Tiny Konoha: ET ロボコン向けのスクリプト処理系の簡素化

志田 駿介† 菅谷 みどり* 倉光 君郎†

ET ロボコンは，2005 年から開催されている組み込みソフトウェアのロボットコンテストで，与えられたロボット筐体の上でモデル記述とソフトウェア開発を競うことを目的としている．近年，TOPPERS OS など，ソフトウェア開発環境の整備も進められているが，こうした基本ソフトウエアの提供により，より多くの開発者がよりソフトウエアを開発しやすい環境が整えられている．我々はスクリプトの利用により，競技会やチューニングの際のプログラムのコンパイル時間を短縮することができると考え，開発している Konoha 言語を組込みシステム向けに簡素化し，Tiny Konoha として提案するものとした．Tiny Konoha は，主に Konoha の省メモリ化を行うことで，ロボコン競技用ハードウエアである NXT の上で動作する実用的な実行環境を提供することに成功した．論文ではこの取り組みの経緯と簡素化のための設計,実装及び評価を述べる．

Tiny Konoha: Downsizing a Scripting Language for ET RoboCon Hardware

SHUNSUKE SHIDA† MIDORI SUGAYA* KIMIO KURAMITSU†

ET RoboCon is a robot contest for embedded system software that has been held since 2005. The purpose of this contest is to develop a competitive software and model that achieves good performance by using a given robot hardware. These days, to improve the efficiency of software development on the robot hardware, basement software such as TOPPERS Operating System have been provided. We consider a scripting language can reduce the time of compiling to improve the efficiency of development by statically linked with OS at tuning time. Based on this idea, we have developed Tiny Konoha based on our developed scripting language Konoha Script, and successfully achieved the purpose of providing a practical runtime environment by downsizing of Konoha for the robot hardware NXT. In this paper, we will describe the design and evaluation of our developed system.

1. はじめに

近年，組み込みシステムにおけるソフトウェア開発の手法としても，スクリプト言語の活用が始まっている．本稿では，ET ロボコン競技会の公式ハードウェアである Mind Storm NXT[1]上，スクリプト言語処理系を搭載する試みを行った．技術課題は，スクリプト言語処理系のコンパクト化の実現である．我々は，静的スクリプト言語として Konoha の設計と開発を行ってきた．Konoha は，もともと組み込みシステム上の応用も想定し，たとえば，静的に型付けされた独自設計のバイトコードは，バーチャルマシン(VM) からの動的型検査の除去と，さらに CPU 資源の少ないマシンでも良好な実行性能を実現している．

一方，Konoha のスクリプト処理系は，コーディングとプログラム実行の一体化された開発サイクルを実現するため，開発環境（パーサからコード生成）と実行環境（言語ランタイムとバーチャルマシン）の統合が必要であった点で開発環境と実行環境が分離した Java VM とは異なる．そのため，従来の Konoha は開発環境の分だけランタイムが大きくなり，我々の Konoha 開発経験では，最小限の構成としてもバイナリサイズで 100KB 以下にはできなかった．このサイズでは，明らかに 64KB の NXT 上で動作させることはできない．

本研究の目的は，競技会に出走可能なレベルのソフトウェア開発が可能なスクリプト言語処理系の実現である．ET ロボコン競技会の公式ハードウェアの上で，スクリプト言語処理系が実現されれば，競技会やチューニングの際のコンパイル時間を短縮することができる．目的の実現のために，本研究では ET ロボコン向けのスクリプト処理系の簡素化である Tiny Konoha の設計および実装を行った．Tiny Konoha は，Konoha とソースコードレベルの互換性を維持しながら，より極小な言語ランタイム，バーチャルマシン，ガベージコレクションを実現している．論文では，言語設計の背景を含めた，実装，評価について述べる．

本論文の構成は以下のとおりである．2 節では，本研究の背景となる ET ロボコン競技会，公式ハードウェア，公式な開発環境，さらに競技会からみたスクリプト処理系への要求を述べる．3 節ではスクリプト処理系の省資源化の取り組みと課題について述べる．4 節では Tiny Konoha の設計と実装について述べ，5 節で性能評価を行う．6 節で関連研究を挙げ，7 節にて結論を述べる．

2. ET ロボコン競技会

ET ロボコンは，社団法人組込みシステム技術協議会

†横浜国立大学 工学府
*横浜国立大学 未来情報通信医療社会基盤センター

(JASA) が主催する組込み技術者の人材育成の一環として実施されるソフトウェア技術を競うコンテストである[2]. ET ロボコンの目的は，組込みシステム開発分野および同教育分野における若年層および初級エンジニアへの分析・設計モデリングの教育機会を提供することである．こうした目的から，ET ロボコンは大学の初等ソフトウェア教育や，新人研修などに利用されている．本節では主に走行体のハードウェア制約, プログラミング環境について述べる.

2.1 ハードウェア性能（資源制約）

ET ロボコンで用いられる走行体は，Lego 社がマサチューセッツ工科大学と共同開発して 1998 年に発表した Mindstorm 自立型ロボットである．この中でも，ロボコンでは NXT を用いた二輪倒立振子型ライントレーサにより競技を行うものと規定されている．本倒立振子型ライントレーサで使用できるアクチュエータはモータが２個，センサは３種類（光センサ，タッチセンサ，ジャイロセンサ）で，これらを利用しながら，ライントレースのコースを攻略するものとなっている．

NXT 本体内部には，32 bit マイクロプロセッサ(ARM7) が内蔵されており，256KB のフラッシュメモリと，64KB の SRAM が搭載されている．また，信号の入力ポート数は４, 出力ポート数３で, Bluetooth 機能を搭載している．これらのハードウェアは，低価格な組み込みシステムのスペックの代表例と考えられる．

2.2 開発環境

我々は NXT 用のプラットフォームとして TOPPERS/JSP を採用した[3]．本プラットフォームは μ ITRON4.0 仕様に準拠したリアルタイム OS である TOPPERS/JSP カーネルをベースとして開発されており，(1) リアルタイム OS で I/O ドライバの API，(2) ANSI C 言語開発環境，(3) μ ITORN に対応したマルチタスクスケジューリング機能が利用できる．また NXT のモータやセンサ等のデバイスに対する倒立振子ライブラリが用意されており，倒立振子 API を 4ms 毎に呼び出すことで走行体を動作させることができる．ET ロボコン協議会のレギュレーションでは，リアルタイム OS と倒立振子ライブラリへの修正は禁止されている. 我々は TOPPERS/JSP カーネルでの開発環境を Linux 上に構築し，GNU-GCC-ARM-ELF コンパイラを用いてカーネル (1.4.4-nxt)，およびその上でドライバ，ライブラリのコンパイルを行った．本環境を用いて NXT 走行体用のプログラムを NXT に転送し，基本的な振子の倒立制御をしながらライントレースを行うものとした．

2.3 開発への要求

ET ロボコンでの走行体プログラムの開発は，机上で行うことが可能である．しかし，実際のコースにあわせてセンサが取得する物理値を調整し，さらにアルゴリズムやモデルを工夫することにより性能を向上させる必要がある．この際，各チームは実際のコースを利用してコースに合わせて指示値（速度指示値や旋回指示値）を変更してテストを行う．しかし，こうしたチューニングでは，プログラムのコンパイル，ロード時間はコストとなる．特に OS を利用している場合には，OS やドライバをプログラムの修正のたびにスタティックリンクでコンパイルし直し, さらに, リモートホストにロードしなくてはならないため，こうした時間のロスは少なくない．この時間を短縮することは開発効率の向上の重要な課題の一つとなっている．こうした時間のロスのうちコンパイル時間は，スクリプト言語の利用により軽減することができる．また，実行環境をハードウエアから独立して構築できる利点もあり，総合的なロード時間を減らすことができると考えられる．しかし，スクリプト言語は一般的には VM や GC などの実行環境が必要となるため，省メモリのハードウェア環境で利用することは一般的に困難とされており十分実現されていない.

3. スクリプト処理系の省資源化の課題

本研究では，開発効率の向上のために，ET ロボコンのハードウェア向けにスクリプト言語を開発することを目的とする．手始めに，我々の研究室で開発している Konoha スクリプト言語の設計について述べ，ハードウェア制約に対する課題について述べる.

3.1 Konoha と Mini Konoha

Konoha は，ユビキタスコンピューティングを応用分野としたスクリプト言語処理系として開発が始まった．当初の開発計画から，組込みシステム上の動作は対象となっていたが，省資源よりアプリケーション記述性に重点が置かれてきた．その後，静的型付けや型推論機構など，プログラミングしやすさ[4]の向上を図る拡張を加えられ，言語仕様も処理系も肥大化する傾向にあった.

Mini Konoha は，言語仕様の最小化と文法拡張可能なパーサ技術[5]による新しい試みの処理系である．言語仕様最小化により，クラス定義や型推論などの機能は取り除かれライブラリ拡張になっている．主な残った言語仕様は以下の通りである：

- C-スタイルの文法（subset of C)
- 静的型付け（static typing)
- 値の型: boolean, int, String, Array, Func(function)
- メソッドの定義/メソッド呼び出し
- if ステートメント, return ステートメント

ET ロボコン走行体向けスクリプト言語処理系は，Mini Konoha をベースにコンパクトを行う予定であった．しかし，Mini Konoha であっても，2.2 節で述べた資源環境では，スクリプトの実行は不可能であった．そこで，次節では，NXT 環境での実行をまず目指すため，より省資源化を行う Tiny Konoha の実装について述べる．本節では，省資

源化前の Mini Konoha の実装を紹介し，省資源化に対する課題を述べる．

3.2 ソフトウェアスタック

Mini Konoha は，名前の示すとおり，従来の Konoha に比べ，コンパイル済みの実行コードで 1/6 程度，100KB 程度に抑えることに成功している．

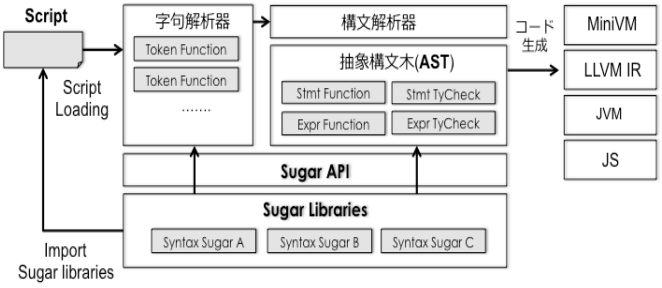

図 1 Mini Konoha のアーキテクチャ

図 1 に Mini Konoha のアーキテクチャを示した．Mini Konoha は，ソースコードレベルでのコンパクト化を目標に開発し，言語実装は実用性能の高いスクリプト処理を実現する実装となっている. これらは, 後から述べるとおり，Mini VM や GC のメモリ使用量に大きな影響を与えている．以降の節にて, Mini VM と GC について述べる．

3.3 Mini VM

Mini VM は，Mini Konoha 用に開発されたレジスタ型の VM 実装であり，言語ランタイムの多くを関数呼び出しの形で実現しているため，VM の命令数は高々十数個となっている．命令セットは，オブジェクトモデルをベースにしているため，整数演算は Int オブジェクトの+メソッドの呼び出しという形で実行する．

Mini VM は，命令セット数は最小化してあるが，省資源を目的とするより, Intel アーキテクチャの CPU のパフォーマンスを重視する実装となっている．実装上は，各命令をバイト単位の代わりにワード単位でエンコーディングし，命令長も 8 ワードに固定している．これにより，キャッシュヒットや分岐予測が効率化し，命令アクセスの高速化を実現している．そのため，簡単なスクリプトであってもかなりのメモリを消費する実装となっている．

3.4 GC

Mini Konoha は，GC 機構のモジュール化を行い，いくつかの GC 実装(インクリメンタル GC，世代別 GC など)を切り替えることができる．標準では，ビットマップによる移動のない世代別 GC 技術を採用し，スクリプト処理系の起動時に, (一般的なテキスト処理等のスクリプトに最も適しているサイズとして) 16MB のヒープ領域を確保する．ただし，GC アルゴリズムや初期値を切り替えることが可能である．他方，プログラミング言語処理系の GC アルゴリズムは，各言語のオブジェクトサイズに依存する．つまり，オブジェクト最小サイズがアロケータの単位として使われる．Mini Konoha は，Intel アーキテクチャの L2 キャッシュラインの大きさにあわせ, 1 オブジェクトあたり, 8 ワード使う設計になっている．

この実装では，ET ロボコン走行体上では，64KB メインメモリを全てアロケータに利用できても，最大 2000 個のオブジェクトしか生成できないことになる．次節では，これらの Mini Konoha の実装をどのように最小化を行ったかについて述べる．

4. Tiny Konoha の設計と実装

本節では，NXT 向けの Tiny Konoha の設計実装について述べる．

4.1 全体構想

スクリプトの利便性は，ユーザがコンパイルを行わずにプログラムを実行できることにある．Mini Konoha ではソースコードのコンパイルと実行を同時に行う事が可能だった．しかし ET ロボコン向けハードウェアでは，スクリプト言語のパーサの搭載は困難であった．なぜならパーサが生成する抽象構文木はスクリプトのサイズに比例して大きくなるため，メモリを大量に消費してしまう問題があるためである．また，パーサ自体の実行ファイルサイズも無視することはできない．スクリプトの利便性を最大限に生かすために, 我々はホスト側でバイトコード生成までを行い，Bluetooth 通信を行って，バイトコードを転送し，NXT 上の Tiny VM 上で動作させるものとした．パラメータを入れ替えながら実行が可能であるため，インクリメンタルな開発が可能である．

```
K.import("konoha.nxt");
K.import("konoha");
int speed = 50;
while (true) {
  System.balanceControle(speed, 0 /* angle */);
  System.waiSem(); // 3ms待機
}
```

図 2 走行体を直進させるスクリプト

図 2 に Tiny Konoha で記述した走行体を直進させるスクリプト例を示した．1 ,2 行目はそれぞれ NXT を走行させるためのパッケージと基本パッケージのインポート，3 行目以降に直進させるための API 呼び出しを示している．また，図 3 にスクリプトを実行させることで走行している NXT に対して, ホスト側から停止命令を送信するインタラクティブな操作の概要を示した．NXT の走行速度は変数 speed によって決定している．この speed 変数をターミナル上から操作するスクリプトをコンパイルし，バイトコードを転送することができればスクリプト言語の特徴といえる対話実行を実現することができる．

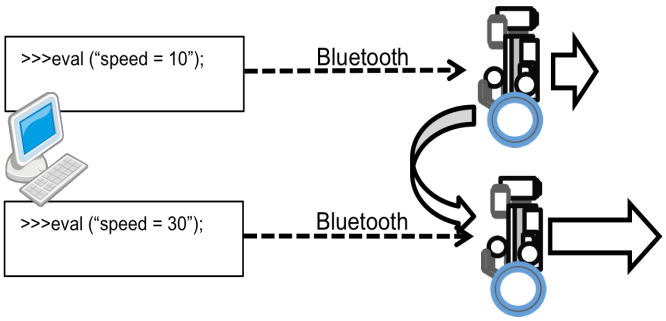

図 3 スクリプトによるインクリメンタル開発

3 節で述べたように, 現状の Mini Konoha の実装は, フロントエンド, バックエンド, ランタイムを構造的に独立させている. 我々は, パーサを独立させても, スクリプト言語として機能することを既に確認していたことから, ET ロボコンのロボットに対しても, ユーザとの対話実行環境を提供できると考えた.以降の節で, 具体的なアーキテクチャについて述べる.

4.2 言語システム

図 4 に, ET ロボコン向けに実装したスクリプト言語システムのアーキテクチャを示した. ここでは, ロボット制御プログラムが Konoha スクリプトで記述されており, またインタラクティブにスクリプトは変更できるものとする. コンソールで記述されたスクリプトのコードは, Mini Konoha のパーサによりフロントエンドでの字句解析, 構文解析が行われ, AST を生成する.

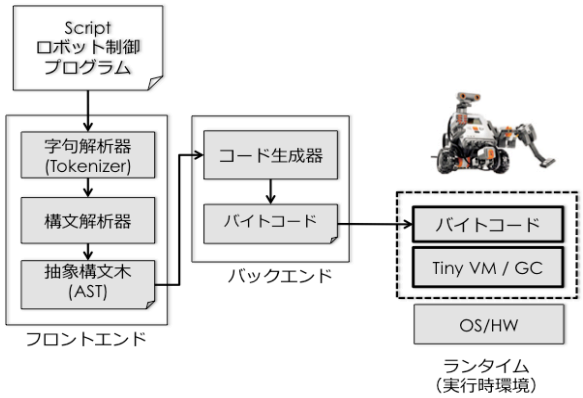

図 4 ロボット向け言語システムアーキテクチャ

通常は構文解析器から生成した抽象構文木 (AST)からバイトコードを生成し, これを Mini Konoha の仮想マシン上で動作させることでスクリプトを実行する. しかし 4.1 節で述べたとおり, NXT 上のランタイムのメモリ量制約は 64KB と限られており, パーサやコード生成器を NXT 上で動作させることは困難であった. そこで, 我々はターミナル側でバイトコードの生成までを行い, そこで生成したバイトコードを, NXT 上の TinyVM 上へ転送して実行する構成とした. 実行時のランタイムを含む VM と GC は, さらに省メモリ化を行うことで, TOPPERS/JSP, および NXT 動作に必要なライブラリと 一緒に提供できるものとした. 以下に Tiny Konoha とそのランタイムやライブラリの構成について述べる.

4.3 Tiny Konoha 言語ランタイムとライブラリ

4.3.1 背景

当初我々は, 3.3 節に示した Mini Konoha 用のバイトコードを実行する仮想マシン Mini VM と 3.4 節で示したメモリ管理を行う GC を, NXT に移植することで, 言語のランタイム (実行環境) を提供することを想定していた. しかし, Intel アーキテクチャに特化した Mini VM は命令長が 8 ワードと長く, 簡単なスクリプトでも大量のランタイムのメモリを使用してしまうことから, NXT 上の 64KB のメモリ上では, バイトコードを十分に実行できなかった. ET ロボコン協議会のレギュレーションでは, リアルタイム OS と倒立振子 API への変更は行えないが, これらのライブラリだけでメモリの半分を必要としてしまい, Mini Konoha の移植を行っただけではコンパイル後の実行コードを 64KB 以内に抑えることすら難しかった. この事から, 我々は新たに省メモリ化を目指して Tiny Konoha 向けの VM,GC, ライブラリの開発を行った.

4.3.2 全体アーキテクチャ

Tiny Konoha の NXT 上での実行環境 (ランタイム) のソフトウェア構成を図 5 に示した. Tiny Konoha では, バイトコードを実行するための Tiny VM, GC を提供するものとした. また, 低レベルのメモリ管理のための klibc を実装し, GC で必要となる ヒープメモリの malloc()/free() などのアロケータを実装して利用できるものとした. また, ロボットを動作させるために必要となる基本的なデバイスドライバや倒立振子のライブラリは, konoha.nxt の FFI(Foreign Function Call)[a]を開発して利用することで Tiny Konoha から呼び出せるようにした.

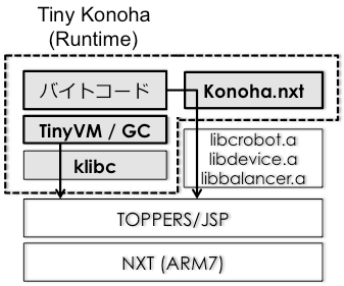

図 5 Tiny Konoha の構成

4.3.3 ライブラリ

Tiny Konoha は, スクリプトを実行する際の, 高レベルなメモリ管理は GC が行うが, ヒープ領域やスタック領域

a FFI とは, あるプログラミング言語から, 他言語で実装されたコンポーネントを利用するためのインターフェースである. 開発者は, FFI を用いることにより, 様々な言語で実装された既存のライブラリを使用し, その機能・性能を享受することができる

などのランタイムが利用するための低レベルメモリ管理は，malloc()/free()などの libc が提供する関数を利用して実現している．しかし，malloc()/free()の使用個所は主に Tiny Konoha の初期化時と終了時であり，頻繁に呼び出されるものではない．このためこれらの関数は独自のライブラリ klibc (konoha libc)として実装した．これは実行速度の向上よりもコンパイル後の実行コードサイズの削減が必要とされたことによる.ランタイムで必要となる主な動的なメモリ領域は，ヒープ領域とスタック領域であるため，これらに必要な API として,以下の関数の実装を行った．

```
static void heap_init(void);
static void *tiny_malloc(size_t size);
static void tiny_free (void *ptr)
```

4.4 Tiny VM

Tiny VM は NXT 上で動作する Tiny Konoha 向けのレジスタ型の仮想マシンである．

4.4.1 バイトコードの命令セット

Tiny VM では，Konoha で記述されたプログラムを実行するために最低限必要な命令を基本命令セット(表 1)として定義した．執筆時現在，基本命令は 13 命令である．例えば，Tiny Konoha プログラム int n = 0; while (n > 100) n++; は，次のようなバイトコードが生成される．

```
NSET 1 0          #定数 0 をセット
NMOV 3 3          #レジスタ間の移動
NSET 4 100
SCALL 3 4 4 9     #Int.opLT 関数の呼び出し
JMPF 2 10         #opLT が偽の場合は RET 命令へ
NMOV 3 1
NSET 4 1
SCALL 3 4 4 6     #Int.opADD 関数の呼び出し
NMOV 1 2
JMP 1             #NMOV 命令へ
RET
```

命令セットは今回，非常に基本的なものしか実装していない．これは，3.3 節で述べたとおり関数呼び出しでオペレータを含む多くの処理をカバーできるためである．

表 1 バイトコードの命令セット

定数命令	NSET ra n / OSET ra n
レジスタ操作	NMOV ra rb / OMOV ra rb
条件分岐	JMP jmppc / JMPF jmppc ra
関数呼び出し	SCALL / VCALL
フィールドアクセス	NMOVX ra rb bx OMOVX ra rb bx XNMOV ra ax rb XOMOV ra ax rb
リターン	RET

以下に，それぞれの詳細を表 1 に従って説明する．

・定数命令 ； TinyVM の ra 番目レジスタ上に定数をセットする VM 命令である．整数，小数，真偽値の場合は NSET が用いられ，文字列や配列などのオブジェクトの場合は OSET 命令が用いられる．

・レジスタ操作 ：変数のレジスタ間の移動を行う VM 命令である．NSET, OSET と同様，非オブジェクトの場合は NMOV が，オブジェクトの場合は OMOV が用いられる．レジスタ rb 番目の値が ra 番目にセットされる．

・条件分岐 ： JMP 命令の場合は無条件で jmppc 番目のバイトコードへとジャンプを行い，JMPF 命令は ra 番目のレジスタの値が偽の場合にジャンプを行い，真の場合は一つ次の命令へと処理を継続する．

・関数呼び出し ：関数呼び出しを行う VM 命令である．ライブラリの呼び出しは SCALL 命令，ユーザ定義の関数は VCALL 命令が使用できる．

・フィールドアクセス ：NMOVX と OMOVX は，フィールドからレジスタへの値の移動を行い，XNMOV と XOMOV はレジスタからフィールドへの値の移動を行う．

・リターン ： 関数を終了するための VM 命令である．

Mini Konoha と Tiny Konoha のバイトコードの構造体レイアウトを図 6 に示す．Tiny VM は 3.3 節で述べた Mini VM 用のバイトコードと同一の命令セットを持つが，組み込み機器という資源な限られた環境であるという点，パーサが生成したバイトコードを転送している点からいくつかの VM 命令のオペランドが変更されている．

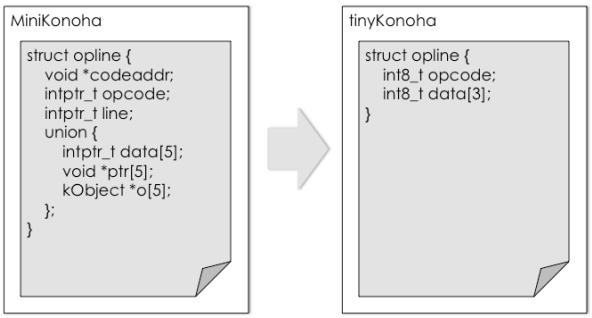

図 6 Min Konoha と Tiny Konoha 構造体レイアウト

4.4.2 バイトコードのメモリ使用量削減

VM が解釈するバイトコードはスクリプトの行数が増えるとともに増加していく．Mini VM のバイトコードは一命令あたり 32 バイトとなっており，わずか 2000 命令で NXT の全メモリを消費してしまう．このため Tiny VM では実行効率よりもメモリ使用量について着目しバイトコードの設計を行った．

Tiny VM はまず Mini VM をベースとして設計を行なっている．Mini VM では実行高速化のため，以下の２つの工夫を行なっている．

・直接スレッデッドコードによる命令ディスパッチ高速化
・定数値のバイトコードへの埋め込み

ただし,これらの高速化はバイトコード 1 命令あたりのメモリ使用量を増加させるものである．我々は Tiny VM を

構築するにあたり実行速度の低下を抑えつつメモリ使用量を抑える設計を行った.

直接スレッデッドコードはインタプリタの命令ディスパッチ高速化を行うための手法である.インタプリタは通常,命令のディスパッチに switch-case 文を用い，命令ごとに処理を分岐させる．直接スレッデッドコードでは，事前にバイトコード中に各命令バイトコードの命令処理部分のアドレスを埋め込んでおき，インタプリタはバイトコード中に書き込まれているアドレスに直接分岐することで，単純な switch-case を用いた場合と比べて命令のディスパッチコストを削減するものである.

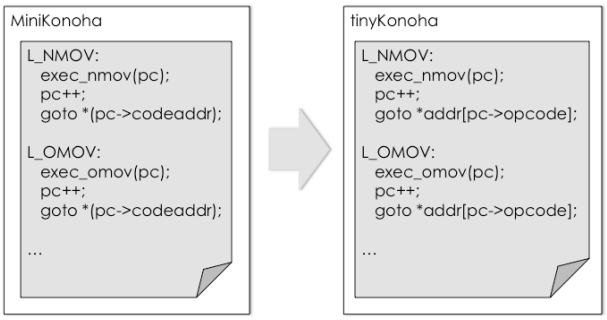

図 7 命令ディスパッチ方法の変更

図 7 に示したとおり，TinyVM ではメモリ使用量の観点から各バイトコードに 8 ビット整数値としてオペコードを埋め込み，また命令のディスパッチ方法に間接スレッデッドコードを用いたディスパッチ方法を用いることとした.

また，Mini VM では，メソッド呼出命令と定数を扱う命令に関して，メソッドオブジェクトや定数値をバイトコードへの埋め込み，実行の高速化を行なっている．Tiny VM では一箇所に定数オブジェクトを集約し，インデックス値のみをバイトコードに埋め込む形を取ることとした.

これらのメモリ削減のチューニングによって我々は 1 命令あたりのメモリ使用量を 4 バイトにまで削減した.

4.5 ガベージコレクション

3.4 節で述べたとおり，Mini Konoha では GC 機構をモジュール化し，いくつかの GC 実装を提供していた．しかし, NXT のハードウェア制約から複数の GC 実装を搭載することは現実的ではない他，Mini Konoha のデフォルトの GC 実装であるビットマップ GC は，オブジェクトの管理のためにビットマップが必要なためメモリ使用量が増加してしまう問題があった．また， Mini Konoha は起動時に 16MB のメモリを確保する仕様であるため，NXT 上で動作しない問題があった．この事から，GC のために必要なメモリ領域がマークを行うための 1 ビットのみと少ない MSGC を実装するものとした. Tiny Konoha の MSGC は，Mini Konoha で提供されている MSGC モジュールに修正を行う形で実装した．最大の修正点は，アロケータの単位の変更である． また 3.4 節で述べたとおり，Mini Konoha のオブジェクトは 8 ワード使用しているが，Tiny Konoha の場合は 4 ワードに削減した.

Mini Konoha の GC アルゴリズムの中にはオブジェクト毎に GC 用のフィールドを必要とするものがある．例えばオブジェクトの非参照数をカウントし，0 になった瞬間に領域の解放を行うリファレンスカウント GC では，オブジェクト毎に非参照数のカウントのためのフィールドが必要になる．Mini Konoha ではあらゆる GC アルゴリズムに対応するために，オブジェクト毎に 1 ワードの GC 用フィールドを用意しているが, MSGC では必要な領域は 1 ビットのみでよいため，Tiny Konoha では GC 用のフィールドは必要無い．この他にもオブジェクトのヘッダ情報を削減することで，オブジェクトを 4 ワードに収めている.

4.6 メモリレイアウト

64KB のメモリしか持たない NXT 上でリアルタイム OS との連携を行った上でスクリプトを動作させるために，データ構造などの省メモリ化を行った.

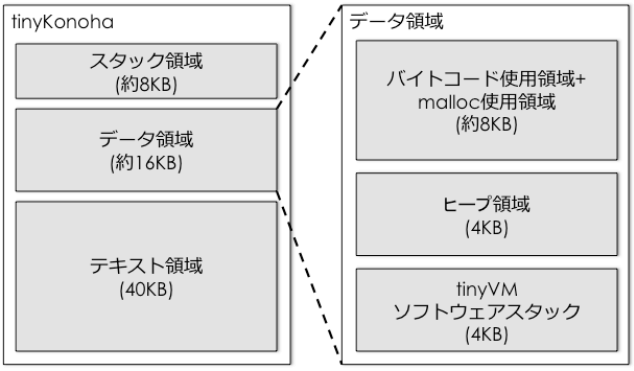

図 8 Tiny Konoha のメモリレイアウト

図 8 に Tiny Konoha のメモリレイアウトを示す．リアルタイム OS と NXT の倒立振子 API，標準 C ライブラリを静的リンクした Tiny Konoha 全体のテキスト領域は 40KB となっている．OS と倒立振子ライブラリのサイズが大部分を占めており, 我々が実装した Tiny Konoha 自体のライブラリサイズは 8KB とテキスト領域の 2 割程度を占めるに留まっている．標準 C ライブラリとして newlib-1.14.0 を用いているが，newlib で定義されている C 言語の標準関数の内，Tiny Konoha で使用しているのは memcpy()と memset() のみである．これは newlib がいくつかのライブラリの集まりであり，一つの関数を使ったとしても newlib 全体をリンクする必要が無いため，使用する関数を最小限に抑えることで不要なライブラリのリンクを避けることができるためである．テキスト領域のみで 64KB というハードウェア制限の 6 割以上を使用してしまう結果となったが，2.3 節で述べたとおり，リアルタイム OS や倒立振子 API への修正を禁止されているため，これ以上のテキスト領域の削減は難しい．残る 24KB のうちデ

ータ領域は約 16KB 確保されており，残りをスタック領域として用いる．このうちデータ領域から，GC のためのヒープ領域や TinyVM のソフトウェアスタックの確保などを行う必要がある．TinyVM のソフトウェアスタックには 4KB が，GC のためのヒープ領域には 4KB が割り当てられ，残りの領域をバイトコードと malloc()のために割り当てている．これは，データ領域の小ささから，Mini Konoha では行われるヒープ領域の拡張は行わず，アロケートが行えない場合は即スクリプトの実行が停止するといった制限をかける必要があったためである．

5. 性能評価

5.1 コンパイル時間の削減

まず始めに，C 言語のソースコード+リアルタイム OS のコンパイル時間を計測した．その結果，最大で 50 秒となった．ソースコードの部分的な修正であれば，これほど大きなコストとならないが，パラメータ等の変更の度にコンパイルと転送が必要な点は，チューニングにおいて大きなコストである．スクリプトはこうしたロスが発生しない事から，開発コストを削減できるといえる．

5.2 メモリ効率

4 節でも述べたとおり，Tiny Konoha では省メモリ化のための構造体のコンパクト化や種々のチューニングを行っている．Mini Konoha と Tiny Konoha の構造体のサイズなどの違いを表 2 に示す．

表 2　Tiny /Mini Konoha 構造体サイズの違い

	Mini	Tiny
実行コードサイズ	100KB	40KB
オブジェクトサイズ(最大)	8 ワード	4 ワード
オブジェクトのヘッダ	4 ワード	1 ワード
ヒープ領域	16MB	4KB
ヒープ領域の拡張	行う	行わない
バイトコード長	32 バイト	4 バイト
ダイレクトスレッデッド	有効	無効

上記のような変更を行っているにもかかわらず，Tiny Konoha では Mini Konoha と同様のスクリプトを動作させることに成功している．しかしその一方で，NXT の資源制約からヒープ領域の拡張が行えない点が，オブジェクトの生成を頻繁に行うプログラムの作成を難しくしてしまっている．生成したオブジェクトがすぐに GC 対象となる場合は問題無いが，ヒープ領域の拡張は行えない．また 4.6 節で述べたとおり，Tiny Konoha のメモリは NXT のメモリ 64KB をほぼ使い切っている．現在は実行コードサイズが 40KB を超えてしまっており，これ以上の実行コードの増加はその他の領域のサイズを圧迫してしまう．このため Tiny Konoha の機能拡張を行う度に，システム全体のメモリレイアウトの変更を行っていかなければならない．

5.3 実行速度

表 3 実行速度の差(単位: ms)

	Fibonacci(30)	ループ構文
TinyVM	30,904	14,901
C 言語	807	306

NXT 上で Fibonacci 数を求める Fibonacci 関数と，ループカウンタのインクリメントのみを 1,000,000 回行う while 文について，Tiny Konoha と C 言語での実行速度の計測を行った．Tiny Konoha のスクリプトは，C 言語で記述した場合に比べて処理速度が非常に大きくなってしまっている．しかし，この速度低下は Tiny VM の基となる Mini VM でも起こりえるものであり，我々の想像を超えるものではなかった．一方で，Tiny VM ではメモリ使用量を削減するために 4.4 節で述べたダイレクトスレッデッドコードなどの手法を取り除いてしまっているため，Mini VM と比べると実行速度は遅くなってしまっていると考えられる．

5.4 リアルタイム性能

我々は組み込み機器の制御を行う際に求められるリアルタイム性について，Tiny Konoha のスクリプトを NXT 上で 4ms　周期で動作させることで評価を行った．実際にライントレースを行うためのスクリプトを図 8 に示す．

```
void mainLoop () {
  ecrobotSetLightSensorActive();
  while (ecrobotIsRunning()) {
    manipulateTail();
    int turn = 0;
    if (ecrobotGetLightSensor() >= 600 /* threshold */) {
      turn = 50;
    } else {
      turn = -50;
    }
    new String(" ");
    balanceControl(50, turn);
    waiSem(); // wait 4ms
  }
}
mainLoop();
```

図 9 サンプルスクリプト

スクリプトは，NXT 走行体下部に取り付けられている光センサの値を読み込み，ラインの上かの判定を行うものである．ライン上と判定された場合には右に旋回を行いながら直進し，ラインをはみ出た場合は逆に旋回を行いながら直進する．これを繰り返すことでラインに沿って NXT 走行体を走らせることができる．System.balanceControl 関数は倒立振子を行うための関数であり，NXT の走行スピードと向きを引数に渡す．このようなリアルタイム性を求められるスクリプトに GC が与える影響を考慮するため，プログラム中で文字列の生成を行い，意図的に GC を発生させている．本スクリプトで一定時間 NXT を走行させ，balanceControl 関数を呼び出す間隔を 4ms 以内に呼び出すことができているか調査を行った結果を表 4 に示した．NXT に搭載されている CPU の周波数は 55MHz であり，

時間は API から得られる最小単位である ms で表した.

表 4 リアルタイム処理時の超過回数の測定

ループ回数	GC 回数	平均 GC 時間	4ms 超過回数
100,000 回	1,428 回	0ms	0 回

balanceControl 関数を 100,000 回呼び出した時点で計測を行ったところ, ループの実行が 4ms を超えることは一度も無かった．現在は GC のためのヒープ領域が小さく，頻繁に GC が発生しているため，一回あたりの GC が 1ms 以下で行えていることが要因として挙げられる.

6. 関連研究

スクリプティング言語によるアプリケーション開発が巨大化するにしたがい, JavaScript などいくつかの言語で静的な型付けの拡張が検討されている．また，型推論を導入することで，バイトコードの性能を向上させる試みも行われている．我々は，あらかじめ静的に型付け言語を導入したが，こうした特徴は，性能を重視する組込みシステムなどにおいても有効であった[4][5].

組み込み専用スクリプティング言語として，Lua[7] など，コンパクトな実装のスクリプティング言語エンジンが開発され，製品レベルで利用されている．また，センサーネットワーク分野では，Script 専用の最小化された設計の言語[8]や，イベントドリブン処理に焦点をあて，省メモリを実現するための言語[9] が開発されている．SwissQVM は，3KB の SRAM メモリ上で動作するバイトコードインタープリタとして提案されている[10].しかし，TinyOS 上で動作することが想定されており，十分な汎用性がない．また, Android 上の Dalvik VM では，ハードウェアクセレーションを用いた高速化の試みなどが行われているが，命令拡張を後から行う場合などの柔軟性は失われる問題がある[11].

言語設計は，応用領域と求められる機能のバランスで考慮する必要である．しかし，低リソースに焦点をあてている言語はオブジェクト指向プログラミングなどが犠牲となっている．Tiny Konoha は言語仕様の最小化を行い，文法拡張可能なパーサ技術を持つ Mini Konoha を基に開発されている．このため，資源の乏しい環境では最低限の文法拡張を行い，そうでない環境では多くの文法や機能を利用するといった選択が可能である．今回も Mindstorms NXT という資源制約が大きな環境においても，必要最小限の文法拡張によりスクリプトを動作させることができた.

7. むすびに

我々はスクリプトの利用により，コンパイルやロード時間を短縮する目的で Konoha 言語のスクリプトの簡素化を目標とした Tiny Konoha の開発を行った．Tiny Konoha は少ない命令セットの VM と，省メモリレイアウトにより NXT 上で動作する実用的な実行環境を提供することができた. 現在 TinyKonoha はスクリプトを読み込み，バイトコードを Bluetooth 通信で転送する方法のみ実現しているが，今後インタプリタ型のインターフェースを用意し，スクリプト言語としての利点を生かしたインクリメンタル開発を支援する．また，リアルタイム OS との連携からスクリプトによる競技会中のチューニングのサポートまで，様々な複合的な課題への取り組みを行う予定である.

参考文献

1) レゴマインドストーム公式サイト：
http://www.legoeducation.jp/mindstorms/
2) ET ロボコン公式サイト：http://www.etrobo.jp/2012/
3) TOPPERS ET ロボコンへの取り組み:
http://www.toppers.jp/etrobo.html
4) Kimio Kuramitsu: KonohaScript: static scripting for practical use. OOPSLA, Demonstration Session, Companion 2011: pp. 27-28,
5) Kimio Kuramitsu: A Very Earlier Report on Syntax Extension with Sugar Parser + Mini Konoha, Summer United Workshop on Parallel, Distributed and Cooperative Processing, (SWopp) 2012, 8.
6)Peter Liggesmeyer, Mario Trapp: Trends in Embedded Software Engineering, IEEE Software, vol.26, no.3, pp.19-25 May/June 2009.7)
7) Roberto Ierusalimschy , Luiz Henrique de Figueiredo and Waldemar Celes. The Implementation of Lua 5.0. Journal of Universal Computer Science. Vol 11. no 5. pages 1159-1176, 2005.
8)Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali：Protothreads: Simplifiying event-driven programming of memory-constrained embedded systems, Proc. Sensys 2006, Colorado, USA, November 2006.
9) J. Hahn, Q.Xie, and P.H. Chou.：Rappit：framework for synthesis of host-assisted scripting engines for adaptive embedded systems. In CODES:ISSS '05：Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, 2005.
10) René Müller, Gustavo Alonso, Donald Kossmann: A Virtual Machine For Sensor Networks. In Proceedings of EuroSys 2007, Lisbon, Portugal, March 21-23th 2007.
11) 太田 淳，三輪 忍，中條拓伯, "Android 端末におけるハードウェアによる Java の高速化手法の提案",情報処理学会論文誌コンピューティングシステム, Vol.4, No.3, pp.115-132 (2011.5).