標の一つである．(このほか，新しい AI プログラミングパラダイムの研究や並列・分散プログラミングの研究のプラットフォームとしての応用が重要な目標である.) つまり，TAO/SILENT の目指しているものを一言でいうと「どこにでもプラグイン可能な実時間 AI エンジン」である．本稿では，このうち TAO/SILENT の目指している実時間性について述べる．

2. 実時間性

2.1 実時間システム

実時間システムとは，仕事の処理の開始時間や終了時間などの時間制約が記述でき，その遵守が期待できるシステムのことである．実時間システムは大きく 2 種類に分類される．一つは時間制約が守れなかった場合，致命的なエラーが生ずるもので，ハード実時間システム [9] と呼ばれる．それ以外のシステムはソフト実時間システムと呼ばれる．後者では時間制約に対する違反は，性能の低下などを招くもののエラーとはならない．難しいのは当然ハード実時間システムであり，現在のホットな研究対象である．

ハード実時間システムで重要なことはシステムの挙動の完璧な予測可能性である．すなわち，時間制約が守れるのか否かが前もって予測できなければならない．実時間システムというと性能さえ高ければよいなどといった間違ったイメージがあるが，たとえ速度が遅くてもシステムの挙動が完全に予測可能であれば，それはハード実時間システムであるというのが今日の一般的見解である．

1) これは olD tAO のつもりであるが，昔の中国で「道」を意味した TAO の現代発音は「ダオ」なので，本当はこちらのほうが新しいのかもしれない．

実時間 OS と呼ばれる既存の OS は基本的な性質として次の 2 点を備えている．

(1) 割り込み応答遅延時間の保証.
(2) 優先度に基づく先取り可能スケジューリング.

割り込み応答遅延時間 (以下，単に応答遅延時間という) とは外界でイベントが発生して，それがシステムに通知されてから，ユーザが指定したルーティンが起動されるまでの最悪時間のことである．実時間システムでは外界との相互作用が本質的である．外界との相互作用がなければ，システムの挙動が予測可能であることは明らかである．なぜなら，システムを一度リハーサルさせればよいからである．従って応答遅延時間が保証されることはシステム全体の挙動を予測可能とするために極めて重要である．先取り可能スケジューリングは複数のタスクがあったときに，どれがいつ実行されるのかを予測するのに必須の機能である．

OS の提供する機能としてはこの 2 点で十分であるが，OS が実時間であるということと，アプリケーション全体として実時間性を満たすように構築できるということとは別の問題である．すなわち，外界の状況の変化 (割り込み) を認知すること (割り込み応答) と，それに対して的確な判断を下して対応動作を行なうこと (ここでは，「割り込み応答処理」と呼び，それに要する時間を「割り込み応答処理時間」あるいは単に「応答処理時間」と呼ぶ) は別の問題なのである．応答処理時間に対する制約は期限 (deadline) と呼ばれ，最近のハード実時間システムの研究でクローズアップされている．

これまで実時間アプリケーションとは主に機器の制御のことであった．このような定型的な処理は応答処理時間が前もってわかるので設計がしやすい．(その場合ですら結局はシミュレーションや実機上でのテストの繰り返しが必要となる.) このような分野では，OS のサポートとしてどのようなプリミティブが必要となるかも，ほぼ決まっている．

しかし，知的ロボットを考えると，腕や足などの制御系に加え，それら全体を知的に制御する「脳」が必要となる．もちろん脳も実時間システムでなければならないが，AI システムを実時間化するにはさまざまな問題を解決しなければならない．最も難しい点は AI においては応答処理時間が予測できないことである．この問題については期限に応じた準最適解を求める戦略などの研究が行なわれているが，個々のアプリケーションごとにアドホックな方法が使われている状況である．

実時間 AI システムすなわち実時間記号処理システムのもう一つの問題はゴミ集めである．一般にゴミ集めは秒あるいは分オーダで割り込みを禁止するため，応答遅延

時間が致命的に大きくなってしまう.

2.2 TAO/SILENTのアプローチ

我々のアプローチはTAO/SILENTを上に述べた意味での「古典的ハード実時間システム」とすることではない. 実時間AIは, 時間制約と解の要求条件に対して厳密な意味での最適解をつねに保証するというアプローチでは実現できないと我々は考える. つまり, ハード実時間システムでも, 過負荷の割り込みにはちゃんと対応できないのと同様, 記号処理やAIでの (ほとんどの場合, 病的ともいえる) 過負荷の内部処理に対して, TAO/SILENTはハード実時間であることを保証することを諦める. 「健全な知能は, 健全な負荷に宿る」は知能の本質だろう.

この意味で, Brooksの包摂アーキテクチャ (subsumption architecture)[10] は, (自己保持のための) 安全第一の反射的動作のレベルから, より高度な知能的判断を要するレベルまでの階層分けを考えたアイデアで, 実時間AIシステムの作り方に有望な示唆を与えている. これは反射運動レベルではハード実時間でありながら, 知的レベルの高いところではソフト実時間になっているという意味で, セミハード実時間ともいえる.

TAO/SILENTは, それ自身だけで (あるいはそれと結合された異種の, いわば脊髄プロセッサなどの助けを借りて) ここでいうセミハード実時間システムを実現することを目指す. すなわち, 適正な負荷 (たとえば, 並行プロセスの数が200個程度以下で, 長大な文字列の内部置換をマイクロのプリミティブを使ってはやらないなどといった制限) のもとでは, システムとして応答遅延時間を保証し, また応答処理時間を個々のアプリケーションで保証しやすいようなプリミティブを用意するという設計方針をとる.

3. TAO/SILENTの概要

ここでごく簡単にTAO/SILENTを紹介する.

SILENTは専用VLSI (SILENTチップ) の載ったボードコンピュータであり図1のような構成になっている. 外部インタフェースはSCSI, FDDI, Etherといった標準のもののほか, 80ビットシステムバス, 32ビット外部バスがある. 80ビットシステムバスは高速浮動小数点演算器TARAI[11] など, SILENT専用のコプロセッサとの接続用である. 32ビット外部バスは異種プロセッサとの接続用であり, 種々の32ビット系マイクロプロセッサとの接続を念頭においている. SILENTの第1版には, 最大256K語の (循環アドレスをもつ) ハードウェアスタックが装備されている.

SILENTには専用のコンソールやディスプレイがない. AIワークステーションとして使いたいときは, 上記インタフェースのどれかを使って好みのワークステーションとつなぐわけである (SCSIインタフェースを使うのが最も手軽であろう). なお, ディスクも標準ではSILENTにつながらない.

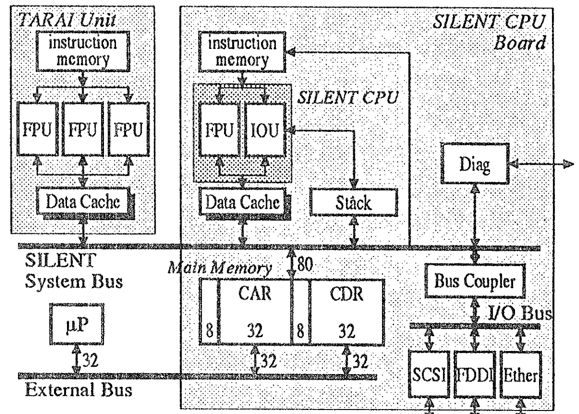

図1. ボードの概念図

SILENTの設計は, ELISのアーキテクチャを踏襲するところから始まったが, 最終的にはまったく互換性のないものになった. SILENTチップ第1版は1991年度内に完成する. これはマシンサイクルは30ナノ秒. つまり, レジスタ+レジスタ→レジスタ (スタックを含む) の演算が30ナノ秒という性能をもつ. キャッシュメモリへのアクセス速度も30ナノ秒である.

4. コンカレンシ

我々が目指すような実時間性を達成するためには, まず強力なコンカレンシ (並行プログラミング機能) が必要である. すなわち, 複数のイベントに対応する複数のプロセスが同時に走り, 外部からの割り込みに対して迅速に反応し, 複数のプロセスに機会均等にCPUパワーなどのリソースを割り振るような機能が必要である. TAO/SILENTでは, TAOが機械語であるから, コンカレンシの制御のうちマイクロコードで書かれていないものはすべてTAO自身で書かなければならない. このため, TAOには強力なコンカレンシ・プリミティブの集合が用意される. これらの多くはDAOのもの[12] を引き継いでいるが, 性能向上を図るためと, 概念整理のためにリファインされたものがある. ここでは, 実時間性に関するところに焦点を当てて述べることにする.

4.1 プロセスとスレッド

TAOはDAOのプロセスをさらに軽くするために, プロセスの概念とスレッドの概念を分離した. 一言でいうと, スレッドは実行環境スタックに対応する概念であり, プロセスは (複数の) スレッドの共通の根となる環境である (多少紛らわしいが混同のないかぎり, これからプロセ

スとスレッドを組み合わせたものもプロセスと呼ぶことがある). スレッドは実行環境スタックを枝分かれさせるかのように, 複数個のスレッドを生むことができるので, 根であるプロセスから図2のようにスレッドが木の形に生える. ある時間に実際に走るのは, あるプロセスのスレッドの木の末端の一つのスレッドであり, そのときの実行環境スタックは, そのスレッドから根のプロセスへのパスにあるスレッドの実行環境スタックをつなげたものである (たとえば, 図2で灰色で示した部分). このことから明らかなように, 子スレッドを生んだ親スレッドが走れるのはすべての子スレッドが終了してからである. しかし, 一つでも子スレッドから制御が帰ってきたときにほかの兄弟スレッドをすべて強制終了させるOR分岐や, どれか二つから制御が帰ればOKとか, 特定の子スレッドからすべて制御が帰ってくればOKとか, 多種多様のスレッド分岐法があり得よう. TAOは可能なかぎり多くの分岐法に対処できるプリミティブを用意する予定である.

一つのスレッドの生成時間は典型的には数10 μ 秒程度である.

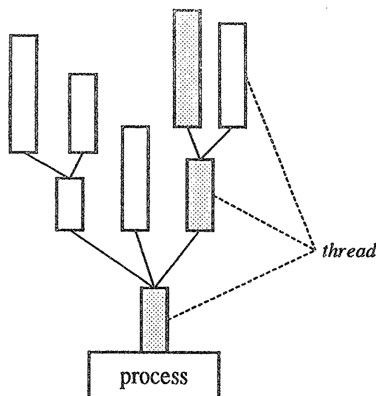

図2. TAOのスレッド

4.2 優先度管理

プロセスあるいはスレッドの優先度をどのように管理するかは, 実時間処理にとって大きな問題である. 動的に生成・消滅するすべてのスレッドに全順序がつけられるような優先度管理から, 大ざっぱに数レベルの優先度しか与えないものまでに効率よく対応するのは実装上困難であろう. TAO/SILENTでは, プロセスキュー管理が容易で高速な32段階の優先度を用意する. 同じ優先度のプロセスはFIFOキューに並ぶ. このFIFOキューはTAOの基本データ型で, マイクロコードで実装される (ついでながら, LIFOスタックも基本データ型である).

TAO/SILENTでは32の優先度を, 応答遅延時間によって大ざっぱに次の4レベルに分ける.

U: 100 μ 秒程度 (urgent)
H: 1ミリ秒程度 (high)
M: 10ミリ秒程度 (medium)
L: 100ミリ秒程度以上 (low)

ここでレベルLを除き, 応答遅延時間が T であれば, 実際の応答処理期限 (deadline) は高々 $2T \sim 3T$ であることが想定されている. たとえば, 人間のキーボード入力割り込みに対するエコーバックの優先度はMといったところであろう. 同じイベントに対する応答を, 包摂アーキテクチャをシミュレートしたようなプロセスの集合体で行なうとき, このレベル分けが包摂アーキテクチャのレベル分け設計の指針となるであろう.

4.3 多重度

スレッドは最大256K語の循環スタックの適当な位置に割り付けられる (ただし, 一つのプロセスから生えたスレッドの最大深さ, つまりスレッドの木の高さは128K語である). こうすると当然スレッド同士で衝突が起こるが, これは基本的には身軽なほうのスレッドが転居することで解消する. これでも解決しないほど循環スタックが混み合ってきたときは, 主記憶へスレッドが待避される. こうなった場合には, 実時間性のグレードが大幅に落ちることになる.

DAOでの経験からすると単純なプロセスは高々500語程度のスタックで走るので, 単純な計算で最大500個のスレッドが主記憶とのスワップなしに動くことになる. 実際には, 2～300個程度のスレッドが実時間性の保証できる範囲であろう. もっとも, このあたりの保証は理論的解析が困難なので, シミュレーションあるいは実使用経験の統計的解析を待たなければなるまい. 実時間制御の問題が定型的で, 解析が十分に行なわれているのであれば, なんの問題もないのはもちろんである.

ハードウェアスタックの上に載っているスレッド同士の間でCPUリソースを移すのに要する時間, すなわちコンテクストスイッチ時間は約5 μ 秒である.

4.4 割り込み

TAO/SILENTの割り込みの受け付けは2段階になっている. 第1段はハードウェアレベルでの受け付けで, 割り込みが起こった瞬間にマイクロコードの実行が中断され, ハードウェア割り込みルーティンが走る. これはもちろんマイクロコードレベルである. ハードウェアの致命的エラーの場合などはこの第1段で必要な処理の大半を終える. このレベルはTAOの記号処理にはまったく干渉しない. 通常の割り込みにおいて, 第1段の仕事はTAOが割り込みを認知できるように御膳立てすることである.

TAOの操作を記述しているマイクロコードには当然の

ことながら非可分なコード列がある．これはTAOの扱っているデータ構造の整合性を保証するために必要なものである．たとえば，cons演算の真っ最中にコンテクストスイッチをするわけにはいかない．そのため，マイクロコードは割り込みを検知する条件分岐（hap条件分岐と呼ぶ ― happenから来ている）を使って，割り込みを陽に受け付ける．マイクロコードはTAOという機械語から見ればハードウェアであるから，これは自然な考え方であろう．割り込み受け付けの第1段は，必要な情報をシステムテーブルに書き込むとともにこのhap条件を立てることである．hap条件が検知されて始めて第2段の割り込み受け付けが可能になる．なお，いくつかの典型的な割り込みは，第1段の処理がハードウェアで行なわれ，自動的にhapが立てられる．これらはスタックオーバーフロー，入出力チャネル割り込み，タイマー割り込みの3種類である．

実時間性の観点からここで一番重要なのは，hapが立ってから実際にそのhapが検知されるまでの遅延時間，すなわちTAO/SILENTの種々の操作の非可分コード（hapチェックから次のhapチェックまでの間のコード）の長さである．我々はこれを100μ秒以下にする目標を立てている．現在のところ，ほとんどのところでこの目標の達成に大きな障害はない．しかし，ありとあらゆる場面でこの値をTAOの言語としての整合性を保ったまま，絶対的に保証することが不可能なのも事実である．たとえば，長さ10,000以上の文字列の途中の“aabcc”を“abc”に変更するような文字列処理を行なった場合（これはTAOの基本演算として用意されている），非常に手間のかかるダルマ落し処理を行なわないといけない．これを非可分操作とすればおおよそミリ秒程度の遅延となる．しかし，非可分操作としなければ，この文字列を共有しているプロセスの間で排他制御を行なわないとデータの整合性が保証されない．この排他制御をシステムが自動的に行なうのか，ユーザが行なうのかも問題となる．このような例外的に負荷の大きい処理が実時間処理に意味があるかどうかも含めて，若干のトレードオフ問題が残っている．

4.5 時間量子

時分割の時間単位（時間量子，time quantum）は，TAO/SILENTの性能バランスや実際の応用局面を考えて，500μ秒あるいは1ミリ秒程度にする予定である．事実上，これ以下の時間量子に意味があるような局面は，少なくとも実時間AIエンジンの応用としてはないだろう．ちなみにTAO/ELISの時間量子は20ミリ秒であった．

4.6 実時間プリミティブ

DAOで予想以上に有効であったコンカレンシ・プリミティブに，タイムアウト付きのプリミティブがある．たとえば，

(p-sem *semaphore*)；セマフォを取る
(receive-mail *mailbox*)；メールボックスの情報を取る

に対して

(p-sem-with-timeout *clock-ticks semaphore*)
(receive-mail-with-timeout *clock-ticks mailbox*)

といったように，待ちに期限をつけるものである．これらは，対話ユーザを想定した

(read-char-with-timeout *clock-ticks stream*)

などと同様，複数プロセス間で実時間的処理を進めるときにきわめて有効である．TAOでもこれは踏襲する．

4.7 負のイベント

DAOのTCP/IPプログラミング[13,14]で有効だったものに，負のイベントによる割り込みがある．これは，「あるイベントが一定時間起こらなければ割り込む」という割り込みである．たとえば，Telnetサーバがバッファに溜っている情報をどのタイミングでパケットに組み立ててフラッシュアウトするかという問題がある．機械的に頻繁に出せば，ネットワーク負荷が重くなり，バッファーにある程度溜ってからというのでは，対話処理に向かない．これには一定時間バッファの情報の増分がなかったときというタイミングが最も適しているであろう．つまり，負のイベントが重要になる．

DAOでは，負のイベント割り込みは入出力ストリームに対してしか用意しなかったが，TAOではこれを一般化して適用範囲を広げたプリミティブとして提供する予定である．

5. 実時間GC

記号処理を実時間化するにあたって一番問題になるのが，ゴミ集め（GC, Garbage Collection）である．実時間GCのアルゴリズムは数多く提唱されてきたが，実際のシステムの中できちんと動かすのはいまでもそれほど容易ではない．記号処理システムにはリストだけではなく，ベクタ，文字列など種々の構造をもったデータ型が共存しているため，モデル的なアルゴリズムだけでは歯が立たない雑多な問題が結構あるからである．

プログラミングの立場からいうと，GCはLispのような記号処理言語をコンベンショナルな手続き型言語と差別化する重大な機能である．しかし，実時間的な応用ではこれが最大の障害になるというのは皮肉である．逆にこれを乗り越えたときにこそ，記号処理は情報処理における真に実用的で重要なジャンルに昇格すると考えられる．

少し話は脱線するが，「AIプログラムとはGCが動くプログラムのことである」というテーゼはあながちウソではなさそうである．ここで「GCが動くプログラム」

とは，自動的なゴミ集めがないととても組む気のしないプログラムというくらいの意味である．Lisp や Prolog が AI プログラミングに使われてきたのは，まさにこの理由からだろう．

さらに話は脱線するが，人間の知能の本質は，脳における新陳代謝とあながち無関係ではあるまい．つまり，いつも「ゴミ」を出すような処理を行なっているから，全体として知能のようなものが発現すると考えられないだろうか．この伝でいうと，人間が起きているあいだ常時起こっている新陳代謝はいわば実時間 GC であるし，睡眠は実時間 GC では間に合わないグランド GC である．従来の一括型 GC は，上の譬えでいえば睡眠に相当するが，困るのはいつどれくらい睡眠するか予測がつかないことである．たとえ睡眠を取るにしても，それが予測可能であれば，実時間応用には十分使えるともいえる．実際，ギガセル程度の Lisp マシンであれば，1 日 8 時間フル労働のロボットが勤務時間中に GC で眠ってしまうことはないだろう．

さて，記号処理を実時間化するには，実時間 GC を実現することのほかに，ゴミを出さないような記号処理の工夫をする対策もある．実時間 GC を行なうにせよ，なるべくゴミが出ないようにシステムを設計することは実時間性を高める側面対策にはなる．TAO は新しい実時間 GC アルゴリズムを実装するとともに，この側面対策も十分に行なう．

5.1 実時間 GC アルゴリズムの概略

TAO/SILENT の実時間 GC は，TAO/ELIS で実現するはずだったものと基本的には同一である (未発表)．TAO/ELIS では，下に示したように，プロセス A が GC を陽に起動する関数を無限ループで回し，プロセス B が 10,000 回の cons 毎に 1 文字を端末に打ち出す無限ループを回すという実験を行ない，ほぼ規則正しく連続的に文字が打ち出されるところまで確認した．

```
Process A: (loop (gc))
Process B: (loop (for i (index 1 10000) (cons 1 2))
                 (write-char #\*) )
```

しかし，最後まで GC プログラムをリファインする時間がなく，実時間 GC の基本動作の確認のみに終わっている (1989)．もっとも，DAO では文字列本体の GC がポインタの破壊的張り替え方式によるコンパクションであったため，文字列の文字数が (メガバイト程度に) 多いと，コンパクションのための非可分セクションが 500 ミリ秒〜1 秒という長さになり，いずれにせよ高い実時間性は実現不能であった．

さて，TAO/SILENT の実時間 GC アルゴリズムは一括型 GC でよく知られた Mark and Sweep 方式を実時間向きに手直ししたものである．以下にその概要を述べるが，話を複雑にしないためにコンスセルのみに話を制限し，システム内部構造を簡略化したモデルを使う．なお，TAO/SILENT は文字列もベクタもコンパクションしない．これは GC の中での非可分セクションの長さを 100 μ 秒以下にするためである．

TAO/SILENT のセル領域はおおむね図 3 のような形をしている．セルの一番左の 1 ビット，つまり car 側のタグの MSB がマークビットである．ここでハッチの入ったセルは使用中のセルを示している．

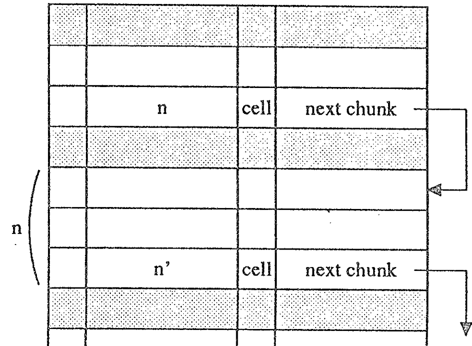

図 3. TAO のセル領域

連続した自由セルはチャンク (chunk) と呼ばれ，cons を行なう際，自由セルのチェインをたどる必要がないようになっている．二つのレジスタ freeccZ (free cell chunk size の意味) と freec (free cell の意味) があり，freeccZ は自由セルチャンクの残りセルサイズ (セルの数) を表わし，freec は一番上にある自由セル，つまり次に取るべき自由セルを指している．ここで，チャンクの一番最後にだけ，次のチャンクへのチェインポインタと，次のチャンクのセルサイズが記入されていることに注意しよう．

このような構造により，cons が自由セルチェインをたどるために書き込もうとしているセルの中身を読み出す必要がないことが大半である．つまり，cons が速くなる．また，GC は自由セルを回収する際，チャンクの一番最後のセルにだけ書き込みを行なえばよい．これは GC の回収フェーズをかなり速くする．

このほかに，回収済みセルの最前線を示す gc-cell-front (回収走査前線) というレジスタがあり，回収フェーズで使われる．

SILENT の psw (Processor Status Word) の bit3-2 は GC に関するモードを表わし，それをもとにマイクロコードはいつでも 3 方向条件分岐ができる．三つのモードは次のとおりである．

- 消費モード: GC は休んでいる．一般に自由セルの残量

がたっぷりある状態.

- マーキングモード: GC がマーキングを行なっている.
- 回収モード: GC がセルを回収している.

消費モードで自由セルの残量が少なくなると, GC が起動され, マーキングモードへ状態が遷移する (自由セルの残量のチェックは cons 毎ではなく, 1～2万回の cons に1回という仕掛けになっている). GC は並行プロセスの一つであり, ほかのたくさんの TAO プロセス (いわゆる mutator) と並行に走る. つまり, GC といえども TAO/SILENT のスケジュール管理や割り込み処理を受ける.

マーキングモードでは, GC プロセスを除く全 TAO プロセスとそのスレッド (つまり, 実行環境スタック — ここに変数の値などが入っている) がマークされる. 最後にいくつかの重要なシステム変数がマークされる. これらがマーキングのルートとなるわけである. マーキングはほかのプロセスも並行に走るという状況で行なわれる (ただし, バックグラウンドジョブのように優先度の非常に低いプロセスは GC が終了するまで走らない). すなわち, マーキングは時分割で行なわれる. マークしている最中のスレッドが走ったら, そのスレッドはまた最初からマークし直される. スレッドが木構造になっているときは, すべての子スレッドがマークを終わって始めて親スレッドのマークに行く. こうして, すべてのスレッドとプロセスがマークされたら, 回収モードに遷移する.

時分割 GC なので, 実際, GC の最中に TAO プロセスはよく走る. だから, GC は TAO プロセスが走るたびにその部分のマークをやり直すことになる. しかし, 次第にマーク済みのデータが増えていき, GC がやがて TAO プロセスに追いつくというのが, 大前提である. これはきちんとした証明を要することであるが, 記号処理プロセスよりも GC のほうがセルを相当速く舐めていくのは事実である. たとえば, TAO/ELIS では 16 メガセル空間での一括型 GC は典型的には 15～20 秒かかるが, この空間を使い尽す Lisp プログラムは速いものでもその 10 倍以上の時間がかかる. もちろん, なかなか追いつかない場合は, GC プロセスの優先度を上げてマーキングの速度を上げる方策をとる.

スタックメモリには 128 または 256 語毎にハードウェアのダーティフラグがついているので, マークをやりなおすときに変更のなかった部分はマークのために舐め直す必要がない. スタックメモリ自身の走査も高速に行なえるので (たとえば, マークの必要のない即値データが並んでいる場合は 1 語あたり 60 ナノ秒), 実際にはマーキングプロセスはかなり速くほかのスレッドやプロセスに追いつく.

なお, マーキング中は後述するように GC ポストにマークすべきデータが次々に投函されてくる. これも適宜マークしてポストから除いていく. GC ポストが満杯になったら, GC ポストのマーキングは先取りスケジュールされる.

こうして, すべてのプロセス, スレッド, GC ポストの投函物, システム変数のマークが終了したところで回収モードに移るわけである.

回収モードでは, GC はセル領域を低い番地から順に走査し, マークのついていないセルをもともとあった自由セルチャンクとともに回収していく. 一般に回収モードが始まる時点では, TAO プロセスの使う自由セルは回収走査前線より高い番地 (図で書くと下) である. これらの自由セルが尽きたとき, それまでに回収された低い番地の自由セルのほうに freec や freeccZ が切り替わるわけである. なお, TAO プロセスが回収走査前線に追いついてしまった場合は, 回収プロセスが先取りスケジュールされる.

こうしてセル領域を回収走査し尽せば, 消費モードに戻る. もし, この間にメモリ残量不足があれば, ただちにマーキングモードに移る. また回収中にメモリが払底した場合には, 適当な制限付きでセル領域を拡大する [2]. ただし, 無制限な拡大はしない.

GC はマーキングにしろ回収にしろ, 100 μ 秒連続走行する前に, 必ず hap 条件のチェックをするようにステップ数管理を行なう. このアルゴリズムが (適正負荷のもとで) 100 μ 秒の応答遅延を守って実現できるかどうかの解析やシミュレーションについては別の機会に報告する. これまでの検討では, 回収モードで起こり得るいろいろな事態への対処がアルゴリズムの詳細化の上で一番面倒なものであることがわかっている.

5.2 実時間 GC に合わせた基本演算の変更

前節で実時間 GC の流れの概略を述べたが, このアルゴリズムに対応して, TAO の基本演算をどのように変更したかをここで述べる. 一般に実時間 GC を導入すれば, car, cdr, cons, rplaca, rplacd などの基本演算のいくつかにオーバヘッドがもたらされるのが常である. どれにオーバヘッドを押しつけ, どれにオーバヘッドを課さないかは実時間 GC アルゴリズムをもつ記号処理システムの性能に大きく影響する. たとえば, car や cdr のように圧倒的に頻繁に使われる基本演算にオーバヘッドを課すことは好ましくない.

TAO/SILENT の GC は car と cdr にはまったくオーバヘッドを課さない. cons には次のような微小なオーバヘッ

2) 実際には新しいセル領域のブロックをシステムが予備として保持しているところから切り出してくる — 記述の簡略化のためここは厳密な記述ができない.

ドがある．回収中，回収走査前線より高い番地で cons が行なわれたとき，そのセルの GC マークビットが立てられる．これはあとから追いついてきた回収走査によって誤って回収されないようにするためである．このため cons のマイクロコードは回収モードのときのみ，1 または 2 ステップ (30 または 60 ナノ秒) の増加となる．ちなみに，Lisp 関数として呼ばれた cons[3] の典型的実行時間は 210 ナノ秒である．

しかし，セルの car や cdr を書き換える基本演算 rplaca, rplacd[4] は GC モードに依存した比較的大きいオーバヘッドがある．マイクロコードでは，セルなどの car や cdr を書き換えるとき，基本的にはすべて wcad ルーティンと呼ばれるマイクロサブルーティンを呼ぶようになっており，オーバヘッド部分を一個所にまとめてある．なお，GC のモードに関係なく直接 write 命令で書き込んでいいような場合がリストアップされていて，究極の効率が望まれるところでは wcad ルーティンを使わなくてすむようになっている．実際，プロセス制御の基本部分のマイクロコードはほとんど wcad ルーティンを使わないようになっている．

我々の実時間 GC で，なぜ rplaca や rplacd が問題になるかは，たとえば図 4 を見ていただければわかるであろう．ここで，マーク済みのセル A を rplaca した場合，新たに car につながれたセル C が最後までマークされない可能性があるのである (C を保持していた変数が消滅し，GC が A を経由してのみ C にアクセス可能なとき)．なお，セルのマークビットはセルの car 側にあることに注意．以下に，rplaca と rplacd の各モードにおける動作を簡単なプログラムで記述する．いくつかの関数の意味は自明であろう．書き込まれるセルのアドレスと書き込まれるデータをそれぞれ addr と data と書く．

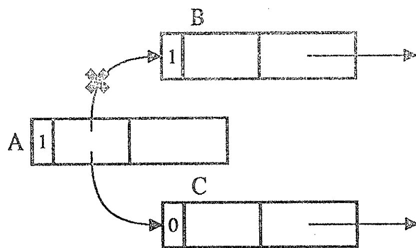

図 4. rplaca

◎ 消費モード: なにも考えずに書いてよい．

```
(rplaca addr data) = (write-car-unmarked addr data)
(rplacd addr data) = (write-cdr addr data)
```

---

3) ユニフィケーションで呼ばれる cons もある．

4) TAO はこれらを関数としては導入しない．

5) GC ポスト用のマーキングを独立プロセスにすることも考えられるが，これは未定．

cdr のタグの MSB は特殊用途に使われ通常はつねに 0 であるので，ことさらに write-cdr-unmarked とは書かない．

◎ マーキングモード: 書き込むセル，つまり addr がマークされているときが問題．黙って書き込むと data がマークされない恐れがある (なぜなら，GC はもうそのセルのマーキングは終わっていると考えているので)．つまり，この場合は data をマークする必要がある．しかし，ここで data に対してマーキングを始めるわけにはいかない (TAO プロセスが GC をやってしまうことになる!)．そこで次のような方策を立てる．

一定時間内にマーキングが終わることがわかっているデータ型に対しては「浅いマーキング」を行なう．即値データのようにマークをつける必要のないもの，またすでにマーク済みの data に対してなにも行なわない．浅いマーキングを行なうもの (easily-markable) は，たとえば，倍長整数，倍精度浮動小数点数，ビグナム，有理数，複素数，文字列はほとんどものが高々 2 回のメモリ I/O でマーキングが終了する (複素数は最大 7 回)．

セル，ベクタ，シンボルのようにマーキングが面倒でマーク未了のものは，GC ポストと呼ばれるシステムテーブルに data を投函する．これによって，GC プロセスに data をマークするように依頼するわけである．マーキングモードの GC は本業のマーキングの合間に GC ポストをときどきチェックする[5]．なお，GC ポストに何度も同じものを投函しないように，GC ポストに投函するものにはマークビットだけ立てておく．

```
(rplaca addr data) =
   (cond ((marked? addr)
           (shallow-mark data)
           (write-car-marked addr data) )
          (t (write-car-unmarked addr data) ))

(rplacd addr data) =
   (cond ((marked? addr)
           (shallow-mark data)
           (write-cdr addr data) )
          (t (write-cdr addr data)) )

(shallow-mark data) =
   (cond ((no-need-to-mark? data) (nop))
         ((marked? data) (nop))
         ((easily-markable? data) (mark data))
         (t (make-mark-bit-on data)
            (post-to-GC-post data) ))
```

◎ 回収モード: addr がマークされていなければ，なにも特別なことはせず data をそのまま書く (SILENT では，Lisp ポインタをオペランドとしてアクセスすると，タグの MSB がデフォールトの TAGALU 演算ではクリアされるので，data を表わす Lisp ポインタのタグの MSB は内部的には

いつもクリアされていると思ってよい). マークされているときは, GC がまだここを走査していないということであるから, GC マークを立てたままにしておくことが必要である. rplacd は消費モードとまったく同じである.

```
(rplaca addr data) =
  (cond ((marked? addr)
         (write-car-marked addr data) )
        (t (write-car-unmarked addr data) ))
(rplacd addr data) = (write-cdr addr data)
```

以上の説明からわかるように, たとえ GC 中でも rplaca と rplacd 自身にもたらされたオーバヘッドはそれほど大きくない. このアルゴリズムは湯浅のアルゴリズム[15]と見かけは似ているが中身は異なる. たとえば, 我々のアルゴリズムは書き込む data のほうをマークするのに対して, 湯浅アルゴリズムは上書きされるほうのデータをマークするところが決定的に異なる. 湯浅アルゴリズムでは GC が始まったときに生きていたセルと GC 中に cons されたすべてのセルを生きたセルとするのに対し, 我々のアルゴリズムは GC が始まったとき生きていたセルでも, GC 中に cons されたセルでも, マーキングプロセスが TAO プロセスと「鬼ごっこ」をやっているうちにゴミになったセルを回収してしまう可能性がある. その分, 回収効率がいいともいえる.

このアルゴリズムは, SILENT ハードウェアの特質をいくつか使っているが, それらは究極の速度達成のためのものであって, アルゴリズムにとって本質的というものではない. マルチプロセス管理をユーザが高速に行なえるのであれば, このアルゴリズムは汎用マシンでも実装可能だと思われる. なお, 我々の実時間 GC アルゴリズムのルーツは, ハードウェア試作も伴って検証された日比野の並列 GC アルゴリズム[16]である.

5.3 実時間 GC の側面支援

実時間 GC の効果を上げるには, ゴミを出にくくすることも重要な側面支援である. しかし, コンスセルは完全にユーザに開放されているので, ここでゴミを出にくくするというのは言語自身を変更することになってしまう. TAO の側面支援は文字列本体のメモリ管理と, システムデータ構造の中で多用されるバディメモリ管理である. ここではこれについて簡単に触れる.

TAO の文字列は DAO とまったく異なる. DAO の文字列は 1 セル相当のヘッダと, そこから指される文字列本体からなっていた. 文字列本体は 1 バイトコードと 2 バイトコードを最密充填したものであった. 文字列に対する基本演算では, 同じ文字列本体を指すヘッダが作られることが多く, 文字列本体の各部分は多数のヘッダから共有されていた. このため, 文字列本体の GC マーキングは 1 文字 1 文字行なわれ, 生きている文字だけがコンパクションされたわけである.

TAO の文字列では 1 バイトコードと 2 バイトコードの最密充填を止め, 混在したときは 2 バイト系の配列の中に 1 バイトコードを埋め込む (どちらかといえば今日ではふつうの) 方式をとる. そして, その文字列を直接指し示すヘッダはただ 1 個だけとする. 基本演算によってその文字列の部分列をとったときは, substring (部分列) というデータ型の 1 セル相当のヘッダが作られる. substring には親文字列へのポインタと, 部分列の範囲を示す記述子が書かれ, 文字列本体へのポインタはない. つまり, 一つの親文字列の部分列はすべて親文字列のヘッダを経由した間接表現となるわけである. このようにするメリットは, 親文字列の長さが増えて, 別の場所に文字列本体をとらないといけなくなっても, 親文字列からの本体へのポインタを書き換えるだけでよいことである. そして, このとき古い文字列本体は基本演算の中で陽に回収してよくなる (図 5).

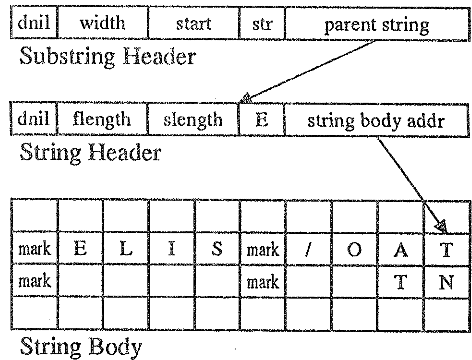

図 5. TAO の文字列

DAO で我々が作った最大の文字列処理プログラム Zen (Emacs のフルセット)[17] は, TAO の文字列の流儀で作ったほうがむしろ効率のよいことが予想されている. ただし, ハード実時間システムにとって文字列処理がどれほど意味があるかは別問題である.

上の文字列本体がそうであったように, TAO ユーザがデータ型として扱えないような内部データ型は, なるべくシステムが陽にゴミを回収できるようにするのが TAO/SILENT の設計思想である. このような目的に合ったメモリ管理としてバディシステムがある. バディシステムそのものはそれほどメモリ使用効率が高くないともいわれており, フィボナッチ型のバディなどいくつかの工夫もなされているが, 我々は一番単純な 2 のベキサイズのバディを採用した. 理由は単純で, 2 のベキサイズにすると TAO/SILENT の領域割り付けにおいて個々のバディブロックがキャッシュメモリのエントリ境界にピッタリ収

まることである．システムで最も多用されるバディブロックのサイズは4セルとか8セルなので，このデータにアクセスするときのメモリアクセスの挙動が予測できるのである(たとえば，先頭のセルにアクセスすれば，あとのセルにはすべてキャッシュミスなしでアクセスできる).

バディは高速連想表(symtab)，ビグナム，(データ型としての) FIFO キューや LIFO スタックの内部表現などのほか，内部的なプロセス管理などに利用される．

6. 終わりに

現在，TAO は完成した言語というよりも，記号処理と実時間処理のための効率のよい機械語を目指すというスタンスで設計の見直しを行なっている．機械語としての完成度が高ければ，S式という強力なプログラミング言語生成能力によって自然に「使える言語」としての実力が備わってくると考えるからである．

本稿では TAO/SILENT の実時間性について，構想中のものまで含めて概観した．ここで応答遅延時間の目標値100 μ秒を掲げているが，この数値を目標にしたことにより，TAO の内部構造を大きく変更せざるを得ないところが多々あった(当初は1ミリ秒を想定していた)．たとえば，メモリ管理は当初の版を廃棄し，全面的に仕様変更せざるを得なかった．また，プロセス管理も大幅な改良が必要である．とにかく，100 μ秒というのは高級言語マシンとしてはノンビリやっていられない値なのである[6]．しかし，この程度の実時間性が実証できなくては，記号処理が現場的 AI で安心して使えるという状況にはならないであろう．

[文献]

[1] 吉田雅治，竹内郁雄，天海良治，山崎憲一：新しい記号処理カーネル SILENT の設計，記号処理研究会，56-1, 1990.

[2] 竹内郁雄，天海良治，山崎憲一：新しい TAO の設計，記号処理研究会，56-2, 1990.

[3] 天海良治，山崎憲一，竹内郁雄：新 TAO のメッセージ伝達式，WOOC-91，ソフトウェア科学会，1991.

[4] 天海良治，竹内郁雄，吉田雅治，山崎憲一：TAO/SILENT のソフトウェア・アーキテクチャ，第8回ソフトウェア科学会大会，1991.

[5] 竹内郁雄，吉田雅治，天海良治，山崎憲一：新しい記号処理カーネル TAO/SILENT の設計，第43回情報処理学会全国大会，1991.

[6] 山崎憲一，竹内郁雄，吉田雅治，天海良治：TAO/SILENT における論理型プログラミング，第43回情報処理学会全国大会，1991.

[7] Y. Hibino, K. Watanabe, and I. Takeuchi: A 32-bit Lisp Processor for the AI Workstation ELIS with a Multiple Programming Paradigm Language TAO, Journal of Information Processing, 13-2, 1990.

[8] I. Takeuchi, H.G. Okuno, and N. Ohsato: A List Processing Language TAO with Multiple Programming Paradigms, New Generation Computing, 4-4, 1986.

[9] J.A. Stankovic and K. Ramamritham: Hard Real-Time Systems, IEEE, 1988.

[10] R.A. Brooks: A Robust Layered Control System For A Mobile Robot, IEEE Journal of Robotics and Automation, RA-2, No.1, 1986.

[11] M. Yoshida, T. Naruse, and T. Takahashi: A Dedicated Graphics Processor SIGHT-2, Advances in Computer Graphics Hardware IV, Springer-Verlag, 1991.

[12] I. Takeuchi: Concurrent programming in TAO — Practice and Experience, US-Japan Workshop on Parallel Lisp, Language and Systems, LNCS No.441, Springer, 1990.

[13] 村上健一郎：オブジェクト指向による TCP/IP プロトコルの実現，コンピュータソフトウェア，6-1，ソフトウェア科学会，1989.

[14] 高田敏弘：オブジェクト指向による X Window System インタフェースの実現，WOOC-89，ソフトウェア科学会，1989.

[15] 湯浅太一：汎用計算機に適した実時間ゴミ集め，記号処理研究会，41-4, 1987.

[16] Y. Hibino: A practical parallel garbage collection algorithm and its implementation, Conf., Proc. 7th Annual Symposium on Computer Architecture, 1980.

[17] 天海良治：オブジェクト指向による画面エディタの部品化，第28回プログラミングシンポジウム予稿集，情報処理学会，1987.

6) 現在の RISC でのコンベンショナルな実時間処理ではこの1桁下が最良値になっているが，AI 的処理とはほど遠いものである．