組込みシステム向けマルチコア・プロセッサのためのソフトウェア開発支援

髙 橋 清 隆[†,††] 柴 山 悦 哉[††]

携帯電話や PDA でもデスクトップ PC 向け高機能アプリケーション（フルブラウザ，マルチメディア・プレーヤなど）や汎用 OS（Linux など）を利用可能とするために，高性能・省電力型プロセッサへの需要が増大している．この需要にこたえるため，組込み機器向けプロセッサ・ベンダは，汎用プロセッサ・コアと特定用途向け専用プロセッサ・コアを組み合わせたマルチコア・プロセッサの開発を進めている．しかし，組込み向けマルチコア・プロセッサのためのソフトウェア開発は煩雑なものとなりがちである．これは，クロス開発を要すること，コアの種類ごとに特有の熟練を要すること，コア間通信に煩雑なプログラミングを要することなどが原因である．本論文では，携帯端末で広く使われている OMAP プロセッサ（ARM の CPU コアと TI の DSP コアを搭載）を例に，組込みシステム用ソフトウェア開発の問題点を明らかにする．そして，ここであげた問題点を解決し，一般のプログラマでもこれまで面倒だったヘテロジーニアスなマルチコア・プロセッサを用いたシステムの開発を可能とするために，異なるプロセッサ・コア向けのソースコードを混在して記述可能で，コア間通信を透過的に扱えるランタイム環境を提案する．さらに，このランタイム環境のプロトタイプ・システムを開発し，既存ソフトウェア開発環境と比較した結果についても報告する．FIR フィルタのような小規模な例題では，ソースコードサイズが 1/3 程度に削減でき，実行時オーバヘッドは，ライブラリのダイナミックリンクに要した時間を除くと，約 1%にとどまった．

Software Development Support for Multicore Processors in Embedded Systems

Kiyotaka Takahashi[†,††] and Etsuya Shibayama[††]

In recent years, there has been rapidly growing demand for feature-rich applications (e.g., full browser, multimedia player) and general purpose OSes (e.g., Linux) even on mobile terminals, such as mobile phones and PDAs. To meet the demand, embedded processors that are not only high performance but also low power consumption are indispensable. Processor venders have been shifting to multicore processors, which combine general and special purpose processor cores in a single package. Such heterogeneous multicore processors, however, makes software development cumbersome and complicated, and requires skills in cross-platform development, both cores, and low-level inter-core communication. Exemplifying challenges for the embedded software development, this paper describes software development for OMAP processors, which include ARM and TI DSP cores and have gained widespread adoption from mobile terminal manufacturers. To address those challenges and let average programmers develop embedded software easily on heterogeneous multicore processors, we propose a runtime environment that enables a unified description of source codes for different processor cores and supports transparent inter-core communication. We also report results of comparison between our prototype environment and a today's standard development environment through evaluation experiments. In a small FIR program example, the prototype reduces the source code size to about one third and its runtime overhead is about 1%.

1. はじめに

近年，携帯電話や PDA（Personal Digital Assistants）などの携帯端末上でオーディオやビデオを利用したマルチメディアサービスが提供され，デスクトップ PC で利用されるような高機能なアプリケーションが利用されている．また，これまでのリアルタ

† ノキア・ジャパン株式会社ノキア・リサーチセンター
Nokia Research Center, Nokia Japan Co., Ltd.
†† 東京工業大学大学院情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology

イム性を保障しメモリ消費量の少ない組込み機器向けの OS だけでなく，Linux などデスクトップ PC で用いられている OS が利用されている．このように高機能化，高性能化が求められる一方で，消費電力を抑え携帯端末の駆動時間を延ばすことも同時に求められている．

これらの要求に対して，組込み機器向けプロセッサを製造している企業はプロセッサのマルチコア化を図り対処している．本研究で対象とするマルチコア・プロセッサは，同一のプロセッサ・コアを 1 つのチップに複数集積したものではなく，ヘテロジーニアスなプロセッサ・コアをチップに集積したものである．このようなチップには，組込み機器で広く利用されている CPU コア，マルチメディア処理用の信号処理を得意とする DSP コア，また動画などイメージング用のアクセラレータを組み合わせたものなどがある．

このヘテロジーニアスなマルチコア・プロセッサの利用には次のような 4 つの特徴がある．

- 高機能なソフトウェアの実行
- マルチメディア処理のサポート
- 既存のソフトウェア資源の活用
- 低消費電力

汎用で高性能な組込み機器向け CPU コアを集積することにより Linux などデスクトップ PC で用いられている OS を実行できる．これにより，デスクトップ PC の環境で利用されている高機能なソフトウェアを移植し，組込み機器で利用することが可能になる．そして，DSP など信号処理を得意とするプロセッサ・コアを集積することによりマルチメディア処理を効率的に行うことができる．このようなプロセッサ・コアには汎用品を利用することが多いので，既存のソフトウェア資産を有効に活用することができる．また，対象にする処理に特化したプロセッサ・コアを利用し，それぞれのプロセッサ・コアの動作周波数を低く抑えるので，消費電力を低く抑えることができる．

このようなマルチコア・プロセッサを利用した携帯電話や PDA などの製品が数多く生産されている．これまでは携帯端末で利用されるハードウェアや OS，またソフトウェア開発環境が一般に公開されることはなく，一般プログラマはそれらに触れることのできないクローズドなものだった．しかし最近では，OS にオープンソースの Linux を使用した製品もあり，マルチコア・プロセッサのソフトウェア開発環境が一般に公開されているものもある．しかし，そのような開発環境は製品開発を専門に行う開発者が利用する煩雑なものか，作成できるソフトウェアが限定される i アプリなどの開発環境だけだった．そこで我々は，このように広く使われ始めている組込み向けマルチコア・プロセッサと Linux を組み合わせた環境において，一般プログラマが煩雑な開発工程をふまずに DSP の機能を利用し，これまで十分に利用できなかったプロセッサの性能を簡単に利用できるソフトウェア開発環境について議論する．

本論文は以下のように構成される．2 章では，携帯端末など組込み機器向けの既存ソフトウェア開発環境の問題点に関して述べる．3 章では，本論文に関連する研究を示す．4 章では，OMAP プロセッサで利用されるソフトウェアやハードウェアの特徴などを説明する．5 章では，実際の OMAP プロセッサを用いた製品開発について解説する．6 章では，OMAP プロセッサにおけるソフトウェア開発環境に関する我々の方針について述べ，ソフトウェア開発環境の要件を再検討する．そして 7 章では，我々の提案するソフトウェア開発環境の設計方針を説明する．8 章では，今回実装したプロトタイプ・システムについて述べる．9 章では，プロトタイプ・システムを用いて行った評価実験の結果を述べ考察する．最後に 10 章で本論文の結論と将来の課題を述べる．組込み分野のソフトウェア開発について必ずしも広く一般に知られているわけではないので，本論文ではやや詳しく解説する．

2. 既存ソフトウェア開発環境の問題点

我々は既存のマルチコア・プロセッサ用ソフトウェア開発環境には次の 2 つの問題点があると考える．

- クロス開発に特有の開発工程の煩雑さ
- 低レベルの通信プリミティブへの依存

まず，マルチコア・プロセッサにおいては，基本的にチップ上に集積されているプロセッサ・コアの種類に応じて該当するクロス開発環境を準備する必要がある．これは PC などのホスト・マシン上にターゲット・プロセッサ用のソフトウェア開発環境を構築したものである．またデバッグ作業は，クロス開発環境とターゲット・プロセッサ環境の両方を使う必要がある．CPU と DSP を集積したマルチコア・プロセッサにおける一般的なソフトウェアの開発工程は，CPU 用と DSP 用それぞれのクロス開発環境でソフトウェアのビルドを行い，各実行モジュールを該当するプロセッサ・コアへダウンロードしてソフトウェアの動作確認を行う．動作確認によりバグが見つかれば，それぞれのクロス開発環境でソフトウェアの該当箇所の修正をして再度動作確認を行う（**図 1**）．また IEEE で標準化された JTAG [5] と呼ばれる集積回路のバウンダリスキャン

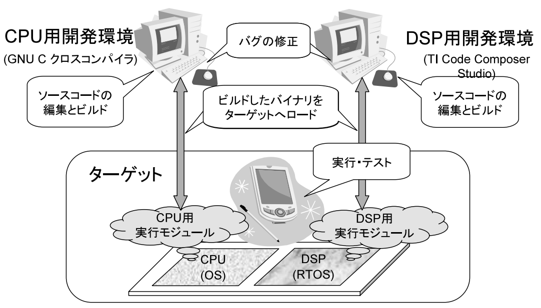


図 1 既存のソフトウェア開発環境の例
Fig. 1 An example of existing software development environment.

テストの手法がある．これは集積回路内の回路をデイジーチェーン接続し，集積回路の外部から集積されている個々の回路の挙動をテストする手法である．そして，JTAG をサポートするプロセッサにおける組込みソフトウェアのデバッグのために，インサーキット・エミュレータ（ICE: In-circuit emulator）と呼ばれるプロセッサの機能をエミュレートするハードウェアの 1 つに JTAG ICE がある．この JTAG ICE を用いることにより，ターゲット・プロセッサの動作をハードウェア・レベルで制御することができるので，ソフトウェアのデバッグ作業に利用することができる．しかし，一般に JTAG のインタフェースはハードウェアのデバッグ用に実装されるので，ハードウェア開発の終盤には実装されなくなり，利用できなくなることが多い．

また，マルチコア・プロセッサでは共有メモリ，専用バス，メールボックスなどを介してプロセッサ間のハードウェア・レベルの通信機能を提供している．通常，このようなプロセッサ間通信機能をアプリケーション・ソフトウェアに提供するためにプロセッサ間通信用ミドルウェアが提供されている．しかし，このような組込みシステムのプロセッサ間通信用ミドルウェアは，リアルタイム処理を実現するためにオーバヘッドの少ない低レベルの通信プリミティブしか提供しない．そのため，プロセッサ間通信をともなうアプリケーション・ソフトウェアのプログラミング・スタイルに悪影響を及ぼす．

ここで，CPU から DSP を利用する場合の例を示す．CPU 側に実装されるアプリケーション・ソフトウェアは DSP 側に実装されている機能と，直感的に関連の乏しい低レベルの通信プリミティブ（e.g. read(), write()）を介して，DSP 側のソフトウェアと通信を行うことにより DSP の提供する機能を利用することになる．また，DSP 側のソフトウェア構造はミドルウェアの規定するプログラミング・スタイルに依存し，プ

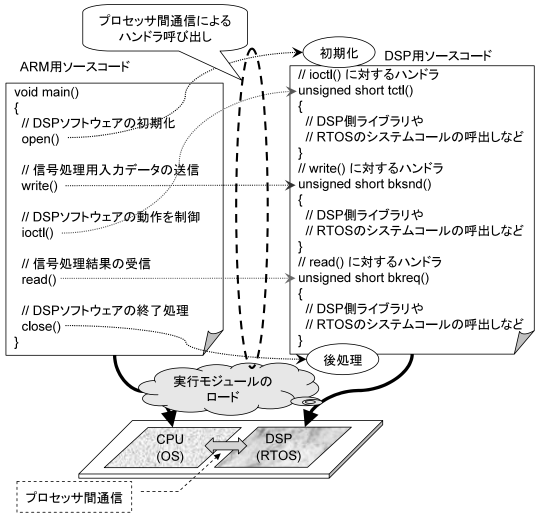


図 2 既存開発環境におけるプログラミング・スタイルの概念図
Fig. 2 Conceptual diagram of the existing programming style.

ロセッサ間通信を実現するためのハンドラ内部（e.g. CPU 側の read() や write() 呼び出しに対応した DSP 側のハンドラ内部）に DSP で提供するサービスの実装を行う必要がある（**図 2**）．

既存の組込み向けマルチコア・プロセッサのソフトウェア開発環境は，企業の開発者が製品の開発にも利用する専門的な開発環境である．しかし，既存の開発環境には組込みシステム特有の開発工程の煩雑さをともなう．また，組込み向けマルチコア・プロセッサ上に実装するソフトウェアは，ここで説明したプロセッサ間通信用ミドルウェアによって規定されるプログラミング・スタイルにより悪影響を受ける．たとえば，RPC であればリモートに実装された関数を一般的な関数呼び出しで利用できるのに対し，以下に示すようにシステムコールを用いて DSP 側のソフトウェアの初期化やハンドラの起動を行わなければならない．

open() ：目的の DSP 側ソフトウェアの初期化
read() ：read に対応するハンドラの起動
write() ：write に対応するハンドラの起動
ioctl() ：ioctl に対応するハンドラの起動
mmap() ：CPU-DSP 間の共有メモリのマップ
close() ：DSP 側ソフトウェアの終了処理

このように，DSP の機能を利用するのに広く利用されている自然なプログラミング・スタイルを適用することができず，低レベルの通信プリミティブによるプロセッサ間通信を意識したプログラミング・スタイルになってしまう．以上の理由から，一般プログラマが既存開発環境を利用することは簡単ではない．

3. 関連研究

本章では，組込みシステムにおけるクロス開発支援のための環境に関連する研究を紹介する．

3.1 開発工程の支援

クロス開発の開発工程の煩雑さを改善するためには次の 3 つのアプローチが考えられる．

- クロス開発のターンアラウンド・タイムの短縮
- ターゲット環境でのテスト，デバッグ作業を削減
- ホスト・マシン環境での作業を削減

ここでは，それぞれのアプローチに関連する研究を紹介する．

Scratchbox[1] は組込み Linux のアプリケーション開発を簡単にするために設計されたクロス開発用ツールキットである．ホスト・マシン上で chroot コマンドを使用しターゲット・ハードウェア用のサンドボックスを実現する．サンドボックス内ではターゲット用のエミュレータを使用することでターゲット用バイナリを実行できる（**図 3**）．そのため，Configure スクリプトを利用するようなソフトウェアのビルドもホスト・マシン上で実行することができる．このとき，コンパイラだけはホスト・マシン上でクロスコンパイラを実行するので，エミュレーションによる実行時のオーバヘッドはない．また，ターゲットのエミュレータの代わりに Scratchbox Remote Shell（sbrsh）を利用することにより，サンドボックスを実現しながらホスト・マシンからビルドしたソフトウェアを実際のターゲット・ハードウェア上で実行させ動作確認することが可能になり，クロス開発のターンアラウンド・タイムを短縮できる．ここで，ホスト・マシンとターゲット・ハードウェアは互いにネットワーク接続されていて NFS によりファイルシステムを共有している必要がある．Scratchbox はシングルコア環境を対象としているが，基本的なアイデア自体はマルチコア環境にも適用可能である．マルチコア環境に適用するためには，chroot サンドボックスを複数構築すればよいだろう．しかし，高度なスキルを仮定した企業内開発者向けの環境が Scratchbox 内で稼動したとしても，一般プログラマが利用するには依然敷居が高いという問題は残される．

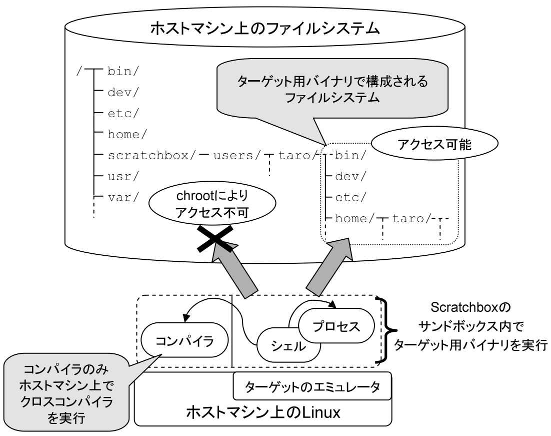


図 3 Scratchbox の概念図
Fig. 3 Conceptual diagram of scratchbox.

QEMU[2] は高速に動作するプロセッサ・エミュレーション環境である．ターゲット・プロセッサ用のバイナリを，実行時にホスト・マシンのネイティブ・コードへ変換することにより高速化を図っている．QEMU には，OS を動作させられるフルシステム・エミュレーションモードと，Linux のユーザプロセスを動作させられるユーザモード・エミュレーションの 2 つのモードをサポートしている．また，前述の Scratchbox と組み合わせることにより，実際のターゲット・ハードウェアで実行するための sbrsh の代わりに，QEMU のエミュレーション環境を利用してホスト・マシン上でターゲット・プロセッサ用のバイナリを実行することも可能にする．したがって，ターゲット・ハードウェアを使うテストやデバッグ作業を削減し，ホスト・マシン上でテスト，デバッグ作業を行うことを可能にする．QEMU もシングルコア環境を対象としているが，基本的なアイデア自体はマルチコア環境にも適用可能である．マルチコア環境に適用するためには，複数のエミュレータが相互作用可能な環境を構築すればよいだろう．しかし，Scratchbox の場合と同様に高度なスキルを仮定した企業内開発者向けの環境が QEMU 上で稼動したとしても，一般プログラマが利用するには依然敷居が高いという問題は残される．

HotPaw Basic[3] は，ターゲット上でプログラミング環境を提供するものの 1 つである．これは Palm OS 上で BASIC のプログラミングを作成し実行できる環境を提供する．HotPaw Basic は豊富なビルトイン関数とコマンドをサポートしているので，PDA プログラミングの初心者でも簡単に GUI を持つアプリケーションを作成することができる．このように，HotPaw Basic は Palm OS の動作するターゲット上で BASIC のプログラミングができるので，ホスト・マシンの開発環境を利用する必要がなく開発工程が単純化される．HotPaw BASIC は高度なスキルを要求しない代わりに，記述能力には限界がある．たとえば，マルチメディア・ストリームの同期などのリアルタイム処理をともなう処理の記述には向いていない．

3.2 プロセッサ間通信の支援

CORBA[4] は，分散システム環境でメッセージ通信を支援するミドルウェアである．CORBA は ORB（Object Request Broker），CORBA ファシリティ，CORBA サービスから構成される．CORBA は，分散したオブジェクトのインタフェースを IDL（Interface Definition Language）と呼ばれるインタフェース定義言語で定義する．インタフェース定義に基づいてスタブとスケルトンを自動生成することにより，ヘテロジーニアスな環境での通信を容易に記述可能とする方式は，RPC やウェブサービスなどでも利用されている．本研究で対象にしているアーキテクチャはヘテロジーニアスで，プロセッサ・コア間の物理的距離が近いマルチコア・プロセッサである．したがって，CORBA が仮定している分散環境より，相対的に通信速度が速く，プロセッサ間でメモリの共有もできるので，すべてのデータをコピーして関数のパラメータとして渡す方式は性能上最適ではないと考える．

以上のように，クロス開発工程の煩雑さの削減や通信プリミティブの高レベル化に関する研究はさかんに行われている．しかし，携帯機器向けのヘテロジーニアスなマルチコア・プロセッサにおいて，DSP でのリアルタイム処理を実現しつつ，システムコールによる DSP 側のハンドラを呼び出すインタフェースに代えて，より自然な手続き呼び出しのセマンティクスを採用し，一般プログラマにも利用しやすいプログラミング環境はまだ提案されていない．

4. OMAP プロセッサとは

現在，多様なマルチコア・プロセッサが製造され幅広い分野の製品で使われている．その中でも，Texas Instruments（TI）の製造する OMAP プロセッサは NTT ドコモが提供する FOMA サービス用端末として製造されている携帯電話で広く採用されている．FOMA 端末を製造している NEC と Panasonic は，自社で製造するすべての携帯電話で OS として Linux を採用している．また，Nokia はインターネット・タブレット端末に OMAP プロセッサと Linux を採用している．Linux OMAP のオープンソース・コミュニティで OMAP 用の Linux のディストリビューションは管理されている．その Linux OMAP のディストリビューションにはプロセッサ間通信用ミドルウェアとして DSP Gateway が採用されている．本研究では携帯端末で広く採用されている OMAP プロセッサに注目する．OMAP プロセッサでは，オープンソースの OS である Linux とプロセッサ間通信用ミドルウェアである DSP Gateway が利用可能であり，また OMAP プロセッサの評価ボードが入手可能であるなど，研究対象としての環境が整っている．

OMAP プロセッサにはいくつかのモデルがあるが，携帯端末では CPU として ARM コアと DSP として C55x と呼ばれる TI 製の DSP コアを集積したモデルが多く利用されている．CPU と DSP 間のプロセッサ間通信にはメールボックスと呼ばれるハードウェアの機能が利用できる．また，プロセッサの外部に接続する SDRAM や DSP の内部 RAM も双方のコアから MMU を利用してアクセスできるので，それらメモリを介して大きなデータを交換することも可能である．

OMAP プロセッサで利用できる CPU 側の OS は，Symbian，Linux，Windows CE などがある．また，DSP 側では TI 製のリアルタイム OS である DSP/BIOS が利用可能である．プロセッサ間通信用のミドルウェアもいくつか利用可能である．既存の OMAP プロセッサ用開発環境では，ホスト・マシン上で動作する CPU 用と DSP 用のクロスコンパイラが提供されている．これらを用いて CPU 側で動作するすべてのソフトウェアを開発することができる．また，DSP 側で動作する DSP/BIOS のアプリケーションや信号処理用プログラムの開発も行うことができる．本研究では，CPU 側の OS として Linux，DSP 側の OS として DSP/BIOS を使用する．また，プロセッサ間通信用ミドルウェアは Linux OMAP のディストリビューションに含まれている DSP Gateway を使用する．この DSP Gateway は Nokia で開発されていて SourceForge で公開されている．

一方で，OMAP プロセッサを利用するうえで 2 つの制約がある．

- DSP に関する限られた情報
- プロセッサ・コアの組合せによる困難

DSP 側のソフトウェア・モジュールは COFF フォーマットで保存されている．TI の提供する OMAP プロセッサ用の開発環境では，一般的な COFF フォーマットに加え TI が独自に拡張したフォーマットを一部で使用していて，その情報は現状では公開されていない．したがって，TI の拡張した COFF フォーマットの OS やライブラリなどのソフトウェア・モジュールとリンクするためには，この未公開の情報を解決しなければならない．さらに，OMAP の開発環境のバージョンが上がると，この未公開情報の内容も変化する傾向にある．

Intel のマルチコアや IBM 等の Cell プロセッサなどの一般的なマルチコア・プロセッサとは異なり，OMAP

ではインテグレーションを考慮せず別々に設計された，ARM コアと TI 製 DSP コアが組み合わされる．これにより，既存のソフトウェア資産を有効に活用できる一方で，プロセッサ・コア間にはエンディアンやアドレッシングなど基本的な相違がある．通常 ARM コアはリトル・エンディアンで動作し，DSP コアはビッグ・エンディアンで動作するので，ARM-DSP 間でデータを交換する場合は注意が必要である．ただし，OMAP プロセッサのモデルによっては，このエンディアン変換をサポートする機能が実装されたプロセッサもある．加えて，ARM では命令領域・データ領域ともにバイト単位でアドレッシングが可能であるのに対し，DSP では命令領域へはバイト単位のアドレッシングで，データ領域へはワード単位のアドレッシングになっている．通常 ARM のデータ領域は 4 バイトにアラインされているのでワード，ダブル・ワードのデータを交換する際に問題が起こることは少ないが，連続した 1 バイトのデータを交換するときなどは，このアドレッシングの違いに注意が必要である．

5. OMAP プロセッサを用いた製品開発

ここでは，OMAP プロセッサを用いた製品開発の経験から得られた知見を基に，実際の製品開発において，次の 3 つの点について解説する．

- プロセッサの利用モデル
- 製品開発の工程
- 既存のデバッグ環境

製品開発の工程については，特にソフトウェアの実装段階に注目し解説を行う．

5.1 プロセッサの利用モデル

携帯端末では，CPU としての ARM コアと信号処理用の DSP コアを集積した OMAP プロセッサが多く採用されている．一般にこのような OMAP プロセッサでは，CPU 側では GUI など汎用的な処理を行い，DSP 側ではその特長を生かした信号処理を行う．その中でも DSP コアは次のような目的で利用されることが多い．

(1) リアルタイム処理の実現

前述のとおり，一般に OMAP プロセッサの CPU 側では汎用 OS が動作し，DSP 側ではリアルタイム OS が動作している．リアルタイム処理を実現する場合，CPU 側の汎用 OS ではリアルタイム性の保障が難しいことから，DSP 側のリアルタイム OS の機能を利用することが多い．この場合，CPU 側からデータを DSP 側へ送り，DSP でリアルタイム処理を行い，DSP から外部デバイスへ出力する．たとえば，オーディオとビデオの同期処理は，DSP で CPU 側から送られてくるデータをデコードし，それらの同期をとり直接その処理結果をフレームバッファやオーディオ・デバイスへ出力する．つまり，リアルタイム処理を実現する場合の多くは，データは CPU から DSP への 1 方向に流れる．

(2) 信号処理用の計算資源

また，DSP は信号処理が得意でありライブラリが充実していることが多いので，DSP の信号処理能力を利用することも多い．この場合，CPU 側からデータを DSP 側へ送り，DSP で信号処理を施し，CPU 側でその処理結果を引き取る．たとえば，画像データの YUV-RGB 変換は，DSP で CPU 側から送られてくる YUV データを RGB へ変換する．その処理結果を再度 CPU 側で加工し出力する．つまり，信号処理用の計算資源として利用する場合は，データは CPU から DSP へ流れ，その後 DSP から CPU へと双方向に流れる．

5.2 製品開発の工程

一般的に製品開発においては複数のチームを構成して各ソフトウェア・コンポーネントの開発にあたる．CPU コアと DSP コアから構成される OMAP プロセッサを用いる場合は次のようなチーム構成が考えられる．

- OS 担当
- ウィンドウ・システム担当
- ペリフェラル・デバイスドライバ担当
- マルチメディア・アプリケーション担当
- DSP ソフトウェア担当
- インテグレーション担当

OMAP プロセッサを用いた製品開発では，各チームが以下のような手順を繰り返し行いながらソフトウェアの開発作業を進める．

(1) 各コア用ソフトウェア・コンポーネントの実装

各チームは要求されるソフトウェアの機能の実装を行い，そのソフトウェアが単体で正しく動作することを確認するためにユニット・テストを行う．また，チーム間に依存関係がある場合もある．この例では，マルチメディア・アプリケーションは DSP ソフトウェアを利用することが想定されるので，DSP ソフトウェアのリリース後に，それらを用いてマルチメディア・アプリケーションのユニット・テストを行うことになる．各チームはユニット・テストにパスしたソフトウェア・コンポーネントをリリースする．

(2) インテグレーション・テスト

全チームからのリリースが揃うと，インテグレーション担当がそれらコンポーネントを 1 つにまとめインテ

グレーション・テストを行う．インテグレーション・テストでは，すべてのソフトウェア・コンポーネントが互いの動作を邪魔することなく正しく動作することを確認する．この時点でバグが確認されると，インテグレーション担当は問題の切り分けを行い，発見したバグを担当するチームを割り当てる．Bugzilla などのバグ追跡システムを利用している場合には，そこに発見したバグについての詳細なレポートを報告する．

(3) **デバッグ作業**

バグの担当として割り当てられたチームは，バグ・レポートを受けてデバッグ作業を行う．バグの修正が次回ソフトウェアのリリースに間に合えば，その修正を含めたソフトウェア・コンポーネントのリリースを行う．

以上の手順を経て，インテグレーション担当から定期的にすべてのソフトウェア・コンポーネントを含み，インテグレーション・テストをパスしたソフトウェア・パッケージがリリースされる．このサイクルを繰り返すことにより，リリースされるソフトウェア・パッケージが含むバグが減少し完成度が高まる．

5.3 デバッグ環境

CPU コアと DSP コアの集積された OMAP プロセッサでは，それぞれのコア上でデバッグ作業を行う必要がある．

CPU 側では，GNU GDB など一般的なデバッガが利用可能である．一方，JTAG ICE などの特別なハードウェアを利用すると，CPU コアと DSP コアの状態の確認や，OS などのデバッグを行うことができる．しかし，DSP 側において JTAG ICE を利用しない場合は，ARM 側へ文字列を出力するための printf のような機能を利用してのデバッグ手法が主になる．

また，CPU 側と DSP 側のソフトウェアが通信しながら動作する状況では，既存の GDB では CPU 側と DSP 側の同時のデバッグに対応していないためデバッグ作業は困難である．一部の JTAG ICE では同時に CPU コアと DSP コアを制御することが可能だが，アプリケーション・レベルのソフトウェアのデバッグに利用することは簡単ではない．

6. OMAP における開発環境の再検討

本章では，2 章で述べた以下の既存開発環境における問題点について検討する．

- ソフトウェア開発工程
- プログラミング・スタイル

そして，我々の考えるソフトウェア開発環境の要求要件について説明する．これらの議論を通じて，OMAP プロセッサにおけるクロス開発環境について再度検討する．

6.1 ソフトウェア開発工程

OMAP は組込み機器向けのプロセッサであり，複数のプロセッサ・コアを集積する各種モデルが製造されている．その中でも携帯端末で現在広く使われている CPU コアと DSP コアを集積したものに注目する．

このようなマルチコア・プロセッサを用いたソフトウェア開発における開発工程の煩雑さは，クロス開発環境を使用しなければならない点と，複数の開発環境を使用しなければならない点とに起因するものと考える．

まず，クロス開発環境について述べる．通常，組込みシステムの開発ではターゲット・ハードウェアとは別のホスト・マシン上にクロス開発環境を準備する必要がある．それは，OMAP を利用する場合も同様である．ソースコードの編集やビルド作業はホスト・マシン上のクロス開発環境で行い，ビルドしたソフトウェアをターゲット・ハードウェア上で実行させ動作確認を行う．バグが発見されれば再びホスト・マシン上のクロス開発環境でソースコードを修正・ビルドして，ターゲット・ハードウェア上で動作確認を行う（図 1）．一般にホスト・マシンとターゲット・ハードウェアは RS232C などシリアルケーブルで接続されており，シリアル端末ソフトウェア（e.g. minicom，kermit）を利用しターゲット・ハードウェアの制御やビルドしたソフトウェアのダウンロードを行う．ターゲット・ハードウェアがネットワーク・インタフェースを搭載している場合は，リモートログイン・プログラム（e.g. ssh, telnet）を利用し高速にターゲット・ハードウェアを制御することもできる．

次に，複数の開発環境を使用する点について述べる．CPU コアとして搭載されている ARM 上で Linux を使用する場合，ソフトウェアのクロス開発環境として GNU から提供されているクロスコンパイラなどを含む ARM 用のツールチェインをホスト・マシン上に準備し利用するのが一般的である．一方，DSP コアとして搭載されている C55x シリーズには TI が供給する統合開発環境の CCS（Code Composer Studio）が利用できる．これは Windows で利用できる開発環境でソースコードの編集，ビルド，JTAG ICE と連携したデバッグ，また GUI ベースでプログラムの実行時の動作解析などを行える．また，Linux 上でソフトウェアのビルドのみ可能な C55x 用コンパイラも提供されている．したがって，ARM 用と DSP 用のクロス開発環境を Linux の動作するホスト・マシン上に準備す

```
int main(int argc, char *argv[])
{
  char *fin = "in.dat";
  char *fcoef = "coef.dat";
  char *fout = "dsp_out.dat";
  float *inbuf;
  float *coef;
  float *outbuf;
  int num = 200;
  int tap = 15;
  int fd;

  fd = open("/dev/dsptask/fir", O_RDWR);

  inbuf = (float *)mmap((void *)0,
                   MMAP_BUFFER_SIZE*sizeof(float),
                   PROT_READ|PROT_WRITE,
                   MAP_SHARED,
                   fd,
                   0);
  coef = inbuf + num;
  outbuf = coef + tap;

  /* Set number of input data */
  ioctl(fd, FIR_NUM_DATA, num);

  /* Set number of coefficient constants */
  ioctl(fd, FIR_NUM_TAPS, tap);

  /* Read input data */
  read_data_array(fin, inbuf, num);

  /* Read coefficient constants */
  read_data_array(fcoef, coef, tap);

  /* Apply the FIR filter */
  ioctl(fd, FIR_EXEC);

  /* Write calculation results */
  write_data_array(fout, outbuf, num);

  close(fd);

  return 0;
}
```

図 4 ARM 用の既存環境を利用した FIR フィルタ・プログラムの抜粋

Fig. 4 A piece of source code of FIR filter program developed using the existing development environment for ARM.

```
#pragma DATA_SECTION(fir_socket_node, "dspgw_task")
struct dsptask fir_socket_node = {
  TID_MAGIC,      // Task ID
  (String)"fir", // Task Name
                  // Task type: Active Block Send
                  //            Passive Block Receive
  MBCMD_TTYP_BKDM |
  MBCMD_TTYP_BKMD |
  MBCMD_TTYP_ASND,
  fir_bksnd,
  NULL,
  fir_tctl,
  &fir_attr,
  &fir_mmap_info,
  NULL
};

...

// Handler definition for ioctl() system call
static Uns fir_tctl(struct dsptask *task,
                    Uns ctlcmd, Uns *ret, Uns arg)
{
  switch(ctlcmd) {
  case FIR_NUM_DATA:
    num_of_data = arg;
    break;

  case FIR_NUM_TAPS:
    num_of_coef = arg;
    break;

  case FIR_EXEC:
    fir_exec(num_of_coef, num_of_data);
    *ret = 0;
    break;

  default:
    break;
  }
  return 0;
}
```

図 5 DSP 用の既存環境を利用した FIR フィルタ・プログラムの抜粋

Fig. 5 A piece of source code of FIR filter program developed using the existing development environment for DSP.

ることは可能であるが，ソフトウェアのビルドに使用するクロスコンパイラなどのツールチェインは ARM 用と DSP 用で別々である．

以上のように CPU コアと DSP コアを集積する OMAP を使用する場合，ホスト・マシン上のシリアル端末ソフトウェアなどを利用し，ビルドしたソフトウェアをターゲット・ハードウェアへダウンロードして実行させなければならない．そして，ホスト・マシン上で ARM 用と DSP 用の 2 種類のクロス開発環境を使用しソフトウェアのビルドを行わなければならない．

6.2 プログラミング・スタイル

OMAP の ARM 上で Linux を利用する場合，プロセッサ間通信用ミドルウェアとして Linux OMAP のディストリビューションに含まれている DSP Gateway を使用することになる．DSP Gateway を使用するためには，ARM 側と DSP 側のソフトウェアは一定のプログラミング・スタイルに従う必要がある．まず，ARM 側のソフトウェアは DSP 側に実装されるソフトウェアの機能を利用するために，DSP 上に実装されるソフトウェアごとに生成されるデバイスファイルに対して open，read，write などのシステムコールを発行する．**図 4** では，我々がサンプルとして実装した FIR フィルタ・プログラムのソースコード（ARM 側）の抜粋を示す．この例では open，mmap，ioctl システムコールを介して，DSP 側に実装された FIR フィルタ機能を利用している．

次に，DSP 側のソフトウェアは DSP Gateway のタスクを定義し実装することにより ARM 側にデバイスファイルが生成され，Linux のアプリケーション・プログラムから当該デバイスファイルを介して DSP 側に実装するソフトウェアにアクセスが可能になる．この DSP Gateway のタスクを定義するためには，いくつかのデータ構造体を定義する必要がある．また，タスクの属性に応じて ARM 側の read，write，ioctl システムコールに対応するハンドラ関数や mmap システムコールでマップされるデータ領域などを定義する必要がある．**図 5** では，サンプルとして実装した

```
// Internal function calling DSP library
void fir_exec(int tap, int num)
{
  DATA *pinq15;
  DATA *pcoefq15;
  DATA *poutq15;
  DATA *pdbuf;

  float *coef;
  float *inbuf;
  float *outbuf;

  inbuf = fir_mmap_buffer;
  coef = inbuf + num;
  outbuf = coef + tap;

  /* Allocate working memory in DSP internal memory */
  pinq15 = malloc(sizeof(float) * num);
  poutq15 = malloc(sizeof(float) * num);
  pcoefq15 = malloc(sizeof(float) * tap);
  pdbuf = malloc(sizeof(float) * (tap + 2));

  /* Adjust arrangement on input data */
  byteorder_conversion(coef, coef, tap);
  byteorder_conversion(inbuf, inbuf, num);

  /* Convert float format into Q15 format */
  fltoq15(coef, pcoefq15, tap);
  fltoq15(inbuf, pinq15, num);

  /* Invoke the FIR library function */
  fir2(pinq15, pcoefq15, poutq15, pdbuf, num, tap);

  /* Convert Q15 format into float format */
  q15tofl(poutq15, outbuf, num);

  /* Adjust arrangement on output data */
  byteorder_conversion(outbuf, outbuf, num);

  free(pinq15);
  free(poutq15);
  free(pcoefq15);
  free(pdbuf);

  return;
}
```

図 6 DSP 用の既存環境を利用した DSP ライブラリを呼び出す関数

Fig. 6 Source code of a function calling DSP library developed using the existing development environment for DSP.

FIR フィルタ・プログラムのソースコード（DSP 側）の抜粋を示す．これは DSP Gateway で規定されたタスクを定義するためのデータ構造の定義と ARM 側で発行される ioctl システムコールに対するハンドラ関数の定義を示す．今回の実装では，ARM 側からこのハンドラ関数を介して FIR フィルタに関するパラメータを DSP 側で受け取り，さらに**図 6** に示す DSP 上に実装されている FIR フィルタの機能を実現するための関数（i.e. fir_exec）を呼び出している．これは DSP 上にあるいくつかのライブラリ関数をグルーイングしている．

6.3 ソフトウェア開発環境の要求要件

本研究ではこれまでに述べた問題点に対して組込み向けプロセッサにおけるソフトウェアの開発工程の煩雑さを低減させ，低レベルの通信プリミティブを利用するプロセッサ間通信に依存しないプログラミング・スタイルを提供し，加えて既存のソフトウェア資産を有効活用することを目指す．また，エンドユーザの中でも一般のプログラミングもできる人が，DSP 固有の特殊機能を少しだけ使う程度のアプリケーションの作成を可能にする．この章では，我々の目標とするソフトウェア開発環境の要求要件を定義する．

(1) **開発工程の改善**

これまでの組込み向けマルチコア・プロセッサにおける開発工程を改善するために，同じファイル中に CPU 側のソースコードと DSP 側のソースコードを混在させた記述を可能にし，ホスト・マシン上のクロス開発環境でのビルド作業をしなくても，ソースコードを直接ターゲット・ハードウェア上で実行可能にする．

小型 LCD など表示領域の限られた表示デバイスを使用した携帯端末上で上記すべての工程を行うことも可能であるが，ソースコードの作成・編集作業はホスト・マシン上で行うことも可能にする．

(2) **ソフトウェア資産の活用**

既成のプロセッサ・コアを組み合わせたマルチコア・プロセッサでは，既存のソフトウェア資産が利用可能である．CPU コアと DSP コアから構成される OMAP プロセッサの場合，次のようなライブラリがすでに存在することが考えられ，それらとの連携を実現する必要がある．

- CPU 側の GUI ライブラリ
- DSP 側の信号処理用ライブラリ

つまり，GUI ライブラリや信号処理用ライブラリなど既存のライブラリで提供される機能そのものは開発の対象とせず，それらネイティブに実装されているライブラリ関数を目的の処理に合わせグルーイングして有効に利用できるソフトウェア開発環境の構築を目指す．

(3) **デバッグ手法**

一般的なデバッガには，実行中のプログラム内部の様子を観察したり，プログラムが異常終了したときにプログラム内部の状況を確認したりできる機能がある．本研究では，特別なハードウェアなしに CPU 側と DSP 側のプログラムをソースコード・レベルでデバッグし，その内部の状態を観察できる機能を提供することを検討する．

7. 提案するソフトウェア開発環境の設計

この章では，我々の提案するソフトウェア開発環境の設計方針について述べる．

- 提案の背景
- 対象とする利用者とアプリケーション
- 提案環境の特徴
- 言語仕様の定義
- ソフトウェア開発環境の実現方法

- 処理系の設計

以降の節では，設計方針に関する上記 6 つの点について解説する．

7.1 提案の背景

高い処理能力を持つ ARM+DSP のようなマルチコア・プロセッサを搭載した携帯電話や PDA が多く生産されている．しかし，一般プログラマには i アプリなど限定されたアプリケーションの開発環境だけが公開されていた．このため，一般プログラマには携帯端末の性能を十分に生かすソフトウェアを開発することが不可能だった．一方で，いくつかの携帯端末では OS にオープンソースの Linux を使用した製品もあり，一般プログラマでもマルチコア・プロセッサ上のすべてのソフトウェア開発が可能になってきた．ただし，このようなソフトウェア開発環境は企業の開発者が製品開発に使うものなので，一般プログラマには使いこなすのが難しい．

7.2 対象とする利用者とアプリケーション

既存の開発環境として ARM 側にはクロス開発用 gcc が利用可能であり，DSP 側には Linux 用ツールチェインと Windows 用の統合開発環境が TI から供給されている．この ARM 側の開発環境は，通常の Linux のプログラミング環境と同等なので一般プログラマでも利用しやすいと考えられる．一方，DSP 側の開発環境では，DSP で動作する OS である DSP/BIOS のアプリケーション・プログラムの開発が可能であることに加え，DSP の特殊機能を利用し高速に信号処理を行うためのアセンブラ・プログラミングも可能である．このような特殊機能は，DSP プログラミングを専門に行う開発者には必須であるが，一般プログラマには使いこなすのが難しい．

ここでソフトウェア開発環境の開発対象とプログラマに要求される熟練度の関係を**図 7** に示す．既存のソフトウェア開発環境は主に製品開発を行う専門家が利用し，OS やコーデックなどのハードウェアに近いソフトウェアからアプリケーションまで，すべてのソフトウェアを開発の対象にしている．また，熟練度の低い一般のプログラマでも利用できる携帯端末用の開発環境である i アプリの開発環境は，DSP の機能を直接使うようなアプリケーションの開発は行えない．

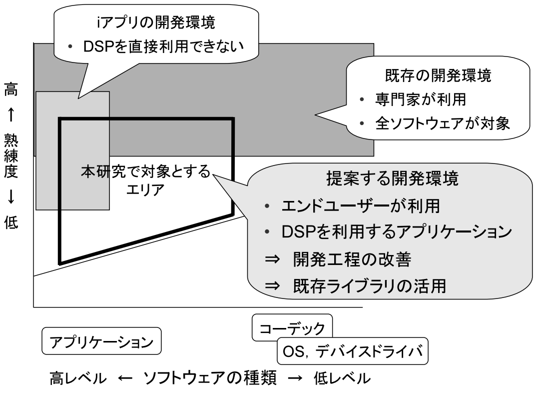


図 7 ソフトウェア開発環境の開発対象と利用者の熟練度
Fig. 7 Relationship diagram between development target and required user's skill.

そこで，我々の提案するソフトウェア開発環境では，利用者として Linux のプログラミングの知識を持つ一般プログラマを対象とし，ソフトウェア製品の開発者は対象としない．また，DSP 固有の特殊機能を利用するようなアセンブラ・プログラミングを行わず，DSP 機能は既存の DSP 上のライブラリを呼び出すことにより利用するものとする．そのため，利用者は DSP/BIOS のシステムコールなど DSP 側のいくつかの基礎的な知識を習得するものとする．このように，提案環境では既存の信号処理用ライブラリが利用可能になり，DSP 側のリアルタイム OS の基本機能を利用可能になる．その結果，一般プログラマは，写真など編集や解析するアプリケーションを既存の信号処理ライブラリを利用して作成できるようになる．また，音声と動画の同期制御が必要になるマルチメディア・プレーヤなども，既存の信号処理ライブラリとリアルタイム OS の機能を利用することにより作成することができるようになる．しかし，提案環境では DSP 側で動作する信号処理ライブラリや DSP/BIOS のデバイスドライバを作成することはできない．これらを作成するためには既存の開発環境を利用する必要がある．

7.3 提案環境の特徴

我々は組込みシステム向けマルチコア・プロセッサのためのソフトウェア開発環境としてインタプリタを用いるランタイム環境を提案する．本研究では，マルチコア・プロセッサとして TI 製 OMAP プロセッサを対象とする．このランタイム環境には次にあげる 3 つの特徴がある．

(1) 統一プログラミング・スタイルの適用

プログラミング言語レベルでプロセッサ間通信をサポートすることにより，低レベルの通信プリミティブを利用するプロセッサ間通信に非依存なプログラミング・スタイルの適用を可能にする．これにより，統一された言語で ARM と DSP の双方のアプリケーション・プログラムを記述することができ，関数単位で ARM と DSP のどちらで実行されるかを指定できる．ARM 側から DSP 側で実行される関数を呼び出す場合，通常の関数呼び出しの記述をプロセッサ間通信によるリモート関数呼び出しに変換することでプロセッサをま

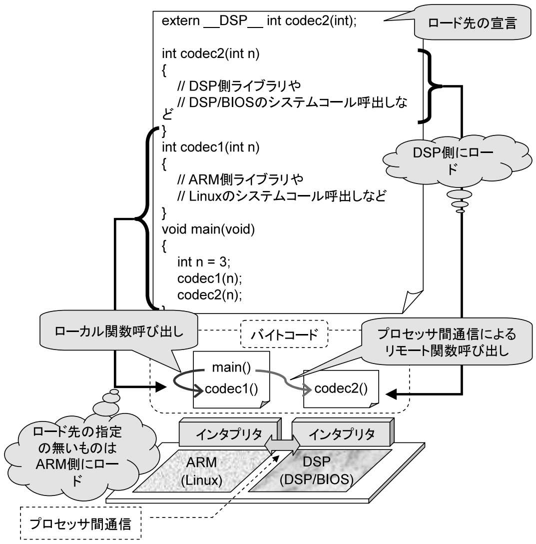


図 8 統一プログラミング・スタイルの概念図
Fig. 8 Conceptual diagram of the proposed unified programming style.

たがって呼び出せる（**図 8**）．この結果，本来実現しようとするアルゴリズムや機能の記述だけに専念できる．

（2） **ソフトウェア開発環境としてのランタイム環境**
ターゲット・ハードウェア上で動作するランタイム環境としてソフトウェア開発環境を提供することにより，クロス開発環境にある開発工程の煩雑さを改善することができると期待される．また，ターゲット・プロセッサに関する詳細な情報がすべて開示されていなくても，ランタイム環境を実装することが可能になる．

（3） **既存ソフトウェア資産の有効活用**
GUI を実現する X Window や GTK+など多くのオープンソースのソフトウェアが利用可能である．また，プロセッサ・ベンダが対象となるプロセッサに最適化された信号処理用ライブラリの実装を提供していることが多い．本研究ではこれら既存のソフトウェア資産を有効に利用し，組み合わせてソフトウェアを実装することを可能にする．

次に提案するランタイム環境の言語仕様とその処理系の設計について述べる．

7.4 言語仕様の定義

本研究では次のように言語仕様を定める．

プログラミング言語 C 言語は，組込みシステムでは現在も広く使われており，一般プログラマにも利用される頻度が高い．そのため，多くの利用者にとって C 言語の文法は理解しやすいと考える．そこで，我々の提案するランタイム環境ではプログラミング言語として C 言語を使用する．

プロセッサ間通信 これまでは，CPU 側から DSP 側に実装されている機能を利用するために，低レベルの通信プリミティブを介したプロセッサ間通信を利用していた．そのため，ソースコードの記述は直感的に分かりやすいとはいえず，ソースコードの可読性は高くなかった．そこで，一般プログラマにもなじみやすい RPC のように，CPU 側から直接 DSP 側の機能を関数呼び出しの形式で利用可能にする．CPU 側から DSP 側の関数呼び出しにより，内部的にプロセッサ間通信が行われ，関数のパラメータや戻り値が交換される．

しかし，我々の対象としているのはプロセッサ・コア間の物理的距離が近いマルチコア・プロセッサなので，一般の RPC や CORBA が仮定している分散環境より，相対的に通信速度が速く，プロセッサ間でメモリの共有もできる．すべてのデータをコピーして関数のパラメータとして渡す方式は性能上最適ではない．そこで，プロセッサ間のデータ転送を効率化する仕組みとして，ソースコード内のグローバル変数をそれぞれのプロセッサからアクセス可能な共有メモリ上に配置する．これにより，CPU と DSP 間で大きなデータをやりとりする機能を効率的に実装することができる．この共有メモリは CPU 側と DSP 側のライブラリからも直接アクセスできる領域に割り当てる．ただし，エンディアンさえ異なるマルチコア間で共有されるため，ヘテロジーニアス共有メモリ[6),7)] になる点には注意が必要である．

ソースコード ソースコードの記述方法が DSP 側においても，特別な作法などなく一般的な C 言語の文法だけで記述できるので，一般プログラマにも DSP 側のプログラミングがしやすくなる．また，CPU 側と DSP 側のプログラムを同一のファイル内に記述できるので小規模なプログラムの場合，ソースコードの見通しが良くなる．

そして，利用者は，自分で定義した関数が CPU または DSP のどちらで実行されるかを，その関数宣言時に静的に指定することができる．今回はプロセッサに対して各関数の自動的な割振り，実行するプロセッサの動的な変更はサポートしない．

ライブラリ 既存のプロセッサ・コアを組み合わせたマルチコアにおいては，各コア用の既存のソフトウェア資産が豊富にある．そこで，CPU 側は Linux 上にあるライブラリを，DSP 側は DSP/BIOS 上にある信号処理ライブラリも利用可能にする．プログラムからは各ライブラリ関数を通常の関数呼び出しの形式で利用できる．

表 1　提案環境に対する実現方法の比較
Table 1　Comparison of implementation methods for the proposed environment.

	(1) スタブコードの導入	(2) ランタイムのコンパイル	(3) インタプリテーション
統一的なプログラミング・スタイル	○	○	○
既存ソフトウェア資産を活用	○	○	○
クロス環境の開発工程を改善	×	○	○
組込み環境における実装可能性	○	△	○
ソフトウェアの実行速度	○	○	△

一般に GUI プログラミングは，発生するイベントに対するコールバックを登録するようなプログラミング・スタイルが適用される．CPU 側で GTK+など GUI のフレームワークを利用可能にするため，外部のプログラムから呼び出し可能なコールバックの記述と実行を可能にする．

7.5　ソフトウェア開発環境の実現方法

次に本研究で提案するマルチコア・プロセッサのソフトウェア開発環境の実現方法について議論する．先にあげた既存のソフトウェア開発環境の問題点をふまえ，3 つの実現方法を比較する．

(1)　**スタブコードの導入**

RPC のようにスタブコードとスケルトンコードを生成する機構を導入し，低レベルの通信プリミティブを利用するプロセッサ間通信に依存したプログラミング・スタイルを隠蔽する．これにより，アプリケーション・ソフトウェアが他のプロセッサ・コア上に実装された機能を直感的に利用するプログラミング・スタイルの適用を可能にする．この場合，クロス開発の煩雑さとコアごとに異なる開発環境を使用する煩雑さは変わらず残してしまう．

(2)　**ターゲット上でのコンパイレーション**

ターゲット・ハードウェア上でソースコードから実行モジュールの生成を行うことにより，クロス開発環境は不必要になる．したがって，クロス開発における開発工程の煩雑さは解消される．また，スタブコードとスケルトンコードを生成する機構もあわせて導入すれば，アプリケーション・ソフトウェアが他のプロセッサ・コア上に実装された機能を直感的に利用するプログラミング･スタイルの適用を可能にする．しかし，利用可能な資源の限られた組込み向けプロセッサ上で既存のコンパイラを実行することは難しい．また，新たにコンパイラを開発する場合，実装に必要なプロセッサに関する詳細な情報がすべて公開されている保証はない．OMAP プロセッサにおいては，DSP に関する一部の情報が公開されていない．

(3)　**インタプリテーション**

ターゲット・ハードウェア上でソースコードを実行することが可能なので，アプリケーション・プログラマはクロス開発環境を使用する必要がない．したがって，ターゲット・ハードウェア上だけでソフトウェア開発が完結するので，クロス開発における開発工程の煩雑さは解消される．また，インタプリットする言語の仕様にプロセッサ間通信を取り込むことにより，低レベルの通信プリミティブを利用するプロセッサ間通信に非依存なプログラミング・スタイルを適用することが可能になる．ターゲット・プロセッサに関する詳細な情報が非公開であってもインタプリタの実装は可能である．しかし，インタプリテーションは実行時のオーバヘッドが他の方法に比べ大きくなるので，実行速度が問題になる可能性がある．

各実現方法の特徴を**表 1** に示す．これらの特徴を詳しく見ると，(1) のスタブコードの導入は，クロス開発環境の開発工程を改善する効果がないので，今回の提案するソフトウェア開発環境の要求要件を満たさない．(2) のターゲット上でのコンパイレーションは，対象のソフトウェアをコンパイルし機械語まで生成するので，既存の開発環境で構築したソフトウェアに近い高速な実行速度が期待できる．このため，開発環境の利用者にとってメリットは大きいが，プロセッサに関するすべての情報が公開されるか分からない DSP などの専用プロセッサ・コアにおいては，そのような開発環境の実装可能性が問題になる．また，今回対象にしているような高性能な DSP の場合，既存のリアルタイム OS やライブラリを有効に活用する必要がある．しかし，それら既存の COFF フォーマットのソフトウェア・モジュールとコンパイルしたモジュールとをリンクして利用するためには，TI の拡張した COFF フォーマットの未公開情報を解決する必要があり難しい．このように，情報が完全には公開されないかもしれない状態で，マルチターゲットのコンパイラのメンテナンスを続けるのは，開発者にとって負担が大きい．(3) のインタプリテーションは，その性質上実行時のオーバヘッドが大きいことが予想される．これは開発環境の利用者にとってはデメリットだが，実行するソフトウェアがライブラリ呼び出しなどをグルーイング

することが主たる目的であれば，インタプリットするソフトウェアのコード量はあまり多くならず，インタプリテーションのオーバヘッドが実行速度に大きな影響は及ぼさないと予想される．一方，開発環境を構築する開発者にとっては，実装時に TI の未公開情報を解決する必要が省かれ，また将来違うプラットフォームへの移植も簡単になるなどメリットが大きい．組込み以外の分野において，インタプリテーションを行うスクリプト言語はその高い生産性や保守性に支えられ，小規模なプログラムから Web アプリケーションなど大規模なプログラムにも使われている．さらに研究レベルでは，性能に関してクリティカルな並列計算分野での活用例も報告されている[8)]．

以上の議論をふまえ，本研究ではマルチコア・プロセッサのソフトウェア開発環境の実現方法として，ターゲット・ハードウェア上で直接 DSP の機械語生成を行わずに，既存のライブラリなど DSP の機能を呼び出し利用するインタプリテーションを採用して DSP 側の機能を活用する．

7.6　処理系の設計

本研究で提案するランタイム環境の処理系を次のように定める．また，**図 9** にこの処理系の概念図を示す．

中間コードの実行　実装するインタプリタの構造を簡単にし，将来的に他プロセッサ・コアのサポートを容易にするために，ソースコードから生成した中間コードを各プロセッサ上で実行する．CPU 側でコンパイラがソースコードから CPU 用と DSP 用の中間コードを生成する．それらを CPU と DSP の各インタプリタが実行する．

ダイナミックリンク・ライブラリ　CPU 側の Linux のライブラリと DSP 側の DSP/BIOS のライブラリを実行時にリンクし利用可能にする．

マルチスレッド対応　CPU と DSP で同時に複数のインタプリタのインスタンスを動作可能にする．

デバッグ環境　特別なハードウェアなしに，ソフトウェアが CPU 側または DSP 側のどちらで実行されていても，透過的なデバッグ作業を行うことができる，

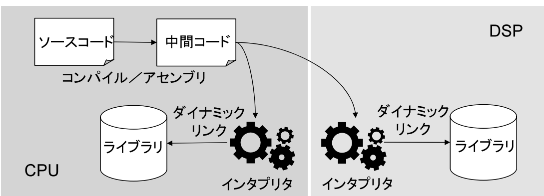


図 9　提案する処理系のモデル
Fig. 9　Conceptual model of the proposed execution environment.

ソースコード・レベルのデバッグ機能を利用可能にする．

8. プロトタイプ・システム

この章では，本研究で提案するソフトウェアの開発環境としてのランタイム環境のプロトタイプ・システムについて述べる．

8.1　利 用 環 境

ターゲット・ハードウェアには TI 製 OMAP5912 プロセッサの評価ボードである OMAP Starter Kit（OSK）を使用した．OMAP5912 プロセッサには CPU として ARM9，DSP として TI 製 C55x の 2 つのプロセッサ・コアが集積されている．**表 2** に OMAP5912 OSK のハードウェアの特徴をまとめた．

プロトタイプ・システムの実装に利用したソフトウェアを**表 3** に示す．プロセッサ間通信用のミドルウェアとして OMAP Linux のディストリビューションに含まれている DSP Gateway を使用した．

8.2　プロトタイプ・システムの実装

次に，先に議論したランタイム環境の設計に基づき実装したプロトタイプ・システムについて述べる．

プログラミング言語　C 言語のサブセットをプログラミング言語として使用した．

プロセッサ間通信　CPU 側で実行しているコードから DSP 側で実行される関数への呼び出しによりプロセッサ間通信が起動する．このとき，関数のパラメータは ARM 側から DSP 側へ渡され，関数の戻り値は DSP 側から ARM 側へ返される．

グローバル変数　ARM と DSP 側からグローバル変数にアクセスさせるために，DSP Gateway で提供されている mmap の機能を利用し両プロセッサからアクセス可能なメモリ領域内に配置した．これらは ARM と DSP 間のデータ転送に使われる程度な

表 2　OMAP5912 OSK の特徴
Table 2　OMAP5912 OSK features.

Processor	CPU	ARM9（192 MHz）
	DSP	C55x（192 MHz）
Memory		DDR SDRAM 32 [MB]
Storage		NOR Flash 16 [MB] × 2
Interfaces		RS232C
		Ethernet RJ45 connector

表 3　利用したソフトウェアの一覧
Table 3　List of software used in the prototype system.

OS	ARM	Linux 2.6.12
	C55x	DSP/BIOS
Middleware		DSP Gateway 3.3

ので，mutex などの排他制御は実装していない．

DSP の関数宣言　関数宣言に使用される extern 構文を拡張することにより，DSP 側で実行される関数の宣言を可能にした．

```
e.g. extern __DSP__ int dspfunc(void);
```

ローカルライブラリ　ARM 側ではインタプリタの実行時に dlopen() などの API を使用して任意のライブラリをダイナミックリンクする．これにより，実行する中間コードから ARM 上にあるライブラリ関数を利用することが可能になる．一方，DSP 側では DSP Gateway が実現している実行時に DSP モジュールをロード，アンロードできる機能を利用する．このとき，DSP Gateway が提供するダイナミックローダ・デーモンが DSP へロードされているモジュールを管理しているので，このダイナミックローダ・デーモンからリンク情報を取得することにより，DSP にロードされているライブラリのダイナミックリンクを実現している．

中間言語　レジスタのサイズや個数が異なりエンディアンも違う ARM と DSP で共通の中間言語を動作させるために，スタック・マシンでバイトコードを実行するモデルを採用した．中間言語としてバイトコードを用いることにより，将来の他プラットフォームへのポータビリティも確保する．また，今回実装したプロトタイプ・システムでは整数型のみサポートする．このバイトコードは大きく分けて，スタック操作，算術・論理演算，条件分岐，関数呼び出しの 4 つのグループから構成される．これらバイトコードの命令名は Java を参考にした．**表 4** はスタック操作に関する命令の一覧である．これはバイトコードを実行するときに割り当てられるスタック領域に対する操作を行うための命令とグローバル変数領域を操作するための命令からなる．

表 5 は算術・論理演算に関する命令の一覧である．これはスタックに格納されている値に対して論理演算・算術演算を行うための命令である．

表 6 は条件分岐に関する命令の一覧である．これは各命令が指定する条件に従いプログラム・カウンタの内容を更新する．プログラム・カウンタはテキスト領域内の実行している命令の位置を示しているので，この内容を更新することにより次に実行される命令を変更できる．また，ここで指定できる条件には，スタックに格納されている値とゼロとの比較，スタックに格納されている 2 つの値の比較，無条件の 3 種類がある．

表 7 は関数呼び出しに関する命令の一覧である．これはバイトコードの配置されているテキストコード内にあるローカル関数の呼び出し，ARM 側から DSP 側のテキスト領域内にある関数の呼び出し，またバイトコードを実行するインタプリタが動的にリンクしたライブラリ関数の呼び出しの 3 種類からなる．

表 4　スタック操作に関する命令
Table 4　Instructions for stack operation.

命令名	動作
load	ローカル変数の値をスタックへプッシュ
load_0	
load_1	
load_2	
load_3	
const_m1	定数をスタックへプッシュ
const_0	
const_1	
const_2	
const_3	
const_4	
const_5	
bipush	1 バイトの即値をスタックへプッシュ
sipush	2 バイトの即値をスタックへプッシュ
store	スタックの値をローカル変数へ格納
store_0	
store_1	
store_2	
store_3	
putfield	スタックに格納されている値をグローバル変数へ格納
getfield	グローバル変数の値をスタックへプッシュ

表 5　算術・論理演算に関する命令
Table 5　Instructions for arithmetical and logical operation.

命令名	動作
add	加算
sub	減算
mul	乗算
div	除算
rem	剰余
neg	ビット反転
iinc	ローカル変数を指定された数だけ増加
and	論理積
xor	排他的論理和

コンパイラ　本プロトタイプ・システムで規定した C 言語のサブセットで記述されたソースコードからバイトコードを生成するためのコンパイラを実装した．今回実装したコンパイラでは最適化処理は行っていない．**表 8** に今回実装した ARM で動作するコンパイラのソースコードの行数（SLOC: Source lines of code）を示す．

インタプリタ　中間言語を処理するためのインタプリタをスタック・マシンとして実装した．一般にインタ

表 6 条件分岐に関する命令
Table 6 Instructions for branch operation.

命令名	動作
goto	無条件分岐
goto_w	
ifeq	スタックに格納されている値とゼロを比較して条件が成立すれば分岐
ifne	
ifle	
iflt	
ifge	
ifgt	
if_compeq	スタックに格納されている 2 つの値を比較し条件が成立すれば分岐
if_compne	
if_comple	
if_complt	
if_compge	
if_compgt	
ifnull	スタックに格納されている値が NULL なら分岐
ifnonnull	スタックに格納されている値が NULL でなければ分岐

表 7 関数呼び出しに関する命令
Table 7 Instructions for function invocation.

命令名	動作
invokestatic	ローカル関数の呼び出し
invokeremote	ARM 側から DSP 側関数の呼び出し
invokelibrary	ライブラリ関数の呼び出し
return	void をリターン
ireturn	整数をリターン

表 8 プロトタイプのソースコード量
Table 8 The amount of source code in the prototype.

対象	ソースコード量 [SLOC]
コンパイラ（ARM）	1,198
インタプリタ（ARM）	2,826
インタプリタ（DSP）	1,761

プリタを実現する方式として主に次の 4 種類がある．

- *Decode-and-Dispatch Interpretation*
- *Indirect Threaded Interpreter*
- *Direct Threaded Interpreter with Predecoding*
- *Binary Translation*

本プロトタイプ・システムで対象にするのが，ネイティブのアセンブラ言語ではなくバイトコードであること，他のターゲット・プロセッサへの移植性が高いこと，構造が単純で実装がしやすいことなどを考慮して，実装するインタプリタの構造としてシンプルな Decode-and-Dispatch Interpretation を選択した．また，今回実装したインタプリタでは実行に関する最適化は行っていない．表 8 に今回実装した ARM 側と DSP 側で動作するインタプリタのソースコードの行数を示す．

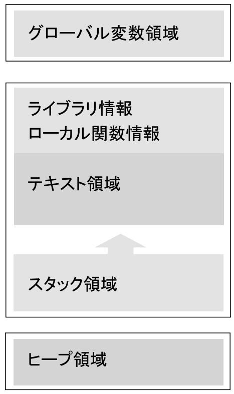


図 10 インタプリタの使用するメモリマップ
Fig. 10 Memory map used by the prototyped interpreter.

8.3 ランタイムのメモリマップ

プロトタイプ・システムとして実装したインタプリタが使用するメモリマップについて説明する．今回実装したインタプリタは**図 10** に示すようにメモリを 6 つの領域に分け使用する．

(1) **グローバル変数領域**
グローバル変数と定数が配置され，ARM と DSP の両方のプログラムからアクセスできる．6 つのメモリ領域の中でグローバル領域だけが ARM と DSP 間で共有される．

(2) **ライブラリ情報**
ダイナミックリンクするライブラリ関数へのアドレスが格納される．また，それらライブラリ関数のシグネチャ情報を保持する．ARM 側には ARM 上のライブラリに関する情報が格納され，DSP 側には DSP 上のライブラリに関する情報が格納される．

(3) **ローカル関数情報**
ローカルに定義された関数の情報が格納される．ここには，関数本体が配置されているテキスト領域内でのオフセット，パラメータ数，また関数内部で定義されたローカル変数の数が格納される．

(4) **テキスト領域**
インタプリタにより実行されるバイトコードが配置される．

(5) **スタック領域**
インタプリタが実行時に使用するスタックが割り当てられる．

(6) **ヒープ領域**
インタプリタが実行時に消費するヒープ領域が割り当てられる．ARM と DSP のインタプリタは各 OS 上で動作する 1 つのプロセス（またはタスク）として実装されているので，各 OS によりプロセス（またはタ

スク）に割り当てられたヒープ領域を使用する．

8.4　未実装の機能

本プロトタイプにおいて次の機能が未実装である．

- 外部プログラムからの呼び出し
- インタプリタのマルチスレッド化
- デバッグ機能

9. 評価実験

この章では，今回実装したプロトタイプ・システムを評価するために行った実験とその結果について述べる．評価実験は次の 5 つの点に注目して行った．

- 基本動作
- プログラミング・スタイル
- ソフトウェアの開発工程
- プロトタイプ・システムの特徴
- 実行速度

TI からは TI 製 DSP で利用可能な標準的なアルゴリズムの実装が DSPLIB というライブラリとして提供されている．この実験では DSPLIB に含まれている FIR フィルタ関数を利用し，既存の開発環境を利用してソフトウェア開発を行う既存手法と，今回実装したプロトタイプ・システムを利用する提案手法とで FIR フィルタ･プログラムを構築してそれらの比較を行った．

9.1　基本動作

まず，既存手法で FIR フィルタ処理を行うプログラムを実装して，今回使用した OMAP OSK 上で正しく動作することを確認した．次に，提案手法で同様の FIR フィルタ処理を行うプログラムを実装し正しく動作することを確認した．これにより，プロトタイプ・システムが既存の開発環境で実現されている基本的な機能を問題なく提供できることを確認した．

9.2　プログラミング・スタイル

既存手法における問題点の 1 つとして指摘したプログラミング・スタイルが，提案手法を利用することによりどのように変化したかを評価する．

図 2 と図 8 に既存手法と提案手法で用いられるプログラミング・スタイルの概念図を示す．このように提案手法では明示的なプロセッサ間通信の記述なしにアプリケーション・プログラムから DSP 側の機能を利用できる．また，アプリケーション・プログラムを構成する ARM 側と DSP 側のコードに対して同じプログラミング・スタイルを適用することができる．

次にプログラミング・スタイルの変化が実際にどのような影響を与えたかを説明する．**図 11** には，プロトタイプ・システムを用いて実装した FIR フィルタ・プログラムのソースコードの関数宣言などを除いたものを示す．このソースコードがランタイムにバイトコードへ翻訳され，ARM と DSP 上で動作するインタプリタにより実行される．図 4 に示す例とは異なり，ここに示す例ではシステムコールを介さずに通常の関数呼び出しのように，目的の機能を実現するために DSP ライブラリ関数をグルーイングしている関数（i.e. fir()）を ARM 側で実行されるコードから呼び出して利用することが可能になった．また，図 5 に示すようなプロセッサ間通信用ミドルウェアにより規定されるデータ構造の定義や低レベルの通信プリミティブ

```
/* Definition of ARM/DSP shared data */
float inbuf[200];
float coef[15];
float outbuf[200];

// FIR filter part executed on DSP
int fir(int tap, int num)
{
  float *pinq15;
  float *poutq15;
  float *pcoefq15;
  float *pdbuf;

  /* Allocate working memory in DSP internal memory */
  pinq15 = malloc(sizeof(float) * num);
  poutq15 = malloc(sizeof(float) * num);
  pcoefq15 = malloc(sizeof(float) * tap);
  pdbuf = malloc(sizeof(float) * (tap + 2));

  /* Adjust arrangement on input data */
  byteorder_conversion(coef, coef, tap);
  byteorder_conversion(inbuf, inbuf, num);

  /* Convert float format into Q15 format */
  fltoq15(coef, pcoefq15, tap);
  fltoq15(inbuf, pinq15, num);

  /* Invoke the FIR library function */
  fir2(pinq15, pcoefq15, poutq15, pdbuf, num, tap);

  /* Convert Q15 format into float format */
  q15tofl(poutq15, outbuf, num);

  /* Adjust arrangement on output data */
  byteorder_conversion(outbuf, outbuf, num);

  free(pinq15);
  free(poutq15);
  free(pcoefq15);
  free(pdbuf);
}

// Main function executed on ARM
int main(void)
{
  char *fin = "in.dat";
  char *fcoef = "coef.dat";
  char *fout = "out.dat";
  int num = 200;
  int tap = 15;

  /* Read input data */
  read_data_array(fin, inbuf, num);

  /* Read coefficient constants */
  read_data_array(fcoef, coef, tap);

  /* Apply the FIR filter */
  fir(tap, num);

  /* Write calculation results */
  write_data_array(fout, outbuf, num);

  return 0;
}
```

図 11　提案環境を利用した FIR フィルタ・プログラムの抜粋
Fig. 11　A piece of source code of FIR filter program developed using the proposed development environment.

を利用する必要がなくなった．しかし，図 11 のソースコード内にはバイトオーダの変換，浮動小数点と固定小数点の変換の記述が含まれている．そこで，関数のパラメータや変数の型情報を有効に利用して，このような型変換を自動化しソースコードの記述を簡略化できるかを今後の検討課題とする．

9.3 ソフトウェアの開発工程

次に既存手法と提案手法を比較してソフトウェアの開発工程に与えた変化について述べる．図 1 と**図 12** に既存手法と提案手法におけるソフトウェア開発環境の概念図を示す．

以下では，FIR フィルタ・プログラムの実装の工程を例にあげ，一般的な開発工程にあてはめて両手法の比較を行う．この FIR フィルタ・プログラムの例では，ARM 側でユーザに対するインタフェースを提供し，DSP 側のライブラリ関数で FIR フィルタ処理を行い，処理結果を ARM 側でファイルに格納する．

（1） **既存手法による開発工程**

既存手法を適用した開発工程の各ステップは以下のようになる．

Step-1 (D)　FIR フィルタ処理ルーチンの実装
Step-2 (D)　処理ルーチンのモジュールテスト
Step-3 (A)　FIR フィルタ・フロントエンドの実装
Step-4 (A)　DSP 側の処理ルーチンとの接続
Step-5 (AD)　インテグレーション・テスト

Step-n (A) は ARM 側，*Step-n (D)* は DSP 側，また *Step-n (AD)* は ARM と DSP の両方を使用した

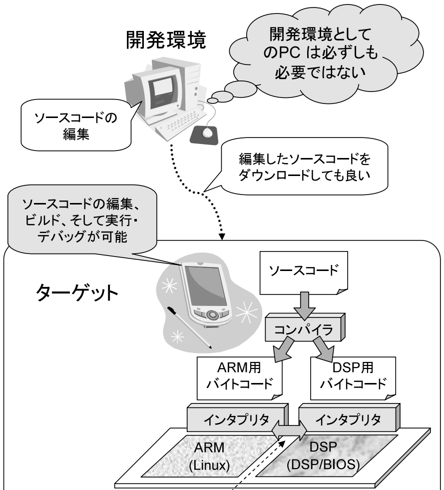


図 12　提案手法におけるソフトウェア開発環境の概念図
Fig. 12　Conceptual diagram of the proposed software development environment.

n 番目の作業である．ここで，ARM 側と DSP 側で並行して作業を始めることができるが，*Step-2 (D)* は *Step-4 (A)* より前に終了している必要がある．最後のインテグレーション・テストで問題が発見されると，*Step-1 (D)* または *Step-3 (A)* へ戻り問題の修正を行う．

（2） **提案手法による開発工程**

提案手法を適用した開発工程の各ステップは以下のようになる．

Step-1 (A)　FIR フィルタ・フロントエンドの実装
Step-2 (A)　FIR フィルタ処理ルーチンの実装
Step-3 (AD)　インテグレーション・テスト

提案手法において，すべての作業を ARM 側で行うことができるので，既存手法のように明示的な DSP 側での作業はなくなる．また，DSP 側で実行される機能も ARM 側で実行される機能と同様に実装することが可能になるので，明示的な ARM 側の機能と DSP 側の機能との接続作業は必要なくなる．最後のインテグレーション・テストで問題が発見されると，*Step-1 (A)* または *Step-2 (A)* へ戻り問題の修正を行う．ただし，提案手法においても，実装するプログラムのサイズが大きくなると，既存手法と同様に個別のモジュールテストが必要になる．

このように，ARM 側で実行される機能と同様に，DSP 側で実行される機能の実装を行うこともできた．また，これまで必要だったホスト・マシン上のソフトウェア開発環境は必ずしも必要ではなくなり，アプリケーション・プログラムのコードの編集，ビルド，そして実行とテストをターゲット・ハードウェア上で行うことができた．したがって，提案手法によりソフトウェアの開発工程が単純化された結果，前述した開発工程の煩雑さが低減されたと考えられる．ただし，本プロトタイプ・システムにおいてデバッグ機能は未実装であるため，デバッグ作業を含めた開発工程を評価することはできなかった．

9.4 プロトタイプ・システムの動的特性

プロトタイプ・システムを評価するために既存手法と提案手法で実装した FIR フィルタ処理を行うための各ソフトウェア・モジュールのサイズを**表 9** に示す．ARM 側のモジュールは ELF ファイル・フォーマットなので，readelf コマンドを使用して各ファイルの情

表 9　FIR フィルタの ARM 側モジュールサイズ
Table 9　Sizes of FIR filter modules on ARM side.

	モジュール名	サイズ	.text サイズ
既存手法	FIR モジュール	14 [KB]	1,436 [bytes]
提案手法	FIR バイトコード	631 [bytes]	190 [bytes]

表 10　FIR フィルタの DSP 側モジュールサイズ
Table 10　Sizes of FIR filter modules on DSP side.

	モジュール名	サイズ	.text サイズ
既存手法	FIR フィルタ	3 [KB]	494 [bytes]

表 11　既存環境で生成された中間コードの量
Table 11　The amount of intermediate codes generated using the existing environment.

	既存環境	
種別	C 言語 [SLOC]	アセンブラ [命令]
ARM 側	35	69
DSP 側	98	113
合計	133	182

表 12　提案環境で生成された中間コードの量
Table 12　The amount of intermediate codes generated using the proposed environment.

	提案環境	
種別	ソースコード [SLOC]	バイトコード [命令]
ARM 側	24	33
DSP 側	22	72
合計	46	105

報を得た．

表 10 に DSP 側の各ソフトウェア・モジュールのサイズを示す．DSP 側のモジュールは TI が拡張した COFF ファイル・フォーマットなので，DSP Gateway 開発のために作成された COFF ファイル解析ツールを使用した．

次にプロトタイプ・システムを構成する各項目についての評価結果を詳細に解説する．

(1)　コンパイラ

まず，今回実装したコンパイラの中間コードの生成効率を評価する．既存環境と提案環境の 2 つの開発環境を使い実装した FIR フィルタ・プログラムのソースコードの行数と生成された中間コード内の命令数を**表 11** と**表 12** に示す．既存環境における中間コードはアセンブラ言語で，提案環境においてはバイトコードである．このとき，既存環境のコンパイラでは最適化オプションを利用していない．また，提案環境のコンパイラでもバイトコードの生成に関する最適化は行っていない．

FIR フィルタ・プログラムの例において，提案環境ではソースコードの行数（SLOC）が既存環境を用いて実現した場合に比べ 35[%] 程度に削減された．また，ソースコードから生成される中間コードの命令数が既存環境で生成されるアセンブラ言語の命令数の 58[%] 程度になった．この減少は既存環境における FIR フィルタ・プログラム内にあったプロセッサ間通信に関する記述が，提案環境におけるソースコードから削減したためだと考えられる．

表 13　コンパイラによる生成される中間コード量の比較
Table 13　Comparison of intermediate codes generated using the compilers.

コンパイラ	中間コード [命令]	種別
既存の ARM 用	75	ARM アセンブラ
既存の DSP 用	58	DSP アセンブラ
プロトタイプ	72	バイトコード

また，既存環境にある ARM 用のコンパイラと DSP 用のコンパイラ，提案環境のコンパイラのコード生成効率を比較するために，FIR フィルタのソースコードの一部を切り出し 3 つのコンパイラで中間コードを生成させた結果を**表 13** に示す．

この結果から，今回評価のために用いた FIR フィルタの一部のソースコードに対して，プロトタイプ・システムのコンパイラは既存の ARM/DSP 用コンパイラと同程度の中間コードの生成効率があることが分かった．

(2)　インタプリタ

次にプロトタイプ・システムのインタプリタの特徴を解析する．ここでは，評価に使用している FIR フィルタ・プログラムのソースコードからコンパイラが生成した中間コードに含まれる各命令の実行コストを測定した．ここでの実行コストとは，中間コードの各命令を実行するのに必要なアセンブラ言語の命令数とする．各命令を ARM 側で実行した場合と DSP 側で実行した場合のそれぞれのコストを**表 14** に示す．

また，インタプリタの動作を詳しく見ると各命令を実行するために，中間コードの格納されているメモリ領域から命令をフェッチし該当する処理ルーチンへ制御を移している．この命令のフェッチとディスパッチにかかるコストを ARM 側と DSP 側で測定した結果を**表 15** に示す．このコストは各命令の実行ごとに必要である．

以上のことから，FIR フィルタ・プログラムを実行するためのコストは，ARM 側の各命令の実行コスト × 各命令の出現回数 + DSP 側の各命令の実行コスト × 各命令の出現回数となり，次のように求めることができる．

$$C_{total} = \sum_{n}^{ARM} Cost(n) \times Freq(n) + \sum_{n}^{DSP} Cost(n) \times Freq(n)$$

表 14 中間コードに含まれる各命令の実行時コスト
Table 14 Execution cost of each instruction in the generated intermediate code.

命令	コスト（ARM 側）	コスト（DSP 側）
bipush	18	13
getfield	40	25
add	25	13
const_0	13	10
const_2	13	10
const_4	13	10
load	23	14
load_0	15	10
load_1	17	11
load_2	17	11
load_3	17	11
mul	25	12
invokelibrary	22	12
invokeremote *	242	n/a
ireturn	161	92
store	23	13
store_1	17	9
store_2	17	9
sotre_3	17	9
return	9	n/a

invokeremote は引数が 2 つで戻り値がある場合
（単位はアセンブラ言語の命令数）

表 15 フェッチとディスパッチのコスト
Table 15 Execution cost of the fetch and dispatch routines.

	フェッチ	ディスパッチ
ARM 側	4	10
DSP 側	3	25

（単位はアセンブラ言語の命令数）

このとき，$Cost$ は

$$Cost(n) = C_{exec}(n) + C_{fetch}(n) + C_{dispatch}(n)$$

とする．各変数は次のように定義する．

C_{total} 全実行コスト
$C_{exec}(n)$ 各命令の実行コスト
$C_{fetch}(n)$ フェッチコスト
$C_{dispatch}(n)$ ディスパッチコスト
$Freq(n)$ 各命令の出現回数

実際に測定された数値に上記の式を適用し実行時のコストを求めると，

$$C_{total} = 947 + 462 + 1{,}015 + 2{,}016$$
$$= 4{,}440$$

提案環境では FIR フィルタ・プログラムを実行するのにアセンブラ言語の 4,440 命令が必要だったことが分かった．既存環境では同等のプログラムを 182 命令で実行できたので，既存環境に比べ約 24 倍の実行コストが必要であることが分かった．

表 16 FIR フィルタ・プログラムの実行時間
Table 16 Execution time of the FIR filter programs.

	既存手法 [ms]	提案手法 [ms]
前処理	135 ± 2	1,068 ± 12
FIR フィルタ処理	37	35
後処理	< 1	8
合計時間	172	1,111

表 17 提案環境での前処理の内訳
Table 17 The detail of preparation time in the proposed environment.

フェーズ	経過時間 [ms]
DSP へインタプリタモジュールのロード	304.5 ± 3.9
MMAP によるメモリのマップ	0.4
ARM 側ライブラリの dlopen による動的リンク	62.3 ± 19.5
DSP 側ソフトウェアのリンク情報取得	589.0 ± 11.9
FIR プログラムのバイトコード読み込み	73.7
インタプリタの初期化	0.1
グローバルデータの配置	0.3
ライブラリのリンク情報の配置	0.2
バイトコードの関数情報の配置	0.4
バイトコードの配置	36.8
合計	1,067.7

表 18 既存環境での前処理の内訳
Table 18 The detail of preparation time in the existing environment.

フェーズ	経過時間 [ms]
DSP への FIR フィルタモジュールのロード	134.2 ± 2.0
MMAP 処理	0.5
データ数の設定	0.3
フィルタのタップ数の設定	0.2
合計	135.1

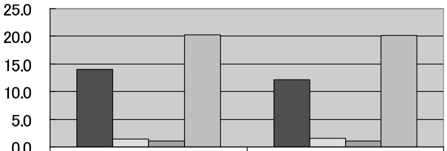
(3) 実行速度

最後に，実装した FIR フィルタ・プログラムの実行時間を解析する．既存環境と提案環境のそれぞれで FIR フィルタ・プログラムを実行したときに，プログラムの各フェーズで経過した時間を**表 16** に示す．以降の実行時間を示す表では，測定した各項目の標準偏差を求め，その値が大きいものは表内に “測定値 ± 誤差” と表記する．

この結果から，提案環境では既存環境に比べ前処理に大きく時間を費やしていることが分かる．提案環境ではインタプリタがプログラムの実行を行っており，前処理に相当するインタプリタ内での実行時間をより詳細に解析した結果を**表 17** に示す．

一方，既存環境における前処理の実行時間の内訳を**表 18** に示す．

提案環境において最も時間のかかっている箇所は，DSP 側ソフトウェアのリンク情報取得であることが分かる．



	既存の実装法 [ms]	提案手法 [ms]
■ データ入力	14.1	12.1
□ 係数入力	1.4	1.5
■ FIRフィルタ処理	1.0	1.0
□ 計算結果出力	20.3	20.2

図 13 FIR フィルタ処理の実行時間の内訳
Fig. 13 Detailed list of execution time of the FIR filter routines.

ここに時間が消費される理由は現在の実装方法によるものと考える．それは現在の実装が DSP 側のリンク情報を取得するために，プロセッサ間通信用ミドルウェアである DSP Gateway が管理しているシンボル情報を，一度ファイルに出力させてから解析しリンクに必要な情報を得ている．したがって，実装方法を改善することにより，このリンク情報取得に要する時間を短縮できる可能性があるものと考える．

次に前処理と後処理を除く FIR フィルタ処理自体の実行時間を解析する．既存環境と提案環境での FIR フィルタ・プログラムの実行時間とその内訳を**図 13** に示す．

ここに示すように，FIR フィルタ自体の処理は既存環境での実行に比べ，提案環境では 1[%] 程度の処理時間の増加にとどまった．これは実行した処理の大半がライブラリ内で処理されインタプリタでの処理量を大幅に上回ったために処理時間の増加が低く抑えられたと考える．また，測定結果において既存環境と提案環境におけるデータ入出力の実行時間の差は，既存環境では Linux のシェアードライブラリを使用したのに対して，提案環境では該当ライブラリを dlopen でダイナミックリンクしたためだと考える．それは，dlopen を使用すると，通常のダイナミックリンク時に最初の一度だけ行われるシンボル解決に要する時間が省かれるためだと考える．

10. おわりに

本論文では，組込み向けマルチコア・プロセッサを用いてソフトウェア開発を行うときの問題点に，組込みシステム特有のソフトウェアの開発工程に煩雑さがある点と，低レベルの通信プリミティブを利用するプロセッサ間通信に依存したプログラミング・スタイルを適用しなければならない点を示した．それらの問題点を考慮してソフトウェア開発環境としてのランタイム環境を設計した．その有効性を評価するために組込み向けマルチコア・プロセッサである OMAP プロセッサ上にプロトタイプ・システムを実装した．このプロトタイプ・システムを用いることにより，提案環境において統一したプログラミング・スタイルが適用可能であること，ターゲット・プロセッサである OMAP プロセッサ上で動作するコンパイラとインタプリタによりソフトウェアの開発工程が単純化されたことを確認することができた．

次に，既存環境と提案環境の両方で FIR フィルタ・プログラムを評価用のサンプルとして実装し，プロトタイプ・システムの評価を行った．その結果，プロトタイプ・システムを使用する場合は既存環境での実行に比べ，FIR フィルタ・プログラム実行のために必要とされるアセンブラ言語の命令数が約 24 倍に増えた．しかし，FIR フィルタの処理時間では既存環境と提案環境で大きな違いを認めることはできなかった．これは提案環境では ARM 側と DSP 側のライブラリを利用できるので，プロトタイプ・システムでインタプリットするソースコードは基本的にライブラリ関数のグルーイングが主目的になり，グルーコードの量をネイティブのライブラリ関数での処理量が大幅に上回ったためだと考えられる．つまり，ネイティブで処理されるライブラリ関数での処理量が多ければ，グルーコードの実行にともなうオーバヘッドの増加量は全体の処理時間に占める割合に比べ小さくなる．この結果から，今回実装したプロトタイプ・システムにおける前処理時間の短縮を図れば，処理時間の観点からは提案環境を実際のソフトウェア開発に適用できる可能性があると考える．

今後の課題としては，DSP で動作するコーデックにより現実に近いオーディオやビデオ・コーデックを用いて，評価基準として実行速度に加えメモリ消費量も考慮し評価を行いたい．加えて，記述するソースコードの量を現在よりもさらに削減するために，対象のソースコードから型情報を取得し自動的な型変換機構の実現可能性を検討したい．これにより，一般プログラマにとってさらに利用しやすい環境になると期待する．また，W3C の SMIL のようなプレゼンテーション記述のセマンティックスを言語仕様に取り入れられるかを検討したい．そして，実際の PDA（e.g. Nokia 770）を用いてアプリケーション・プログラムのプログラミング環境としての適用可能性の検討，提案環境を拡張しデバッグ環境としての利用方法などの検討を進めたい．

参 考 文 献

1) Savola, T.: Scratchbox, Cross-compiling a Linux distribution, *FOSDEM* (2005).
2) Bellard, F.: QEMU, a Fast and Portable Dynamic Translator, *USENIX Annual Technical Conference*, FREENIX Track (2005).
3) HotPaw Basic Version 1.4.5, HotPaw Company (2005).
4) Common Object Request Broker Architecture, Core Specification version 3.0.3, OMG (2004).
5) IEEE Std 1149.1-1990 IEEE Standard Test Access Port and Boundary-Scan Architecture - Description (1990).
6) Zhou, S., Stumm, M., Li, K. and Wortman, D.: Heterogeneous Distributed Shared Memory, *IEEE Trans. Parallel and Distributed Systems*, Vol.3, No.5, pp.540–554 (1992).
7) Wortman, D.B., Zhou, S. and Fink, S.: Automating Data Conversion for Heterogeneous Distributed Shared Memory, *Software—Practice & Experience*, Vol.24, No.1, pp.111–125 (1994).
8) 湯山紘史，津邑公暁，中島 浩：タスク並列言語 MegaScript における高精度実行モデルの構築，情報処理学会論文誌：コンピューティングシステム，Vol.46, No.SIG 12(ACS 11), pp.181–193 (2005).
9) Ousterhout, J.K.: Scripting: Higher Level Programming for the 21st Century, *IEEE Computer*, Vol.31, No.3, pp.23–30 (1998).
10) Takahashi, K. and Mori, E.: Architectural Design of a DSP Scripting Language for Mobile Multimedia Terminals, *AINA*, Vol.2, pp.245–248 (2004).
11) Smith, J.E., Nain, R. and Nair, R.: *Virtual Machines: Versatile Platforms For Systems and Processes*, Morgan Kaufmann Pub. (2005).
12) Aho, A.V., Sethi, R. and Ullman, J.D.: Compilers — Principles, Techniques, and Tools, Addison Wesley (1986). 原田賢一（訳）：コンパイラ 原理・技法・ツール I, II，サイエンス社 (1990).
13) Kobayashi, T.: Linux DSP Gateway Specification Rev. 3.3, Nokia (2005).
14) TMS320C5000 DSP/BIOS 5.20 Application Programming Interface (API) Ref. Guide (Rev. J), Texas Instruments (2005).
15) TMS320C55x DSP Library Programmer's Reference (Rev. H), Texas Instruments (2004).

(平成 18 年 7 月 4 日受付)
(平成 18 年 12 月 6 日採録)

髙橋 清隆（正会員）

1994 年東洋大学工学部卒業．同年（株）ソリトンシステムズ入社．その後，日本ルーセント・テクノロジー（株）を経て，現在，ノキア・ジャパン（株）ノキア・リサーチセンター勤務．2004 年から東京工業大学大学院情報数理工学研究科博士後期課程在学中．携帯端末をはじめとする組込みシステムのソフトウェアに関する研究に従事．

柴山 悦哉（正会員）

1981 年京都大学理学部卒業．1983 年同大学院修士課程修了．同年東京工業大学理学部助手．その後，龍谷大学理工学部講師，東京工業大学理学部助教授，同大学院情報理工学研究科助教授を経て，2000 年より同研究科教授．理学博士．プログラミング言語，ソフトウェアセキュリティ，ユーザインタフェースソフトウェア等に興味を持つ．ACM，USENIX，IEEE CS，日本ソフトウェア科学会各会員．