要求駆動型XML処理のスケジューリングおよびメモリに関する効率化

山中 真和[†,☆] 鎌田 十三郎[††]

本論文は，Producer/Consumer スタイルにより要求駆動を実現した XML 処理系 Nanafusi について，その効率化手法を述べる．Nanafusi は，遠隔 XML データ処理の効率化を目指した Java 上の計算環境であり，要求駆動に処理を行うことで全データの到着を待たずに処理を開始できる．また，不要メモリを順次解放することでメモリ使用量の削減も可能である．本研究では，スレッドとメモリの両面から，実行時オーバヘッドのさらなる削減を目指す．まず，データ転送速度に応じたスレッドスケジューリング機構を導入することで，反応速度を維持しつつ，スレッド切替えのオーバヘッドを削減する．一方，メモリ面では参照管理を改善することで，世代別 GC 上での効率的なメモリ資源の回収を達成している．加えて，実行途上で不要となった Producer スレッドの回収も実現する．本機構の性能評価は，複数のデータ転送速度で測定され，Producer/Consumer 同期を用いた本処理系のオーバヘッドは，イベント駆動で実現された場合に比べて 2〜4%程度まで抑制することができた．本実装技法は，遠隔データを対象とした Producer/Consumer 実装に対して効率化を行っている．

Adaptive Scheduling Mechanism and Memory Reduction Techniques for Efficient Demand-driven XML Processing

Masakazu Yamanaka[†,☆] and Tomio Kamada[††]

This paper presents runtime techniques to improve efficiency of a demand-driven XML processing environment, Nanafusi, which uses Producer/Consumer thread coordination for implementation. Using demand-driven processing, Nanafusi can start processing of a remote XML document before the arrival of the whole data, with discarding tree nodes that are already accessed and will be never used. In this paper, we improve our runtime system in thread scheduling and memory management, and reduce runtime overheads. First technique is an adaptive scheduling mechanism that observes data transfer speed and controls the frequency of thread context switches, to reduce scheduling overheads with returning initial responses quickly. For memory overheads, we reform reference management for Producer/Consumer coordination to utilize effectiveness of generational GC, and also present a way to terminate dormant producer threads that will not become active. Using these optimize techniques for Producer/Consumer coordination, Nanafusi shows only 2–4% runtime overheads against event-driven programs for various data transfer speed. These techniques will be useful for programs that incrementally process network data using Producer/Consumer coordination.

1. はじめに

近年，ネットワーク上に規格化された XML データが公開される機会が増えている．これらのデータをネットワーク越しに取得・処理する場合，全データ受信を待ってから計算開始したのでは，データ受信待ち時間の影響で反応時間が遅くなる可能性がある．いち早く結果を返し始めるためには，全データ到着を待たずに処理を開始することが重要である．

そこで我々は，要求駆動に処理を行うことで遠隔 XML データ処理の効率化を目指した Java 上の計算環境 Nanafusi を提案・実装している[1)〜3)]．実現にあたっては Producer/Consumer スタイルのマルチスレッド実装を用いている．構文解析や各種合成演算を要求駆動で行うことで，データの完全な到着を待たずに結果を出力可能である．また，不要となったメモリを順次解放することで必要メモリ量の削減も図れる．実験では，低速ネットワークで問題となるネットワーク遅延の隠蔽に成功し，結果出力開始までの反応速度

† 神戸大学大学院自然科学研究科情報知能工学専攻
Graduate School of Science and Technology, Kobe University
†† 神戸大学工学部情報知能工学科
Department of Computer and Systems Engineering, Faculty of Engineering, Kobe University
☆ 現在，日本電信電話株式会社
Presently with NTT Corporation

（以下，単に反応速度）を向上させただけでなく，しばしば全実行時間の短縮にも成功している．一方，本研究でさらなる性能評価を行った結果，高いデータ転送速度では遅延隠蔽効果が低くなり，イベント駆動に処理を行った場合に比べて，2 割程度の実行時オーバヘッドを引き起こしていたことが分かった（2 章）．

本研究の目的は，様々なデータ転送速度に対して，本処理系が反応速度を確保しつつ，実行時オーバヘッドを抑えられるようにすることである．このため，スレッド管理とメモリ管理の両面から，実行時システムの性能向上を目指す．

スレッド管理のポイントは，ネットワーク速度に応じてスレッド切替え頻度を調整することにある．Producer/Consumer 同期を行っている場合，一般にスレッド切替えが頻繁に行われると反応速度は向上するが，オーバヘッドは大きくなる．このため，データ転送速度が速いときは切替え頻度を抑え，遅い場合はスレッド切替えを重視するといった適応的戦略が重要と考える．本研究では，一般の Java 処理系上で適応的スケジューリングを実現するための実装技術を紹介する（3 章）．

一方，メモリ管理のポイントは，Producer/Consumer 関係の中で，今後利用されないと分かった部位を効率的に回収する点にある．Nanafusi では，プログラマの判断によって，今後アクセスしない部分木を親木から切り離し，Garbage Collection（GC）によって回収している．ただし，Consumer が複数存在するような Producer/Consumer 実装は，現在広く利用されている世代別 GC と相性が悪い．また，計算途中で Consumer がいなくなった場合，Producer スレッドも適切に回収する必要がある．本研究では，一般の Java 処理系上で，不要になった資源を効率的に回収するための実装技術を紹介する（4 章）．

2. Nanafusi 概要

2.1 機能および記述イメージ

Nanafusi では，XML データを Tree として扱うことで記述性を維持する一方，要求駆動により反応速度の向上や必要メモリ量の削減を目指している．システムは，入力 XML ストリームを XTree と呼ばれるデータ構造にマッピングするデータバインディング部と，XTree に関する合成演算ライブラリから構成される．また，システム側でスケジューリング変更ができるように，XTree への更新操作を許していない．

プログラマは，XML ストリームから切り出したい要素をスキーマ上で指定することで，データを**図 1** のような XTree 構造に変換・要求駆動に基づいたアクセスが可能である．この例では，XML データ全体が `HouseInfo` クラスにマッピングされ，各 `House` 要素へは `XList⟨House⟩` を介して順次アクセス可能である（逆順アクセス不可）．また XTree のアクセサは 2 種類あり，通常の `get` 系アクセサの他に，メモリ使用量削減を意図して `remove` 系アクセサを提供している．`XList` については `removeIterator()` が準備される．各部分木の取得が今後行われないと分かっている場合は，プログラマは `remove` 系アクセサにより各部分木を親木から切り離し，GC の対象にできる．切り離し後のアクセスでは実行時例外が発生する．

```
/** Nanafusi 標準クラス */
interface XTree { /* marker interface */ }
abstract class XList<T> {
    abstract XIterator<T> iterator();
    abstract XIterator<T> removeIterator(); }
interface XIterator<T> {
    T next();
    boolean hasNext();
}
/** 型クラス */
abstract class HouseInfo implements XTree {
    abstract XList<House> getHouses();
    abstract XList<House> removeHouses(); }
abstract class House implements XTree {
    abstract Integer getPrice();
    abstract Integer removePrice();
    abstract String  getAddr();
    abstract String  removeAddr();
}
/** 実装クラス */
class HouseInfoImpl extends HouseInfo { .. }
class HouseImpl extends House {  ...  }
```

図 1 XTree クラス（例）
Fig. 1 `XTree` class (sample code).

```
/** テンプレート利用例 */
class JoinImpl extends Join<House,Shop,JHouse>{
  JoinImpl(XIterator<House> in0,XIterator<Shop> in1){
    super(in0,in1);  }
  JHouse constElem(House in0,Shop in1){
    return new JHouseImpl(in0,in1);  }
  boolean condition(House in0,Shop in1){
    return in0.getAddr().startsWith(in1.getAddr()); }
}
/** 利用例 (main 文などに記述) */
XList<House>  h = new HouseInfoImpl(H_URL).getHouses();
XList<Shop>   s = new ShopInfoImpl(S_URL).getShops();
XList<JHouse> j = new JoinImpl(h.removeIterator(),
                               s.removeIterator());
```

図 2 集合演算テンプレートとその利用例
Fig. 2 XList operation (sample case).

XTree の合成演算に関しては，データベース的な集合演算や ListMap などの演算を合成演算ライブラリとして提供している．**図 2** は `House` 型と `Shop` 型の各 `XIterator` を入力としてうけ，各 `House` 要素を住所が近い Shop 要素と結合し，`JHouse` 型の要素リストを生成する例である．ライブラリクラス（Join）を継承し，結合条件と結果要素の生成法をメソッド記述

するだけで利用可能である．これらのライブラリは，2 入力を並行して取り込むことで，早い時点から最初の解を出力できるようにデザイン/実装されている．

これらの機能を用いることで，反応速度を向上させるために従来必要であった，イベント駆動での処理やプログラムのマルチスレッド化，スレッド間の同期などの実装作業からプログラマは解放される（詳細は文献 2) を参照）．

2.2 基本的な実装技術

現在の Nanafusi では，XTree の遅延構築の実現には，マルチスレッドを利用した Producer/Consumer スタイルの実装を行っている[1),2)]．データバインディング部では，イベント駆動型の XML パーザが Producer スレッドとして実行され，アクセサと協調動作する．一方，`Join` などの複数入力に対する合成演算においても，各入力データのアクセス（にともなう入力データ生成）用に別スレッドを準備し，ネットワーク遅延に起因して計算がブロックされることを避けている．

一方で，頻繁なスレッド切替えによるオーバヘッドを避けるために，Producer は結果をバッファリングしている．Consumer からアクセス要求があった時点で Producer は再開され，バッファが充足した段階で `wait()` を行う．このため，バッファが大きいとスレッド切替えは減るが，反応速度も遅くなりがちである．一方，バッファが小さいとスレッド切替え増加により，オーバヘッドが大きくなる．このように，反応速度向上とオーバヘッドの削減はトレードオフの関係にある．

また，本処理系ではメモリ使用量削減にむけたデータ構造のデザイン・実装を行っているが，この説明については 4.1 節に譲る．

2.3 旧実装の性能分析

本節では，文献 2) で利用した処理系に対して，データ転送速度やバッファサイズを変化させた場合，反応速度や全実行時間がどのように変化するか観察する．

評価環境は，Pentium4 Xeon 2.4 GHz，1 Gbyte memory，Red Hat Linux 8.0 uniprocessor 仕様を用い，また，Java VM は SUN jdk1.5.0，構文解析部は RelaxNGCC ver1.1 を利用している．データ供給は 1 Gbit Ethernet で接続された別計算機から行い，転送速度を変更するために，4 Kbyte ごとに一定時間データ送信を停止させるプログラムを作成した．

最初の解が出るまでの反応時間（first）と，全実行時間（total）の測定方法は，クラスロードの時間やデータ転送速度の揺らぎの影響を抑えるため，評価プログラムを連続して 13 回実行し，最初の 3 回の値を除いた 10 回の平均値をとって測定した．

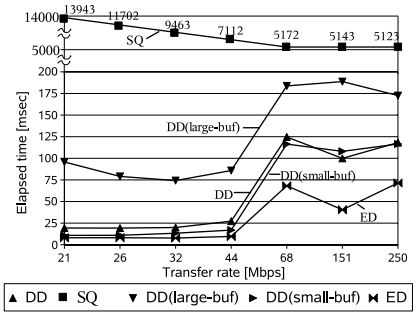

図 3 実験結果 1（反応時間）
Fig. 3 Performance results 1 (response time).

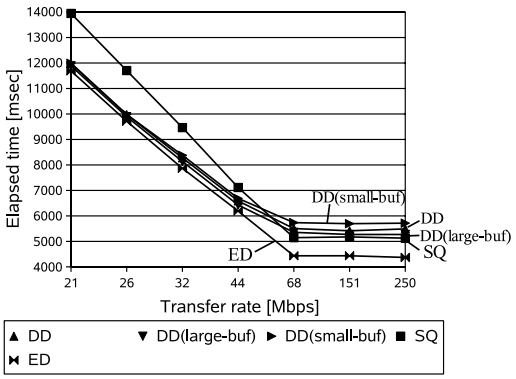

図 4 実験結果 1（全実行時間）
Fig. 4 Performance results 1 (total time).

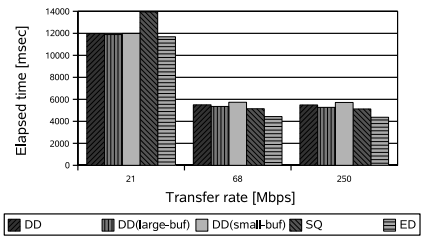

図 5 実験結果 1（全実行時間の一部）
Fig. 5 Performance results 1 (total time).

測定プログラムとして，以下の 3 種類を準備した．

- **SQ**：データを完全に木構造に展開し，すべての計算を行って次の段階の処理を実行する（本システムを改変して実装）．
- **ED**：全処理をイベント駆動型スタイルで実装（構文解析部は RelaxNGCC を利用）．反応速度向上や遅延隠蔽のために，マルチスレッドを用いて，2 入力を並行して読み進めている．
- **DD**：Nanafusi 利用プログラム．バッファサイズを変えた $DD_{[small\text{-}buf,\ large\text{-}buf]}$ を準備．

図 3，**図 4**，**図 5** は，XMLData Repository[4)] 上

表 1 入力データ
Table 1 Input data.

	File Name	lineitem	orders
サイズ（Mbyte）		30.6	5.1
要素サイズ	平均（byte）	534	357
	範囲（byte）	504〜560	321〜394
	標準偏差	11	18

にある TPC-H Benchmark [5] のデータを利用して，2 つの入力データ（**表 1**）に対し，ハッシュを利用した結合演算（HashJoin）を行った結果である．

まず，反応速度について見ていく．今回は，特にスケジューリングの反応時間への影響を計測するため，ソート済みデータを利用しており，早い時点で最初の解を求めることができる．そのため，DD の反応速度はネットワーク転送速度にかかわらず SQ より大幅に速く，200 msec 以内でレスポンスを返している．また，$DD_{[large\text{-}buf]}$ は全般的に ED や $DD_{[small\text{-}buf]}$ より反応速度が遅いが，一方で，転送速度が高速になるにつれて，ED や $DD_{[large\text{-}buf]}$ も逆に反応速度が低下するという症状が起きている．これは，入力スレッド間のスケジューリング調整を行っていないため，データ転送速度が高速な場合，一方のデータを集中して読み進める場合があるのが原因である．

次に，全実行時間について見ていくと，データ転送速度が 44 Mbps 以下の場合，SQ より 6〜14%程度短縮することに成功し，ED に対しても 2〜8%のオーバヘッドに抑えられている．これは，データ待ち遅延隠蔽効果により，要求駆動に要するオーバヘッドが吸収されたためであり，バッファサイズによらず同程度の値を示している．一方，データ転送速度が 68 Mbps より高速になると，CPU 負荷が大きくなりネットワーク待ち時間がほとんどなくなるため，DD ではオーバヘッドが表面化してくる．たとえば，250 Mbps では，DD は ED に対して約 25%のオーバヘッドが出ている．ただし，$DD_{[large\text{-}buf]}$ では，スイッチ回数を削減することで，オーバヘッドのうち，約 2 割を削減できる．しかし，メモリオーバヘッドなどもあり（詳細は，4 章）約 20%のオーバヘッドが残る．

3. スケジューリング管理

3.1 基本方針

3 章では，データ転送速度に応じたスレッドスケジューリングの自動化による反応速度向上と実行時間の抑制の両立手法について述べる．2.3 節での評価をふまえると，Producer/Consumer スタイルの実装では，

- データ転送速度が低いときは，反応速度向上のために，スレッド切替えを優先すべき．ネットワーク待ち時間が存在するため，少々のオーバヘッドは隠蔽可能，
- データ転送速度が高いと，オーバヘッドが表面化するため，全実行時間の短縮には，スレッド切替えの抑制が重要．また，反応時間の向上には，公平なスケジューリングが重要，

となる．ただし，対象がネットワークであるため，

- 転送速度の判別や，転送速度の変化への対応

も重要である．

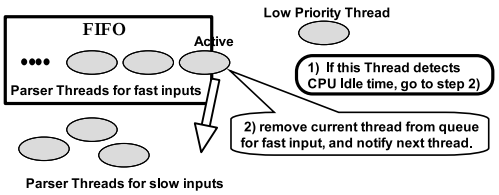

図 6 スケジューリング管理のイメージ
Fig. 6 Adaptive scheduling mechanism (overview).

上記のスケジューリングを一般の Java 処理系上で実現するため，我々は，以下の方針をとっている．

- 高速入力データを扱うパーザスレッドは，FIFO キューに基づく明示的スレッド管理により，公平なスケジューリングを行う．
- 低速入力データを扱うパーザスレッドは，いち早くデータ受信に反応させるため，スレッドスケジューリングに制約を課さない．
- パーザスレッドは，バッファを充足させるのに要する時間を定期的に測定し，バッファサイズをそのつど，データ転送速度に応じて再設定する．
- 合成演算に関しては，入力データ側のバッファサイズに応じたスケジューリングを行う．加えて，スレッド切替えを抑えるための実装技術を施す．

以下の節では，パーザスレッド管理と合成演算に関する実装技術を述べたあと，その評価を行う．

3.2 パーザスレッドのスケジューリング管理

高速パーザ管理部は，実行要求を受けたパーザスレッドを FIFO キューで管理し，同時に 1 つしかスレッドが実行されないようにスケジューリング管理を行っている．加えて，スレッドがデータ到着待ちを起こしているのを検出し，低速分類に切り替える機能を有する．パーザスレッドは，その開始時点では高速パーザ管理部に割り当てられ，順番に実行されるが，データ待ちで処理がブロックしたものは，そのつど管理から外される（**図 6**）．

入力データ待ちの検出は，優先度を低く設定したスレッドを準備し，CPU のアイドル時間を検出した場

```
/* CPU の idling 時間検出用 Thread の run メソッド */
public void run(){
  while(true){
    long before = System.currentTimeMillis();
    call_yield(); /* Thread#yield() を複数回呼び出す */
    long time = System.currentTimeMillis() - before;
    if(time < SOME_RANGE )
     hasCPUIdlingTime(); /* CPU の idling 時間を検出 */
    ....
  }
}
```

図 7 CPU アイドル時間検出用プログラム（一部）
Fig. 7 Program to detect CPU idle time.

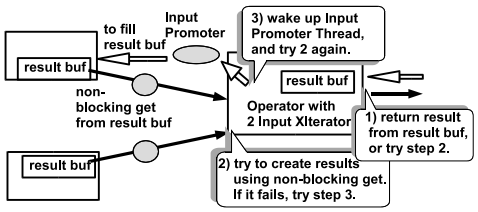

図 8 合成演算の挙動
Fig. 8 Internal behavior of XList operation.

合に，データ待ちを生じる遅い入力と判定している．遅い入力用と判断されたパーザスレッドに対しては，オーバヘッドを許容してでも，スレッド切替え優先のスケジューリングを行う．アイドル検出用のスレッドは，Java のスレッド優先度機構が実装されていない場合に備えて☆，**図 7** のような機構を準備している．アイドルを検出すると，高速パーザ管理部は，現在実行中パーザを低速分類に変え，次のパーザスレッドの再開を指示する．

一方，低速分類されたパーザスレッドは，自由に実行しており，加えて，転送が高速化した場合に備え，バッファを充足させるのに要する時間の測定と，それに応じた高速分類への切替えを行っている．

パーザスレッドは，構文解析を再開してからバッファを充足させるまでの時間を定期的に測定することで，そのバッファサイズを決定している．高速の場合は 10 msec，低速では 5 msec パーザスレッドを進めるようにバッファサイズを設定することで，反応速度の確保（数十 msec）とスレッド切替えの抑制を行っている．低速時のバッファ設定が小さいのは，スレッド切替えを優先的に行うためである．このように時間計測に基づいたバッファサイズ設定を行うことで，XML データ構造の複雑さの差異による影響も吸収することができる．

3.3 合成演算部のスケジューリング管理

合成演算においてマルチスレッド実装を行っているのは，`Join` などの複数入力を扱うケースであり，一方の入力がブロックされた場合でも処理が進められるようにするためである．今回，このスレッド切替えのオーバヘッドを抑えるため，入力データ側のスケジューリングに適応した合成演算部のスケジューリングを行う．その基本方針は，以下のとおりである（**図 8**）．

- システム提供の `XList` ライブラリでは，non-blocking get を内部実装する．このメソッドは，他のスレッドを再開しない範囲で結果を返す．
- Consumer は生成済の結果がない場合，まず non-blocking get を用いて自力で計算を進める．結果が得られない場合は，各入力データ取得用のスレッドを再開し，入力データが溜まるのを待つ．
- 入力データ取得用スレッドは，Consumer として入力 `XList` 側の計算を進める．当初は non-blocking get を用いて計算を行い，指定要素数が生成できなかった場合には，入力 `XList` の入力取得用スレッドを再開し計算を進める．

つまり，基本的に入力データ側のバッファの範囲で計算を進めることで，入力データ速度に応じたスケジューリング調整を行っている．結果，外部データ速度に応じたスケジューリングが行われることになる．

上記は，Join などの 2 入力合成演算の場合であり，1 入力演算の場合は，そもそもマルチスレッド実装は行われない．同様に，2 入力演算の場合も，一方の入力取り込みが完了すれば，取り込みを行うべき入力集合は 1 つとなるため，マルチスレッド実装は利用されなくなる．つまり，Consumer 自ら入力 `XList` の計算を進めており，Consumer と入力データ取得用スレッド間での切替えが発生しなくなる．この実装技術は，一方の入力取り込みが早期に完了するような，たとえば，入力データサイズが不均等な場合に特に有効である．

3.4 評　　価

2.3 節と同様のアプリケーション/測定環境を利用して，スケジューリング管理機構の性能評価を行う．Java のオプション指定については，測定環境が単一プロセッサ環境であり，`-XX:+UseSpinning` オプションは指定していない．また，測定環境（Linux）では優先度機構が有効に機能しないため，`-XX:+UseThreadPriorities` オプションを指定せず，代わりに図 7 のプログラムを用いている．さらに，本

☆ Java のスレッド優先度機構は実装が保証されておらず，Solaris では，SUN jdk 1.4.2 以降で有効であったが，評価環境に用いた Linux では，現在の最新版である SUN jdk 1.5.0 上で `-XX:+UseThreadPriorities` 指定した場合でも有効に機能していないことを予備実験で確認した．予備実験は，最高/標準/最低優先度のスレッドを 3 種同時に作成・並行実行させ，その進行状況を測定したものである．

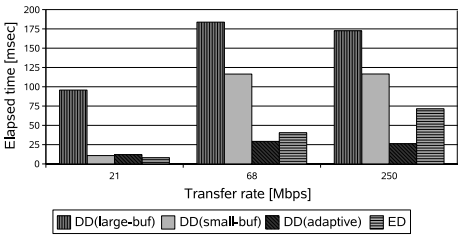

図 9 実験結果 2（反応時間）
Fig. 9 Performance results 2 (response time).

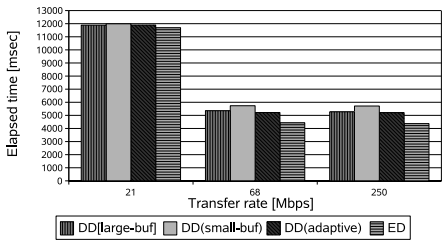

図 10 実験結果 2（全実行時間）
Fig. 10 Performance results 2 (total time).

測定では，`-server` オプションを利用していない．これは，本評価が主に反応速度の観測を目的としているためであり，`-server` 指定による速度変化を含めた全実行時間の評価は，4.4 節で行う．

比較対象は，本節のスケジューリング管理を適用した場合，$DD_{[adaptive]}$ の性能を旧版である $DD_{[small\text{-}buf,\ large\text{-}buf]}$ ならびに ED と比較する．**図 9**，**図 10** が評価結果を示しており，実験データや実験内容は 2.3 節と同様に lineitem.xml［約 30 Mbyte］と，orders.xml［約 5 Mbyte］の結合演算である．

まず，反応速度についてであるが，$DD_{[adaptive]}$ は，データ転送速度にあまり影響されず，良好な数字を示している．データ転送速度が低速な場合には，バッファを小さく設定することで，$DD_{[small\text{-}buf]}$ に近い値を示している．たとえば，21 Mbps の場合，レスポンスを 12 msec で返すことに成功している．一方，ネットワーク転送速度が高速化した場合においても，入力取り込みスレッドの FIFO 管理による公平なスケジューリングを行うことで，数十 msec 程度の反応時間を示している．たとえば，データ転送速度が 250 Mbps の場合，$DD_{[small\text{-}buf]}$ が 120 msec なのに対して，30 msec で最初の解を返している．

一方，全実行時間についても，$DD_{[adaptive]}$ では，改善を見せている．データ転送が低速な場合は，遅延隠蔽効果によりオーバヘッドが顕在化しないが，一方，データ転送速度が高速な場合も，バッファサイズを大きく設定することで，$DD_{[large\text{-}buf]}$ と同様の値を示すことができている．

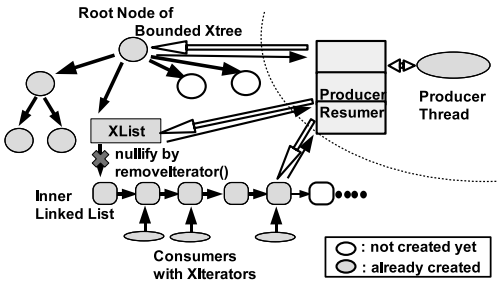

図 11 Producer/Consumer 実装
Fig. 11 Reference management for XTree binding.

4. 効率的なメモリ資源回収の実現

4.1 Producer/Consumer の実現

4 章では，Nanafusi が用いる Producer/Consumer 実装と現在広く使われている Java 処理系の間のメモリ管理上の問題と対応策を紹介する．議論に先立ち，データバインディング時の参照関係を示した**図 11** を用いて，本処理系で行っている参照管理の基本について説明する．

Nanafusi では，データバインディングや合成演算の際，XTree へのアクセス要求に応じて，Producer スレッドが遅延構築を行う．このため，各 XTree，XList の実装クラスは遅延構築を促すための Producer への参照を保持する．一方で，Producer 側は構築再開ポイントの情報を図 11 の Producer Resumer として保持する．

XTree は，遅延構築を行っていただけではメモリ使用量削減にはつながらない．プログラマが今後アクセスしないと判断した部分木を，`remove` 系アクセサによって参照取得と同時に親木から切り離すことで，GC 対象とできる．図 11 の `XList` の例をもとに，内部実装の紹介を行う．`XList` は逆順アクセスを許していないため，内部で一方向線形リストを作成し，`iterator()` の要求に応じてその先頭への参照を返却する．加えて，`removeIterator()` の場合は `XList` から先頭要素への参照削除も行う．この例では，3 個の `XIterator` が取得され，それぞれ線形リストを読み進めている．`removeIterator()` を利用した場合は，線形リスト先頭への参照が失われるため，すべての `XIterator` が通過した要素から順次ゴミとして扱われることになる．

4.2 世代別 GC との問題と対応

前節で述べたような Producer/Consumer 実装を行った場合，オブジェクト間の参照関係は，自然に古いものから新しいものへの参照となる．また，XList

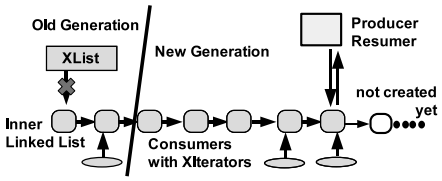

図 12 XList と世代別 GC の関係
Fig. 12 XList references with generational GC.

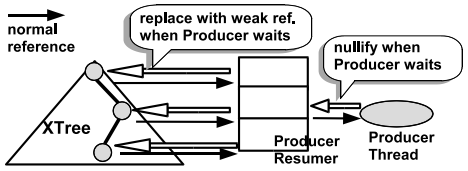

図 13 Producer スレッドの回収にむけて
Fig. 13 Internal behavior to terminate Producer thread.

の要素についても，古いものから先にゴミとなる．このため，メモリ割当ての速さや雑多なオブジェクトを早期に回収できるといった世代別 Copying GC の利点が発揮できないケースが起こる．

世代別 GC では，オブジェクトは新・旧 2 つの世代に分けられ，Full GC で全メモリ空間を GC の対象とするのに対し，新世代 GC では，新世代オブジェクトのみを GC の対象とする．その際，旧世代から参照される新世代オブジェクトも live オブジェクトとして扱われる．何度かの新世代 GC を生き残った場合，オブジェクトは旧世代として扱われるようになる．

このため，もし XTree，XList の要素が 1 つでも旧世代になると，そこから参照されるオブジェクト群は新世代 GC ではゴミにならない．`remove` 系アクセサによる部分木の切り離しも，切り離し対象がすでに旧世代であるなら，新世代 GC に対し有効ではない．深刻なのは要素数の大きな XList のケースである（**図 12**）．内部実装で利用される線形リストの一部でも旧世代になると，それ以降のリストは新世代 GC では回収できなくなってしまう．

世代別 GC を効果的に利用するための解決策として，利用し終わった線形リストをこまめに分断することにした．新世代にいる間に参照を分断できれば，それ以降のリストは新世代 GC の対象にできる．実装にあたっては，`XIterator` が 1 つしか取得されなかった場合と，複数取得されうる場合を区別している．

最初から `removeIterator()` が利用された場合，他の `XIterator` を気にする必要がないため，`next()` 取得のたびに参照を null 化している．一方，通常の `iterator()` が利用された場合は，自身が最後のアクセスであるかどうか確認を行いながら，線形リストの分断を行っている．この操作は他スレッドとの同期操作が必要であるため，コスト削減のため確認は 10 要素程度ごとに行われる．そのうえで，他の `XIterator` がすでに当該要素を読み終えている場合に，参照の分断を行っている．加えて，`XIterator` が読み進められることなく GC された場合に備え，`finalize()` メソッドを利用した処理も行っている．

4.3 Producer スレッドの GC

本処理系のように要求駆動で計算を行う場合，一部のデータしか必要でなかった場合，計算量自体を抑えることができる．ただし，本処理系のように Producer/Consumer スタイルの実装を行った場合，途中で停止している Producer スレッドを終了させる必要がある．加えて，Consumer が存在するかどうかは，参照関係からのみ判定されるため，GC 機構との連携が必要となる．

本処理系が Java 上の実装でとった基本方針は，

- Producer 側から XTree への参照は，その停止中は WeakReference のみによって行う，
- 各 XTree は Producer 再開用データ構造を介してスレッドを再開する，
- 再開用データ構造の `finalize()` 処理としてスレッドの終了処理を記述する，

というものである．Consumer 側からの参照がなくなった時点で XTree ならびに再開用データ構造は GC の対象となり，Producer スレッド自身も停止する．

ただし，Producer スレッドから XTree への参照を，常時 WeakReference で行うのは，WeakReference オブジェクトの生成を含めコストが大きい．このため，各 Producer スレッドは，停止する段階で参照を WeakReference 化し，再開時点で実参照に復帰している．また，Producer 自身も，停止中は再開用データ構造への参照を破棄し，再開時に再設定を行っている（**図 13**）．

4.4 評　　価

2.3 節と同様のアプリケーション/測定環境を利用して，本章のメモリ資源回収機能を評価する．全実行時間におけるメモリオーバヘッド削減効果を確認するため，オーバヘッドが表面化する高速転送時の結果のみ示す．そのため，データ転送速度を 250 Mbps とし，反応速度については省略する．また，現在広く利用されている世代別 GC を本評価で用いる．

まず，実験 3 で SQ，ED および前章のスケジューリング管理のみを適応した場合（$DD_{[A\text{-}M]}$）と，メモリ資源回収機構も含めてすべて実装した場合（$DD_{[A+M]}$）

表 2 Java VM のオプション指定一覧
Table 2 Java VM options.

各プログラムの実行環境	DD/SQ/ED	DD_{O1}	DD_{O2}	DD_{O3}
-Xmx	160 M	160 M	160 M	320 M
-XX:NewSize	640 K（default 値）	640 K	64 M	128 M
-XX:NewRatio	12（default 値）	6	1	1
-XX:SurvivorRatio	8（default 値）	1	8	8
-XX:TargetSurvivorRatio	50（default 値）	99	50	50
-XX:MaxTenuringThreshold	31（default 値）	63	31	31
その他のオプション	default 値（-Xms については本文参照）			

を比較し，本手法の効果を確認する．次に，実験 4 において，従来の DD と世代別 GC との相性の悪さをより詳細に分析するために，$DD_{[A\text{-}M,\ A+M]}$ に対し，Java のメモリパラメータ変更を施した場合の速度変化も観察する．

表 2 に，設定した Java オプションのパラメータ値を示す．実験 3 で用いる DD，ED，SQ では，`-Xmx` オプションによりヒープの最大サイズを 160 Mbyte に指定した以外，メモリ設定を変更しない．ヒープの最大サイズを変更した理由は，SQ の実行には最低 52 Mbyte のメモリを必要としており，標準値（ヒープ総量 64 Mbyte）では不十分なためである．対象環境では，各プログラムのヒープサイズは，自動メモリ管理機構により自動調整され，SQ で約 61 Mbyte に，ED，$DD_{[A\text{-}M,\ A+M]}$ では約 23 Mbyte に調整されていた．評価にあたっては，プログラムを複数回連続実行し，最初の 3 回の値を除外することで，ヒープサイズ調整までのオーバヘッドを除いている．一方で，実験 4 では，新世代領域のサイズや比率の影響を見るために，メモリパラメータ（$DD_{O1,\ O2,\ O3}$）を準備している（詳細後述）．

測定結果として，全実行時間に加えて，メモリ資源の回収状況を示すために，処理に最低限必要なメモリ量（mem）と GC に要した総時間を示す（**表 3**）．加えて，参考データとして，`-server` オプション指定を行った場合（3 章の $DD_{[large\text{-}buf]}$ 含む）の結果を**表 4** に示す．処理に必要なメモリ量は，頻繁に明示的な Full GC を起動するプログラムにより測定し，GC に要した時間は，`-verbose:gc` オプション指定することで，測定を行った．これとは別に，各プログラム実行中のヒープ使用量の時間的推移を測定するために，各 GC（世代別 GC 含む）後のヒープ使用量の推移（**図 14**）を `-verbose:gc` オプション指定により測定した．

実験 3 の結果：まず，処理に最低限必要なメモリ量については，DD では，SQ に対して 66%削減されている（表 3）．これは，`remove` アクセサ利用に加え，結

表 3 実験 3 測定結果
Table 3 Performance results 3.

実験結果 3	全実行時間（msec）	minor gc（msec）	full gc（msec）	mem（Mbyte）
ED	4,371	234	75	15.6
$DD_{[A+M]}$	4,476	266	121	17.0
$DD_{[A\text{-}M]}$	5,207	699	313	17.6
SQ	5,126	608	267	52.7

表 4 -server オプション指定時の測定結果
Table 4 Performance results 3 with -server option.

	全実行時間（msec）	gc（msec）
ED	3,077	303
$DD_{[large\text{-}buf]}$	4,089	1,131
$DD_{[A+M]}$	3,211	342
$DD_{[A\text{-}M]}$	3,946	936
SQ	3,648	781

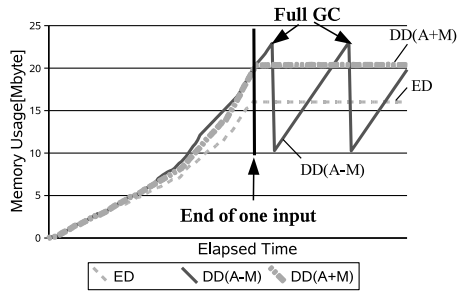

図 14 GC 後のヒープ使用量の時間的推移
Fig. 14 Memory usage after each GC.

合演算では，一方の入力をすべて読み込んだ時点で他方の入力集合への参照を破棄し，それ以降は保持しないようにしているからである（詳細は文献 2)）．しかし，ヒープ使用量の推移（図 14）を見ると，$DD_{[A\text{-}M]}$ では，一方の入力の取り込みを完了した後も新世代 GC だけではゴミを回収しきれず，ヒープ使用量が増加していく．このように，Full GC でしか回収できないゴミが大量に発生しており，世代別 GC が効果的に働いていない．

一方，$DD_{[A+M]}$ では，XList 内の線形リストを分断

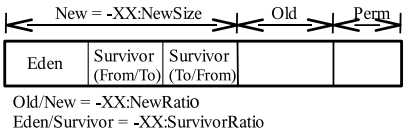

図 15 ヒープ構造（SUN HotSpot Client VM）
Fig. 15 Heap layout (SUN HotSpot Client VM).

することで，ED 同様の新世代 GC によるゴミ回収を可能にしている．このため，一方の入力が尽きた時点からは，`Join` 実装の効果もあり，ヒープ使用量の増加が抑えられている．一方の入力が尽きた以降における，新世代 GC 1 回あたりの回収率は $DD_{[A\text{-}M]}$：66.7%から $DD_{[A+M]}$：96.2%に向上した．このため Full GC を行う必要が薄れ，その実行頻度も平均 1.4 回から 0 回へと減少していた．結果として，GC 時間の総計については，$DD_{[A+M]}$ は $DD_{[A\text{-}M]}$ に対し 33%程度しか要していない．最終的に，本機構の導入により，全実行時間は，SQ に対して，約 13%短縮に成功し，ED と比べても 2.4%の増加に抑えられている．つまり，旧版が ED に対して持っていたオーバヘッドの 8 割程度を，GC の効率化によって削減できたといえる．

一方で，`-server` オプションを指定した場合（表 4），$DD_{[A+M]}$ の ED に対するオーバヘッドが 4.4%へと若干増加しており，これは，DD 内部で，同期処理を多用しているために，JIT コンパイラが ED ほど効果的に働かなかったためかと考える．

実験 4 の結果：実験 4 では，世代別 GC の各領域サイズを変化させた際，性能に与える影響を評価する．**図 15** は，SUN Java のヒープ構造に加え，Java オプションによって設定指示可能な領域サイズや比率を示している（詳細は文献 6)，7) 参照）．ヒープ構造中の新世代領域は，Eden と 2 つの Survivor 領域に分かれており，さらに Survivor 領域は From/To の 2 つに分けられる．最初，Eden に生成されたオブジェクトは，新世代 GC では Survivor に移され，Survivor 領域が溢れるか，最大で `-XX:MaxTenuringThreshold` 指定回数（標準値は 31）生き残った場合に，旧世代領域に移される（殿堂入り）．実験 4 では，Java のメモリパラメータを表 2 の $DD_{O1,\ O2,\ O3}$ のように設定し，その性能変化を測定した（**表 5**）．ただし，Java は各領域サイズを自動調整することがあり，安定状態に入ったときの領域サイズを測定したのが，表 5 下段である（実行によって揺れがあるため，代表的実行の結果を記載している）．

まず，標準設定の $DD_{[A\text{-}M]}$ について，その殿堂入り状況を調べてみると，ほとんどのオブジェクトは Survivor 領域を 1 度生き残っただけで旧世代へと移動

表 5 実験 4 測定結果
Table 5 Performance results 4.

実験結果 4	全実行時間 (msec)	minor gc (msec)	full gc (msec)
$DD_{[A+M]}$（再掲）	4,476	266	121
$DD_{[A\text{-}M]}$（再掲）	5,207	699	313
$DD_{O1[A+M]}$	5,348	879	86
$DD_{O2[A+M]}$	4,341	155	83
$DD_{O3[A+M]}$	4,141	112	0
$DD_{O1[A\text{-}M]}$	6,035	1,452	338
$DD_{O2[A\text{-}M]}$	4,769	648	185
$DD_{O3[A\text{-}M]}$	4,212	114	0

安定時のメモリ環境（目安）	Eden (Kbyte)	Survivor (Kbyte)
$DD_{[A+M]}$	1,280	128
$DD_{[A\text{-}M]}$	1,344	128
$DD_{O1[A+M]}$	1,216	1,216
$DD_{O1[A\text{-}M]}$	1,280	1,152
$DD_{O2[A+M/A\text{-}M]}$	52,480	6,528
$DD_{O3[A+M/A\text{-}M]}$	104,960	13,056

させられていた．これは Eden 領域に対する Survivor 領域の比率（標準値：1/8）が小さいため起こった症状といえる．今回のアプリケーションのように結合演算を行っている場合，一方の入力の取り込みを完了するまでは，取り込んだデータを保持する必要があり，ゴミの割合が低下する．結果，殿堂入りが早くなり，旧世代から新世代へのリンクも多発することになり，新世代 GC の効率低下が引き起こされる．

これをふまえて，早期の殿堂入りを抑えるべく，Eden 領域のサイズをそのままに，Survivor 領域を Eden 領域なみに拡大させた（表 5 下段）のが，DD_{O1} である．結果，旧世代への移動は，Survivor 領域を 4 回程度生き残った時点で行われていたが，逆に新世代 GC でのコピー操作が増加し，GC 時間増加を招いていた．実際，$DD_{O1[A\text{-}M,\ A+M]}$ では，新世代 GC に要する時間が $DD_{[A\text{-}M,\ A+M]}$ より 600 msec 以上増加している．このように，殿堂入りまでの期間を長くしようとすると，新世代 GC のコピー操作が増加し，速度低下を招く．

$DD_{O2,\ O3}$ は，単純に新世代領域全体を拡大することで，GC の頻度と殿堂入りを抑えたものである．たとえば DD_{O2} では，新世代領域のサイズ（合計）を，処理に必要な最低メモリ量（16.7 Mbyte）の 4 倍程度（標準の 100 倍）に設定することで，新世代 GC を平均 3 回に抑え，GC 時間を短縮している．しかし，$DD_{O2[A\text{-}M]}$ においても，$DD_{O2[A+M]}$ に対して 1 割程度のオーバヘッドが残っている．$DD_{O3[A\text{-}M]}$（必要量の

7倍）で，ようやくFull GCを抑え，$DD_{O2,\ O3[A+M]}$なみの実行速度が達成されている．

以上の実験のように，Nanafusiのようなアプリケーションでは，メモリ設定だけで殿堂入りを防ぎ，GCの効率化を行うのは難しい．一方，$DD_{[A+M]}$は，旧世代からの参照の影響を新世代GCに及ぼさないようにするものであり，従来の世代別GCの効率化手法をそのまま享受することができた．

5. 議　　論

まず，各種の遅延隠蔽や反応速度向上を目指したXML処理系を紹介し，本研究との関係を考察する．haskell上のXML処理系であるHXml[8]は，haskellの遅延評価機能を用いることで，本研究同様XMLをTreeとして扱いつつ，入力XMLの受信完了を待たずに処理を進めることができる．一方，本研究とは異なり，合成演算ライブラリを提供していない．このため，プログラマ自身が合成演算を記述する際に逐次的に処理を行うと，`Join`演算などでは早い段階で解を出力することは難しい．加えて，一方のデータアクセス時にデータ受信待ちが発生した場合も，他方のデータを読み進めるといった遅延隠蔽効果は期待できない．また，haskell上のXML処理系であるHaXml[9]やHaskell XML Toolbox[10]は，Nanafusiなどと異なり，入力XMLの受信や構文解析完了を待ってからでないとXMLデータに対してアクセスできない．

XMLのパーズ処理の進捗をコントロールする方法として，最近XML pull parser[11),12)]が提案されている．プログラマが必要に応じて要素のタグなどにアクセスし，順次構文解析を自ら進めていくことができる．ただし，処理系が返すデータ構造は抽象度の低いデータ構造であり，XMLデータからTree構造を作成したい場合には，プログラマが自らTree作成を行わなければならない．

合成演算について，その出力開始を早期に取得するための研究は，データベースシステムでも行われている[13),14)]．たとえば，文献14)では，早い段階でより多くの出力結果を得るために，複数の入力ストリームを並行処理しながら，より多くの解を出力可能なストリームの処理を優先する．このような処理は，ワーキングセットの拡大を優先するスケジューリングにもなりうるため，実行速度低下につながる可能性も存在している．Nanafusiでも，公平なスケジューリングを行っているが，初期の解取得とワーキングセット抑制の関係については，今後の課題として考えたい．

次に，本研究で述べた各種実装手法について考察する．まず，本研究では独自のスケジューリングを導入するため，Java上にスケジューラを配置，明示的スケジューリングを行った．ただし，Javaのスケジューリング方針が拡張可能であれば，明示的スケジューラの必要性はなかったと考えられる．今回のケースでは，I/O受信を再開した場合や，各種スレッドを再開した場合のスケジューリング順位の設定が必要と思われる．Producerスレッドの GCについても，Javaシステム側からのサポートがあれば，問題の解決は簡単であった．今回必要であったのは，休止状態にあるスレッドについて，それを当初からGCのルートセットに加えず，参照関係でスレッドがliveと判定されたときのみルートセットに加えるというものである．オブジェクトに連想されたデーモンスレッドのような機能である．別のいい方をすれば，これらの機能を通常のJava上で導入するため，今回，各種実装手法を提案したともいえる．

最後に，世代別GCについて述べる．今回問題となった性能低下は，旧世代でゴミが発生したことを，新世代のGCですぐに利用できていない点に問題があった．本実装では，新世代の参照関係を改良することで問題を回避したが，一方で，旧世代GC改良（文献15)など）の重要性を示す事例ともいえる．

6. ま と め

本研究では，Producer/Consumerスタイルで実装された要求駆動型XML計算環境の性能向上を，スレッド管理とメモリ管理の両面から行った．

スレッド管理の目標は，データ転送速度に応じたスレッドスケジューリングを行うことで，処理系の反応速度を落とすことなく，スレッド切替えのオーバヘッドを削減することにある．そのために，低速転送されるデータに対してはスレッド切替えを優先するが，高速データに対してはスレッド切替えを抑制するという戦略をとった．一般のJava処理系上に同戦略を実現するにあたり，高速パーザ用のスケジューリング管理機構を独自に設け，公平なスケジューリングやデータ転送速度の管理を行っている．この適応的戦略により，データ転送速度にかかわらず反応速度向上しつつ全実行時間にかかるオーバヘッドの削減が行えた．

一方，メモリ管理の問題は，Java処理系上で効率的にメモリ資源の回収を行うことである．現在，世代別GCがその効率性から広く採用されている．ただし，Producer/Consumerの実装を行う場合，古いオブジェクトから新しいオブジェクトへの参照関係が多用され，新世代のGCだけではゴミが回収できない

ケースが存在する．本研究では，プログラム側でオブジェクト参照管理を積極的に行うことで GC の効率化を行った．また，実行途上で不要となった Producer スレッドの回収方法についても紹介した．

従来より，Nanafusi では記述性を指向しており，加えて，上記の最適化により，Producer/Consumer スタイルで実現された本処理系が，プログラムのマルチスレッド化などを要するイベント駆動プログラムと比べても，2〜4%の実行時オーバヘッドで実行可能となった．この数字はデータ転送速度に関わらないものであり，また，最初の解を返すまでの反応時間についても良好な結果を得た．これらの実装技法は，ネットワークデータを対象とした Producer/Consumer 実装全般に広く利用できると考える．

参 考 文 献

1) 鎌田十三郎，新村健治，田中敬一：要求駆動による遠隔 XML データ処理の効率化，*Proc. SACSIS2003* (2003).
2) 新村健治，山中真和，鎌田十三郎：要求駆動型 XML 計算環境 Nanafusi の実装と評価，情報処理学会論文誌：コンピューティングシステム，Vol.45, No.SIG 11(ACS 7), pp.238–247 (2004).
3) Tutorial of Nanafusi. http://www.cs26.scitec.kobe-u.ac.jp/~pl/nanafusi/
4) XMLData Repository (2002). http://www.cs.washington.edu/research/xmldatasets/
5) TPC Transaction PRocessing Performance Council (2001). http://www.tpc.org/tpch/
6) Java HotSpot VM Options. http://java.sun.com/docs/hotspot/VMOptions.html
7) Tuning Garbage Collection with the 5.0 Java Virtual Machine. http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html
8) HXML (2002). http://www.flightlab.com/~joe/hxml/
9) HaXML (2002). http://www.cs.york.ac.uk/fp/HaXml/
10) Haskell XML Toolbox (2002). http://www.fh-wedel.de/~si/HXmlToolbox/
11) XML Pull Parser. http://www.extreme.indiana.edu/xgws/xsoap/xpp/
12) Streaming API for XML. http://jcp.org/en/jsr/detail?id=173
13) Urhan, T. and Franklin, M.J.: XJoin: Getting Fast Answers from Slow and Bursty Networks, Technical report, Maryland (1999).
14) Urhan, T. and Franklin, M.J.: Dynamic Pipeline Scheduling for Improving Interactive Query Performance, *27th VLDB Conf.* (2001).
15) Blackburn, S.M. and McKinley, K.S.: Ulterior Reference Counting: Fast Garbage Collection without a Long Wait, *Proc. OOPSLA*, pp.344–358 (2003).

(平成 17 年 1 月 24 日受付)
(平成 17 年 5 月 8 日採録)

山中 真和

1979 年生．2003 年神戸大学工学部情報知能工学科卒業．2005 年同大学大学院自然科学研究科情報知能工学専攻修了．2005 年 4 月より日本電信電話株式会社．XML 技術や Web サービス等に興味を持つ．

鎌田十三郎（正会員）

1970 年生．1993 年東京大学理学部情報科学科卒業．1995 年同大学大学院理学系研究科情報科学専攻修士課程修了．1998 年同博士課程単位修得退学．1996〜1998 年日本学術振興会特別研究員（東京大学）．1998 年より神戸大学工学部助手．博士（工学）．並列・分散処理，言語処理系等に興味を持つ．日本ソフトウェア科学会，ACM 各会員．