アセンブリ言語レベルでの異種計算機間のヒープとスタックの共有機構

上田陽平[†] 山本泰宇[†]
関口龍郎[†] 米澤明憲[†]

本論文では，ソフトウェア分散共有メモリを用いた異種計算機間のメモリ上のデータの共有を特定の言語処理系に依存せずに実現する手法を示す．異機種間のデータ共有にはデータの表現形式の違いの解決が必要である．従来の手法では高級言語からメモリ上のデータの型情報を得て，通信時にデータ表現形式を変換していた．一方，我々の手法ではメモリ上のデータを，計算機固有のデータ表現形式ではなく，計算機間で共通のデータ表現形式を用いて表現し，メモリアクセス時にアセンブリ命令の型を基にデータ表現形式を変換する．この手法は高級言語の型情報に依存せずに異機種間のデータ共有を実現でき，さらに，C や C++などの型キャストができる言語もサポートできる．また，我々はこの手法をスレッドのスタックにも適用し，計算機間のスレッドの移動に必要なスタックを共有する手法も考案した．我々はこの手法を用いて，SPARC と x86 プロセッサの計算機の間でヒープとスタックを共有可能なシステムを実装し，そのオーバヘッドを測定した．GCC と比較した場合，我々の手法のオーバヘッドによる実行時間の増加は SPARC では平均 14.7%，x86 では平均 49.0%であった．

An Assembly-language-level Mechanism of Shared Heaps and Stacks for Heterogeneous Computers

Yohei Ueda,[†] Hirotaka Yamamoto,[†] Tatsurou Sekiguchi[†] and Akinori Yonezawa[†]

This paper describes a language-independent mechanism for software distributed shared memory (DSM) on heterogeneous computers. Sharing data among heterogeneous computers requires conversion of data representation. In traditional approaches, representation of memory data is converted at every communication occurrence using type information provided by high-level languages. In our mechanism, data on memory are represented not in a native format but in an architecture-independent format, where a data representation is converted at every memory access depending on types of assembly instructions. Since this mechanism does not depend on type information from a high-level language, we can share data among heterogeneous computers without language support. As a result, our mechanism can be used with the languages that allow type-casting, such as C and C++. With minor enhancement, our mechanism can share call stacks of threads as well as data on heaps among heterogeneous computers. Shared stack representations enable threads to migrate to another computer. We have implemented a system that can share heaps and stacks between SPARC and x86 computers, and measured overheads of our mechanism. The average overheads in execution time is 14.7% on a SPARC processor and is 49.0% on an x86 processor as compared with GCC.

1. はじめに

ネットワーク技術の発展により，多数の計算機を用いた分散計算の重要性が高まっている．しかし，1 台の計算機でのプログラミングと比べて，分散した計算機を使ったプログラミングは一般に非常に労力を要する．さらに，異なる機種の計算機を含む環境ではその困難さが増大する．したがって，分散性や異機種性を吸収し，分散計算機を 1 つの計算機に見せるシステムが望まれている．このようなシステムが持つ特徴は単一システムイメージ（Single System Image，SSI）[2] と呼ばれている．SSI は異機種分散計算機の環境に共有メモリ型のプログラミングモデルを提供し，透過的な資源のアクセスを可能とする．したがって，分散共有メモリや計算機間のスレッドの移動といった技術が SSI の実現にとって有力な技法となっている．様々なシステムがこの SSI の実現を目指しているが，それ

† 東京大学大学院理学系研究科情報科学専攻
Department of Information Science, Graduate School of Science, University of Tokyo

らのシステムの多くは，SSI の実現に必要な計算機の違いの吸収を特定の言語システムのランタイムのレベルで行っている[1),6)~8)]．このため，それ以外の言語や言語システムおよびソフトウェアの再利用を妨げてしまう．

言語非依存に SSI を実現するにはソフトウェア分散共有メモリ（Distributed Shared Memory，DSM）[3)~5)]を利用することが考えられる．ソフトウェア DSM はプログラミング言語に依存せずに分散した計算機間に共有メモリ空間を提供するシステムである．ソフトウェア DSM は分散環境に共有メモリ型のプログラミングモデルを提供するため，分散プログラミングを簡単にする．多くのソフトウェア DSM では，ページなど一定の単位でメモリ空間を分割し，その単位で共有メモリの一貫性を維持している．そのようなソフトウェア DSM は特定のプログラミング言語に依存することなく実現することができる．

しかし，異機種の計算機間，特にデータの表現形式が異なる計算機間では，単純にソフトウェア DSM を実現することはできない．データの表現形式をその機種で使われるものに変換するためには，そのデータの型情報が必要だからである．たとえば，ビッグエンディアンとリトルエンディアンの計算機間で整数を転送する場合にはバイトオーダーを交換しなければならないが，文字列に対しては変換してはならない．今日までに，様々な異種計算機間のソフトウェア DSM のシステムが実現されてきたが，これらのシステムでは特定の言語のランタイムがすべての型情報を管理するか[8)]，プログラマが明示的に型を指定する必要があるか[6),9)]，使用できる型を制限している[7)]．これらの制限の原因はデータの表現形式の変換をデータ通信時に行っているからである．通信時のデータ変換を実現するにはページに含まれる任意のアドレスに格納されているデータの型が判別できなければならない．しかし，これは Java のようにランタイムがデータの型情報を管理している言語では可能だが，C や C++などの実行時に型情報を持たない言語ではほとんど不可能である．

我々は，データの変換をデータ通信時に行うのではなく，メモリアクセス時に行うことでデータ表現形式の違いを解決した．つまり，メモリ上のデータを計算機固有のデータ表現形式ではなく，計算機間で共通のデータ表現形式で表し，メモリアクセス時にデータの表現形式を変換する．変換に必要なデータの型はアセンブリ言語のメモリアクセス命令の種類によって判断し，適切なデータ表現形式の変換コードを各メモリアクセス命令に付加する．この方法はアセンブリ言語レベルの情報しか必要とせず，特定の言語に依存せずに実現できる．我々はこの手法を**単一メモリイメージ**（Single Memory Image，SMI）と呼んでいる．

また，我々は，ヒープだけではなくスタックにも単一メモリイメージを適用することにより，計算機間でのスレッド移動に必要なスレッドのスタックの共有を実現する手法も考案した．これは，計算機間で同一のスタックフレームのレイアウトを用い，さらに，スタック上の機種依存のデータを計算機間で共有可能にしたものに置き換えることで実現する．

我々は，単一メモリイメージを SPARC と x86 プロセッサの計算機の上に実装し，単一メモリイメージが低いオーバヘッドで実現できることを確認した．実装には機種非依存なアセンブリ言語である Machine Independent Code（MIC）[10)] を用いた．MIC コードから実際の機械語へ変換する際に，メモリアクセス命令にデータ変換コードを付加している．また，MIC を用いることで，異機種間でのスレッドの移動に関わるいくつかの問題を解決することができる．そして，単一メモリイメージのオーバヘッドを SPARC と x86 プロセッサで測定し，さらに Eager Release Consistency [3)] に基づく page-based なソフトウェア DSM を実装して，その上で単一メモリイメージが動くことを確かめた．

本論文の残りの構成は次のとおりである．2 章では，単一メモリイメージの枠組みについて述べる．3 章では，MIC の簡単な説明を述べる．4 章では我々が行った SPARC と x86 プロセッサ上での単一メモリイメージの実装について述べる．5 章では単一メモリイメージのオーバヘッドの測定について報告する．6 章では既存の異機種ソフトウェア DSM や SSI システムと我々の枠組みとの比較を行う．最後の 7 章では，本論文のまとめと今後の課題について述べる．

2. 単一メモリイメージ

単一メモリイメージとは，計算機間で共通のデータ表現形式をあらかじめ選んでおき，メモリ上のデータをその共通のデータ表現形式で表したものである．つまり，すべての計算機のメモリ上のデータは，機種によらず同じ表現形式で表される．したがって，ソフトウェア DSM を用いて異機種間で単一メモリイメージを共有することにより，透過的な分散計算環境を実現することができる．異機種間でデータの共有を実現するには，まず，データの表現形式の違いを解決しなければならない．メモリ上のデータの表現形式の違いに

表 1 主要プロセッサのデフォルトのデータ表現形式
Table 1 Data representation formats on commodity processors.

プロセッサ	バイトオーダー	1 語の長さ	負の整数	浮動小数点数	バイトオーダー変換命令
Alpha	リトルエンディアン	64 ビット	2 の補数	IEEE 754	なし
PowerPC	ビッグエンディアン	32/64 ビット	2 の補数	IEEE 754	`lwbrx/swbrx`
R10000	ビッグエンディアン	32/64 ビット	2 の補数	IEEE 754	なし
SPARC	ビッグエンディアン	32/64 ビット	2 の補数	IEEE 754	`lda/sta`
x86	リトルエンディアン	32 ビット	2 の補数	IEEE 754	`bswap`

はバイトオーダー（ビッグエンディアン，リトルエンディアン），1 語の長さ（32 ビット，64 ビット），負の整数の表現（2 の補数，1 の補数），浮動小数点数の形式（IEEE 754 など）がある．

表 1 は主要なプロセッサのデフォルトのデータ表現形式を示している．すべてのプロセッサで負の整数の表現に 2 の補数を採用している．また，浮動小数点数もすべてのプロセッサが IEEE 754 フォーマットを採用している．このことから，負の整数と浮動小数点数は同じフォーマットであると仮定しても一般的な計算機では問題はない．一方，バイトオーダーと 1 語の長さはプロセッサごとに異なっている．したがって，これらの違いを吸収しなければ異機種間でのデータ共有は不可能である．

また，データの表現形式の違いを吸収するだけでは十分ではない．変数や配列などのデータや関数の実行コードは機種ごとに異なる場所に配置されるため，変数や関数などへのポインタは機種ごとに異なった値になる．したがって，ポインタの値を異機種間で共有するには，ポインタの値も異機種間で同じ値になる必要があり．このデータのレイアウトの違いも解決しなければならない．

さらに，異機種間でのスレッドの移動を可能にするためには，スレッドのスタックを共有する必要がある．しかし，スタックフレームのレイアウトは機種ごとに異なり，また，スタック上にはフレームポインタや関数のリターンアドレスなどの機種依存データが保存されているため，単純にソフトウェア DSM を用いてスタックを共有することはできない．

以上をまとめると，異機種間でのデータ共有で解決すべき問題点は次のとおりである．

- データの表現形式の違い
- ポインタの値の違い
- スタックの機種依存性

次節以降では，単一メモリイメージでのこれらの問題点の解決方法を述べる．

2.1 メモリアクセス時変換

単一メモリイメージではメモリ上のデータにアクセスしたときにデータ表現形式を変換することにより，異機種間でのデータの共有を実現する．アセンブリ言語のメモリアクセス命令は 32 ビットの整数，16 ビットの整数，8 ビットの整数，64 ビットの浮動小数点数，32 ビットの浮動小数点数などの単位でデータにアクセスする．したがって，どの種類のメモリアクセス命令が用いられているかによって，アクセスするデータの型が判別が可能である．そして，あらかじめ計算機間で共通のデータ表現形式を決めておき，メモリ上のデータはその共通のデータ表現形式で表すことにする．そして，メモリからレジスタに読み込むときに機種固有のデータ表現形式に変換し，メモリに書き出すときは逆に共通のデータ表現形式に変換する．メモリアクセス時のデータ変換は，変換コードをロード命令およびストア命令に付加することによって実現できる．

表 2 実行時のメモリアクセス命令の割合
Table 2 Ratios of memory access instructions during execution.

	整数	浮動小数点数	合計
LU	0.9%	39.2%	40.1%
RAYTRACE	6.4%	16.1%	22.5%
N-QUEEN	30.7%	0.0%	30.7%
QSORT	26.8%	0.0%	26.8%

表 2 は MIC コードにコンパイルされたプログラムを実行したときの，実行された MIC の命令数に対するメモリアクセス命令数の割合である（MIC は 3 章で，各プログラムの詳細は 5 章で説明する）．この表が示すように，プログラムに占めるメモリアクセス命令の割合は少なくなく，メモリアクセス時に大きなオーバヘッドが生じると全体の実行時間に大きな影響を及ぼすと考えられる．したがって，メモリアクセス時のデータ変換に必要なオーバヘッドは最小限におさえる必要がある．

データの表現形式の違いで最も問題となるのはバイトオーダーの違いであるが，多くのプロセッサでバイトオーダーを変換する命令が用意されている．たとえば，i486 以降の x86 プロセッサには `bswap` という整数レジスタ上のデータのバイトオーダーを逆にする命

令が用意されている．また，SPARC v9 には lda/sta というバイトオーダーを逆にしてロード/ストアする命令が用意されている．PowerPC にも lwbrx/swbrx というバイトオーダーを逆にしてロード/ストアする命令がある．したがって，これらのプロセッサではメモリアクセス時にバイトオーダーの変換を効率良く行うことが可能である．特に，共通のデータ表現形式をリトルエンディアンにした場合には，表 1 のプロセッサのうち 4 つまでを効率良くサポートできることになる．一方，共通のデータ表現形式をビッグエンディアンにした場合にも表 1 のプロセッサのうち 4 つをサポートできるが，x86 の bswap は整数レジスタ専用であり，x86 での浮動小数点レジスタ上のデータのバイトオーダーの変換には比較的大きなオーバヘッドがかかってしまう．

2.2 共通のメモリレイアウト

計算機間で変数や配列が配置されるアドレスが異なると，ポインタの値が異なってしまうため，単一メモリイメージでは，計算機間で変数や配列をそれぞれ同じアドレスに配置している．このような共通のメモリレイアウトを採用することよって，変数や配列へのポインタは異機種間でも同じ値となる．実行時に動的に確保されるヒープも計算機間で同じアドレス配置できるため，ヒープ上の変数や配列へのポインタも計算機間で同一の値となる．

実行時にヒープを動的に確保する際には，確保されるメモリ領域がすでに他の計算機で確保されているヒープと重ならないようにしなければならない．これは各計算機がヒープを確保できる領域を計算機間で重ならないようにあらかじめ決めておき，その領域からヒープを割り当てていくことで実現できる．もし，その領域が不足した場合には他の計算機と交渉して新たな領域を確保すればよい．

また，データの大きさの違いもメモリのレイアウトに違いを生じさせる．単一メモリイメージではすべての計算機で 1 語の長さを 32 ビットとすることで解決している．したがって，64 ビットの計算機では 32 ビット幅のメモリ空間のみが用いられることになる．この方法では 64 ビットの計算機の機能を十分使い切っていないことになるが，データとメモリ空間を 32 ビットの計算機と共有する際には当然必要となる制限であると我々は考えている．

2.2.1 間接コール・間接リターン

関数へのアドレスも計算機間で同じ値にならなければならない．単一メモリイメージでは計算機間で変数や配列を同じサイズにするので，同一のアドレスに配置するのは容易である．しかし，関数の実行コードは機種によって異なるサイズとなるため，そのままでは同じアドレスに関数を配置するのは難しい．

したがって，関数呼び出しは関数コードに直接ジャンプするのではなく，**間接コール**を用いることによって，関数へのアドレスが同じ値であるようにプログラマに見せる必要がある．

また，異機種間の実行コードはまったく別のものであるため，関数のリターンアドレスも異機種間で異なる値となるが，間接コールと同様に**間接リターン**を用いればリターンアドレスを計算機間で同じ値にそろえることができる．後に述べるように，リターンアドレスを計算機間で共有できればスタック共有の実現が容易になる．

2.2.2 共通のスタックフレームレイアウト

計算機間でスレッドのスタックが共有できれば，計算機間でのスレッドの移動が可能となる．しかし，計算機本来のスタックフレームのレイアウトは機種ごとに異なり，また，スタック上に様々な機種依存のデータがあるため，今まで述べた手法だけではスタックを共有するには不十分である．単一メモリイメージでは，異機種間で共通のスタックフレームのレイアウトを用いたスタックを，計算機本来のスタックとは別の場所に作ることで，スタックの共有を実現している．この単一メモリイメージのスタックの上には計算機間のスレッド移動を妨げるような機種依存のデータは置かれない．

スレッドのスタックには関数の局所変数が配置されているが，これらの変数にも共通のデータ表現形式を用いる．また，スタックフレームを共通のレイアウトとすることで，局所変数のアドレスも異機種間で同じになり，局所変数へのポインタの値も計算機間で同一にすることができる．

計算機本来のスタックの上にある機種依存のデータには，まず，フレームポインタの値と関数のリターンアドレスがある．フレームポインタの値は計算機間で同じスタックフレームのレイアウトを用いることで同一の値にすることができる．また，関数リターンアドレスの違いの解決方法には前述の間接リターンを用いればよい．

局所変数やフレームポインタ，リターンアドレスなどのほかに，計算機本来のスタックの上には，整数と浮動小数点数との変換などの用途に用いるテンポラリなメモリ領域がある．このテンポラリなメモリ領域も異機種間で同一サイズで適当な大きさのものを用意する．

さらに，コンテキストスイッチ時のレジスタの退避領域がスタック上に確保されている機種があるが，このような機種依存のデータはスタック上に置かないようにする．この場合，OS が想定しないスタックのレイアウトを用いることになるので，本来，スタックポインタを表しているレジスタをスタックポインタとして用いることはできない．したがって，このような機種の単一メモリイメージでは別のレジスタをスタックポインタとして用い，本来のレジスタはまったく使わないようにしておく．このようにしておくことで，OS からは本来のスタックがまったく増減していないように見え，コンテキストスイッチの際のレジスタの退避も正しく行われる．

2.2.3 レジスタの退避

計算機間でスレッドが移動する際には，すべての局所変数が共通のデータ表現形式でスタック上に保存されていなければならない．これは，関数呼び出しの際に行われるレジスタの待避をつねに呼び出し側で行うこと（コーラーセーブ）で保証できる．つまり，呼び出す側がレジスタの値をスタックに退避する場合は，コンパイラはそのレジスタの値が表している変数を知っているため，普通は適切な型のストア命令で，対応する変数のスタック上の領域に保存するコードを生成する．ストア命令が適切な型であれば，正しく共通のデータ表現形式でスタックに書き込むことができる．もし，呼び出された関数が破壊するレジスタの値を退避する場合には，そのレジスタ上のデータの型が分からないので，不適切な型のストア命令でスタックに保存することになり，共通形式に変換してスタックに保存することができない．

2.3 バイトコードの利用

スレッドの移動を実現するためには，移動前のスレッドの状態が移動後にも復元されなければならない．しかし，各機種専用のコンパイラによって生成された実行コードをそれぞれの計算機で用いる場合にはいくつかの問題点がある．たとえば，各機種で異なる大域的な最適化を行うことによって，変数が生きている期間が機種ごとに異なる可能性がある．この場合，スレッドの移動の前に生きていた変数が，移動後には死んでいるといった状況が起こりうる．このように，各機種専用のコンパイラを用いた場合，スレッド移動前の状態をスレッド移動後に復元するのが一般に困難である．

我々は，このような問題点を解決するために，異機種間で共通のバイトコードを利用することにした．コンパイラが出力したバイトコードをさらに各機種の機械語に変換するのである．大域的な最適化はコンパイラが行い，バイトコードから機械語への変換時には局所的な最適化のみが行われる．したがって，バイトコードを利用することによってスレッドの移動に必要な状態の復元が容易になる．

コンパイラが生成するバイトコードは一定の条件を満たしている必要がある．その条件とは,(1) メモリ上のデータへのアクセスは適切な型のロード/ストア命令で行い,(2) 単一メモリイメージのスタックフレームの使用法を守り,(3) 単一メモリイメージの関数呼び出し規約を守ることである．これらの条件が守られていれば，単一メモリイメージを用いた実行が可能となる．

2.4 スレッドの移動

これまで述べてきた単一メモリイメージの条件下で，関数の先頭での計算機間のスレッドの移動が可能となる．これは，関数呼び出し時には関数の局所変数はすべてスタックに保存されており，また，レジスタ渡しされる引数も，コンパイラはその型が分かっているため，正しい型のストア命令でスタックに書き出すことが可能であるからである．

3. MIC

我々が単一メモリイメージの実験に利用した MIC（Machine Independent Code）[10] とは異機種環境に容易に適応可能なアセンブラ言語である．MIC で記述されたコードは，機種固有の C コンパイラで生成されたコードとほぼ同程度の速度で動作可能である．MIC コードの意味は MIC 抽象機械として定義されている．この章では MIC に関して簡単な説明を行う．

3.1 命令セット

図 1 に MIC 命令セット（*opcode*）とアドレッシングモード（*addr*）を示す．命令の中で括弧を付けられているものはバージョン 2（MIC2）で導入された命令を表す．本論文の結果はバージョン 1（MIC1）に基づいているため，この節では MIC1 についてのみ説明を行う．オペランドの形式について詳細な解説は行わないが，命令の大部分は 3 アドレス形式であり，第 1 オペランドと第 2 オペランドの値を利用して演算を行い，その結果を第 3 オペランドに格納する．命令が演算対象とする値のデータ型は命令ごとに定まっている．

3.2 データ型

データ型として 32 ビット整数，64 ビット整数，32 ビット浮動小数点数，64 ビット浮動小数点数がある．またメモリ上でのみ存在するデータ型として 8 ビット整数，16 ビット整数がある．ビットオーダー，バイ

$opcode$ ::= nop
(ロード命令)
| ldsb | ldub | ldsh | lduh | ld | ldd | ldf | lddf
| ldfi | lddfi
(ストア命令)
| stb | sth | st | std | stf | stdf | stfi | stdfi
(ムーブ命令)
| mov | setdf | (movfgs) | (movgfs) | (movfgd)
| (movgfd)
(条件ムーブ命令)
| (cmovSC) | (fcmovSF) | (fcmoveSF)
(非分割実行命令)
| swap
(論理命令)
| not | and | or | xor | andn | (orn) | (xnor)
(シフト命令)
| sll | srl | sra
(符号拡張命令)
| zxhtos | zxqtoh | zxqtos | sxhtos | sxqtoh | sxqtos
(算術命令)
| neg | add | sub | addd | subd | smul | umul
| smuld | umuld | sdiv | udiv | smod | umod
(条件分岐命令)
| cb$C$
(無条件分岐命令)
| ba | btt
(関数支援命令)
| prologue | epilogue | call$T$ | ret$T$
(精度変換命令)
| fstod | fdtos
(浮動小数点演算命令)
| fmovs | fmovd | fnegs | fnegd | fabss | fabsd
| fsqrts | fsqrtd | fadds | faddd | fsubs | fsubd
| fmuls | fmuld | fdivs | fdivd
(浮動小数点比較命令)
| fcmps | fcmpd | fcmpes | fcmped
(浮動小数点条件分岐命令)
| fb$F$

$addr$ ::= $[r]$ | $[r+r]$ | $[r \pm disp]$ | $[r + r * scale]$
| [got + $label$]

T ::= ε | i | l | f | d

S ::= i | f | d

C ::= ne | e | g | ge | l | le | gu | leu | geu | lu

F ::= ne | e | g | ge | l | le | u | ug | ul | lg | ue
| uge | ule | o

$deci$ = 31 ビット正整数

hex = 0x から始まる 32 ビット正 16 進整数

$const$ = $deci$ | $-deci$ | hex | $label$ | $label \pm deci$

$disp$ = 12 ビット正整数

$scale$ ::= 2 | 4 | 8

図 1 MIC 命令セット
Fig. 1 MIC instruction set.

トオーダーは未定義である．したがって，たとえばメモリに 32 ビット整数として書き込み，同じアドレスから 8 ビット整数として読み込んだ結果は未定義である．ただし 64 ビット整数のワードオーダーはリトルエンディアンと定義する．浮動小数点数のメモリ上での表現は ANSI/IEEE 754-1985 に準拠する．

図 1 中で T，S という記号で表されている文字は，命令に付加され，その命令が扱うデータ型を指示する．ε は空文字列であり，値のやりとりをしないことを意味する．i，l はそれぞれ 32 ビット整数と 64 ビット整数を意味する．f，d はそれぞれ 32 ビット浮動小数点数，64 ビット浮動小数点数を意味する．

3.3 レジスタ

レジスタには一般レジスタ，浮動小数点レジスタ，スタックポインタ（sp），フレームポインタ（ap），引数ポインタ（ap），大域的オフセットテーブル（got）の 6 種類がある．一般レジスタと浮動小数点レジスタの数は MIC1 ではそれぞれ 16 個，MIC2 では無限個ある．一般レジスタは 32 ビット整数の値を保持することができる．浮動小数点レジスタは 32 ビット浮動小数点数か 64 ビット浮動小数点数のどちらか一方を保持することができる．浮動小数点レジスタは暗黙のうちにどちらの数を保持しているか覚えており，正しい型の命令を使う必要がある．32 ビット浮動小数点数をレジスタに読み込み，それを 64 ビット浮動小数点数として書き出したときの結果は未定義である．

スタックポインタは関数呼び出しを行うときに引数をスタックで渡すときと，`alloca` のように関数に局所的なメモリ領域を確保するときに使われる．フレームポインタは関数に対して局所的な領域を使用するときに使われる．引数ポインタは，スタックで渡された関数の引数を参照するときに使われる．大域的オフセットテーブルは位置独立コード（PIC）を生成するときに使われる．フレームポインタ，引数ポインタ，大域的オフセットテーブルには代入することができない．

MIC 抽象機械には内部状態として浮動小数点数同士の比較の結果を保持する 4 ビットの値がある．各ビットはそれぞれ，(1) 比較された 2 つの値が等しい，(2) 最初の値が小さい，(3) 最初の値が大きい，(4) 順序がない，かどうかを表している．図 1 中で F という記号はこの情報を表す指示子である．F は条件分岐命令に付加され，どのような条件のときに分岐を行うのか指示する．

MIC 抽象機械は一般レジスタを比較した結果を表すフラグを持たない．その代わりに比較と分岐を同時に行う条件分岐命令を持つ．図 1 中で C という記号で表されているのが，どのような条件のときに分岐を行うか，という情報である．

3.4 スタックフレームの利用規約

図 2 が MIC 抽象機械のスタックフレームのレイアウトである．スタックフレームはスタックポインタ（sp），フレームポインタ（fp），引数ポインタ（ap）を

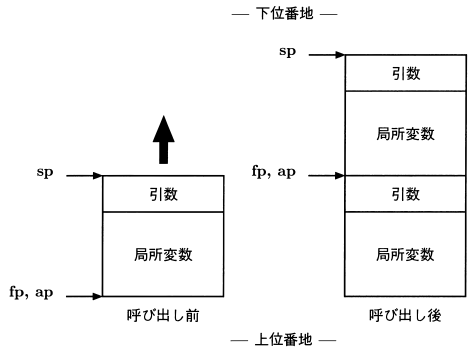

図 2 MIC のスタックフレーム
Fig. 2 Stack frame layout of MIC.

通してアクセスされる．これらのレジスタの値は関数のプロローグとエピローグで自動的に設定される．これらのレジスタは定められた目的以外には使うことができない．たとえばフレームポインタと引数ポインタは概念上同じアドレスを指しているが，引数ポインタを利用して局所領域にアクセスすることはできない．この理由は，MIC コードを機械語に変換するときに，対象となるアーキテクチャのスタックフレームの利用規約を尊重する形式に変換できるようにするためである．

関数呼び出しのときに最初の 4 ワードの引数と返り値はレジスタを使って渡される．それ以外の一般レジスタと浮動小数点レジスタを使って関数境界を越えて値の受け渡しを行うことはできない．また一般レジスタと浮動小数点レジスタは関数呼び出し後に関数呼び出し前と同じ値を保持している保証はない．

4. 実　　装

我々は単一メモリイメージを SPARC プロセッサと x86 プロセッサに対して実装した．実装には機種非依存なアセンブリ言語である MIC を利用した．そして，予備的な実験として，Eager Release Consistency に基づく page-based なソフトウェア DSM を実装し，その上で単一メモリイメージを構築した．実験では，SPARC プロセッサとして UltraSPARC を，x86 プロセッサとして Pentium II を用いたが，SPARC は SPARC V9 の仕様を満たすプロセッサ，x86 は i386 以降のプロセッサが提供する機能のみを用いている．

4.1 MIC の採用理由

これまでにも述べたように，我々は単一メモリイメージに関する実験を行うプラットフォームとして MIC を採用した．MIC コードを実行するときにはその機種固有の機械語に変換された後に実行される．この変換器の，(1) メモリアクセス，(2) 関数呼び出し，(3) 関数のプロローグとエピローグ部分の変換方式のみを変更することで単一メモリイメージを実現可能でことができる．これが我々が MIC を採用した理由である．

4.2 共通のデータ表現形式

表 1 にあるように，SPARC と x86 は 32 ビットの整数レジスタと 32 ビットおよび 64 ビットの浮動小数点数レジスタが使用可能であり，メモリのアドレス幅は 32 ビットである．浮動小数点数のフォーマットはどちらも IEEE 754 をサポートしている．データの表現形式で違いがあるのはバイトオーダーだけであり，SPARC はビッグエンディアン，x86 はリトルエンディアンを採用している．

我々は，共通のデータ表現形式として x86 のリトルエンディアンを採用した．したがって，x86 ではデータ表現形式を変換する必要はないが，SPARC ではメモリアクセス時にデータの表現形式を変換する必要がある．

SPARC プロセッサの計算機で必要なデータアクセス時のデータ表現形式の変換には `lda`/`sta` というロード/ストア命令を用いた．これらの命令はバイトオーダーを逆転して 32 ビットの整数データにアクセスすることができる．16 ビット，64 ビットの整数や 32 ビット，64 ビットの浮動小数点数のメモリアクセスにも同様の命令が用意されている．

4.3 間接コール・間接リターン

関数アドレスやリターンアドレスを計算機間でそろえるため必要な間接コールと間接リターンは各関数に**スタブ関数**を用意することによって実現した．スタブ関数は本来の関数の実行コードのアドレスにジャンプするだけの小さなコードであり，異機種間でも同じ固定サイズになるようにコードのサイズが調節してある．スタブ関数は計算機間で同じサイズであるので，計算機間で同じアドレスに配置することが可能となっている．したがって，プログラマには関数へのポインタの値としてスタブ関数のアドレスを見せるようにすれば，関数へのポインタの値は計算機間で同一のものとなる．現在の実装ではこのスタブ関数の大きさは 12 バイトである．**図 3** はスタブ関数の例である．これらは `sqrt` という名前の関数のスタブ関数であり，12 バイトのコードになっている．x86 の方は大きさを調節するために `nop` 命令が加えられている．本来の関数コードには `_smi.sqrt` という名前のラベルがつけられている．

図 4 は間接コールと間接リターンの例を示している．最初に関数 `f` の実行コードである `f_code:` を実行しているとする．次に関数 `g` を呼び出すと関数 `g` の

```
— SPARC —
sqrt:
    sethi %hi(_smi.sqrt),%g1
    jmpl %g1+%lo(_smi.sqrt),%g0
    nop
```

```
— x86 —
sqrt:
    jmp _smi.sqrt
    nop
    nop
    nop
    nop
    nop
    nop
    nop
```

図 3 スタブ関数
Fig. 3 Stub function.

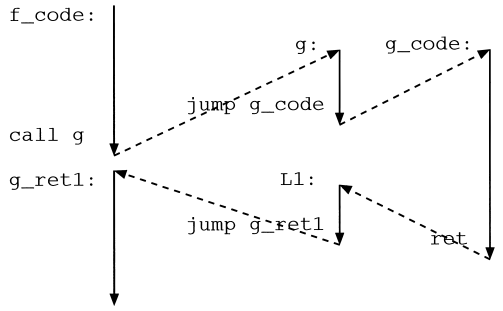

図 4 間接コール/リターン
Fig. 4 Indirect call and return.

スタブ関数 g: にジャンプする．このときリターンアドレスとして，実際のリターンアドレス g_ret1: ではなく，そのスタブ関数 L1: を保存する．スタブ関数 g: には関数 g の実行コードへジャンプする命令が書かれているので，関数 g の実行コード g_code: を呼び出すことができる．関数 g から呼び出した関数に戻る場合は，まず，リターンアドレスとして保存した L1: へジャンプする．L1: には実際のリターンアドレスである g_ret1: にジャンプする命令が書かれている．したがって，関数 f の中の正しい位置に戻ることができる．

スタブ関数を用意する方法以外に，関数の実行コードをテーブルにしておく方法もある．関数テーブルのインデックスを関数ポインタとしてプログラマに見せ，関数呼び出しの際にこのテーブルを引いて，実際のコードのアドレスを得る方法である．どちらの方法が良いかは一概にはいえないが，我々は MIC 変換器の改造を少なくおさえられるスタブ関数による間接コールの方式を採用した．

また，リターンアドレスに関しては，間接リターンを用いずに実際の実行コード中のリターンアドレスをスタック上に保存しておき，スレッドの移動時に変換する方法も考えられる．つまり，関数を呼び出している命令のアドレスをハッシュテーブルなどに登録しておき，仮想的なリターンアドレスを得られるようにしておく．スレッドの移動時には，スタック上に保存されているリターンアドレスからハッシュテーブルを使って仮想的なリターンアドレスを得て，さらに，仮想的なリターンアドレスから移動先の実際のリターンアドレスに変換するのである．この方法は，間接リターンよりもオーバヘッドが小さくなると考えられるが，現時点では実装が容易であることからスタブ関数で表す方法を採用した．

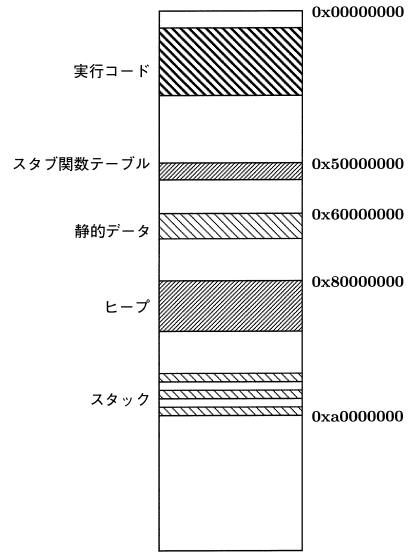

図 5 メモリレイアウト
Fig. 5 Memory layout.

4.4 メモリレイアウト

共通のメモリレイアウトは**図 5** のようになっている．静的に確保されるデータを 0x60000000 から，動的に確保されるヒープは 0x80000000 から確保している．また，間接コールと間接リターンで使われるスタブ関数のテーブルは 0x50000000 に配置した．スタブ関数の大きさは 12 バイトとし，余った部分は nop 命令で埋めてある．スレッドのスタックは 0xa0000000 から下位のアドレスの方向に確保していく．各スレッドのスタックは，同一計算機内はもちろん，異なる計算機のスレッドのスタックとも重ならないように配置する．こうすることで，スタック上のデータさえもソフトウェア DSM で共有することができる．

関数のスタブ関数や静的な変数を指定のアドレスに配置するのは，リンクエディタ（ld）の機能を用いて実現した．

4.5 スタックフレームレイアウト

異機種間のスレッドの移動を可能にする共通のスタックフレームのレイアウトは**図 6** のようになっている．退避領域というのは，MIC コードが使用する

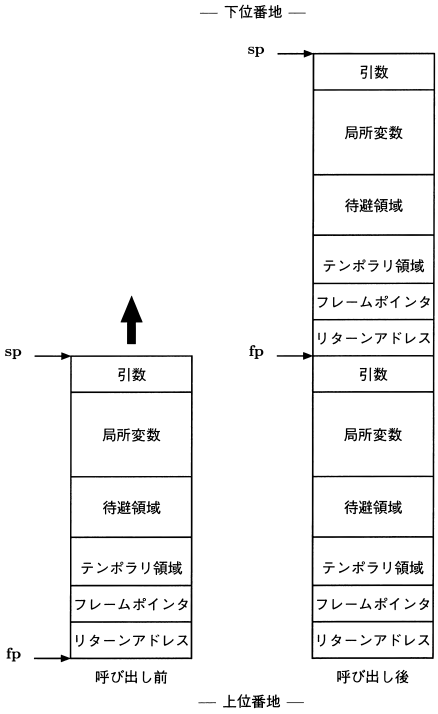

図 6 共通のスタックフレームのレイアウト
Fig. 6 Common layout of stack frame.

MIC レジスタの数が実際のプロセッサのレジスタ数を超えた場合に，使用しない MIC レジスタの値をスタックに退避するための領域である．

レジスタの退避領域はすべてのアーキテクチャで同一の大きさで確保される．この領域には機種依存のデータが書き込まれることになる．しかし，単一メモリイメージでは関数呼び出し時の MIC レジスタの退避は呼び出し側で行うことを仮定しているため，使用中の MIC レジスタの値がたとえスタック上に退避されていたとしても，関数呼び出しの際に適切な型のストア命令で局所変数領域に保存される．したがって，関数呼び出しが行われている時点ではスタック上に退避されている MIC レジスタがなくなるので，スレッドの移動が可能となる．

テンポラリ領域は，浮動小数点数と整数の変換などに用いるもので，24 バイトを確保してある．この領域は機種依存なデータが書き込まれている可能性があるが，一時的に用いられるだけあり，スレッドの移動が可能な関数の先頭で使用されることはない．スタックフレームの値は共通のデータ表現形式で書き込まれている．リターンアドレスの値も間接リターンに使われるスタブ関数のアドレスが共通のデータ表現形式で書き込まれている．

表 3 仮想レジスタのオフセット
Table 3 Offsets of pseudo registers.

MIC コード		機械語コード
$\mathrm{sp} + x$	$\rightarrow$	$\mathrm{sp} + x$
$\mathrm{fp} - x$	$\rightarrow$	$\mathrm{fp} - x - (n + 24)$
$\mathrm{ap} + x$	$\rightarrow$	$\mathrm{fp} + x$

SPARC の計算機では，機種本来のスタックのスタックポインタとして使わている o6 レジスタが指すスタック上の領域にコンテキストスイッチ時のレジスタの退避を行う．このような機種依存なデータをスタック上に置かないようにするため，我々の実装では，o6 レジスタではない別のレジスタを単一メモリイメージのスタックポインタとして使い，本来のスタックとは別の位置に新たにスタックを確保するようにした．したがって，o6 レジスタが指し示す領域はプログラムの実行中，つねに同じ場所を指すようになり，コンテキストスイッチ時のレジスタの退避も単一メモリイメージのスタック上には行われなくなる．

MIC 抽象機械のスタックフレームのレイアウト（図 2）と単一メモリイメージではスタックフレームのレイアウトは異なるため，MIC コードから機械語への変換時にスタックへのアクセスを正しく補正する必要がある．これは，コード変換時に MIC でスタックにアクセスする際に用いられる sp，fp，ap レジスタに対して適切なオフセットを加えることで実現できる．コード変換時に sp，fp，ap レジスタに加えるべきオフセットの値は**表 3** のとおりである（n は MIC レジスタの退避領域の大きさ）．

5. 実　　験

我々は，単一メモリイメージのオーバヘッドを単一プロセッサ上で測定した．そして，異機種の計算機間でソフトウェア DSM を用いて単一メモリイメージが正しく動くことを確認した．

5.1 単一プロセッサでのオーバヘッドの測定

単一メモリイメージを実現するために生じるオーバヘッドを測定するために，単一プロセッサ上で単一メモリイメージを用いて動作する逐次プログラムの実行時間の増加を測定した．実験環境には UltraSPARC 168 MHz（Solaris 2.6）と Pentium II 333 MHz（Linux 2.2）の計算機を用いた．実行時間は 10 回計測した平均値を採用した．

使用したプログラムは **LU**，**RAYTRACE**，**N-QUEEN**，**N-BODY**，**QSORT** の 5 つである．**LU** は double 型の 256×256 正方行列をピボット部分選択を行うアルゴリズムで LU 分解するものである．**RAY-**

表 4 SPARC での実行時間（ミリ秒）
Table 4 Elapsed time on SPARC (millisecond).

	LU	RAYTRACE	N-QUEEN	N-BODY	QSORT
GCC	2702.6	5379.4	5208.4	7984.0	5177.7
GCC (-mflat)	2710.8	5400.1	5560.3	8612.8	5578.4
無改造 MIC	2669.5	5394.9	6080.9	8297.2	5790.8
SMI	2708.4	5699.9	6587.8	9415.4	6376.6
直接コール&直接リターン	2677.7	5366.7	5996.6	8171.6	5678.0
間接コール	2695.6	5489.5	6235.5	8962.8	5929.2
間接リターン	2676.7	5578.8	6343.5	8753.3	6073.4

表 5 x86 での実行時間（ミリ秒）
Table 5 Elapsed time on x86 (millisecond).

	LU	RAYTRACE	N-QUEEN	N-BODY	QSORT
GCC	2502.2	4026.6	2732.8	5693.7	2906.6
無改造 MIC	3073.4	5838.4	5037.7	8816.1	4670.2
SMI	3155.9	5180.8	5048.5	8039.6	4784.8
直接コール&直接リターン	3170.2	5159.8	5070.9	7908.2	4693.7
間接コール	3156.5	5162.7	5098.3	7997.8	4730.0
間接リターン	3106.5	5166.7	5134.5	7962.0	4743.2

TRACE は 5 つの壁に囲まれた透明な球をレイトレーシング法を用いて描くものである．**N-QUEEN** は縦横 13 個の盤面で n クイーン問題を解くものである．**N-BODY** は重力ポテンシャルのもとで 100 個の質点の動きをシミュレートするものである．**QSORT** は 262144 個のランダムな整数をクイックソートのアルゴリズムを用いて並べ替えるものである．これらのプログラムは C で書かかれている．**LU** と **RAYTRACE** はループが主体のプログラムであり，一方，**N-QUEEN**，**N-BODY**，**QSORT** は再起呼び出しを多用するプログラムである．コンパイラの最適化オプションは `-O2` であり，x86 の GCC では `-malign-double` も指定している．**表 4** と **表 5** が実行時間の計測結果である．

図 7 と **図 8** は GCC でコンパイルしたコードの実行時間を 100%としたときの，無改造の MIC の変換器を用いて変換したコードと我々が改造した変換器を用いて変換したコードの実行時間を表している．さらに，図 7 には SPARC のレジスタウィンドウを用いずに GCC でコンパイルした場合の実行時間（gcc-mflat）も載せてある．

SPARC においては，無改造の MIC の実行時間の増加は最大で 16.7%であり，平均で 6.3%である．特に **LU**，**RAYTRACE** では GCC と比べて遜色がない性能が出ている．しかし，MIC はレジスタウィンドウを用いていないため，レジスタウィンドウが効果を出している **N-QUEEN**，**N-BODY**，**QSORT** では実行時間が 3.9%から 16.7%増加している．単一メモリイメージでの実行時間の増加は最大 26.4%，平均 14.7%であり，**LU** 以外は有意なオーバヘッドが生じている．

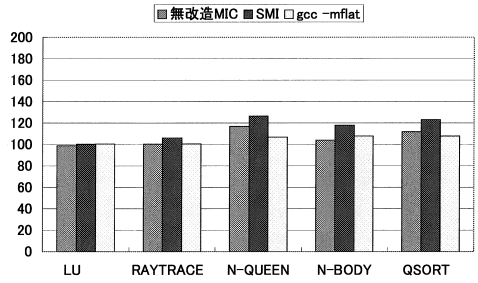

図 7 SPARC での実行時間（GCC の場合を 100%とする）
Fig. 7 Elapsed time on SPARC (relative to code by GCC).

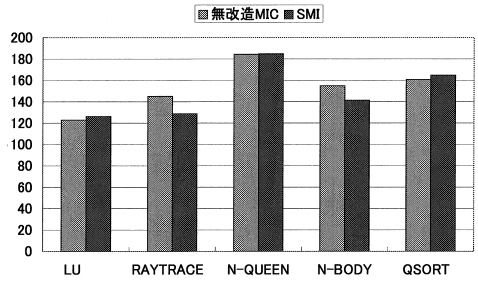

図 8 x86 での実行時間（GCC の場合を 100%とする）
Fig. 8 Elapsed time on x86 (relative to code by GCC).

x86 においては無改造の MIC の実行時間の増加は 22.8%から 84.3%，平均で 53.5%と大きなオーバヘッドになっている．これは，我々が利用した MIC1 の x86 コードを生成する変換器にはやや問題があり，多くの場合，8 個の一般レジスタのうち 6 個しか使わないためだと考えられる．最新のバージョンである MIC2 ではこの問題が解決されており，GCC と同程度の性能

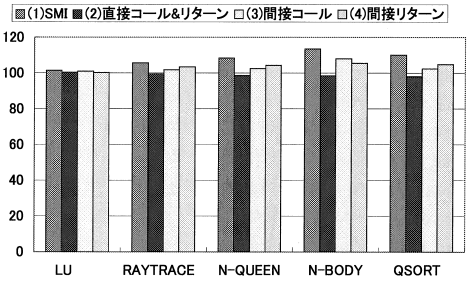

図 9 SPARC での実行時間（無改造 MIC の場合を 100%とする）
Fig. 9 Elapsed time on SPARC (relative to unmodified MIC).

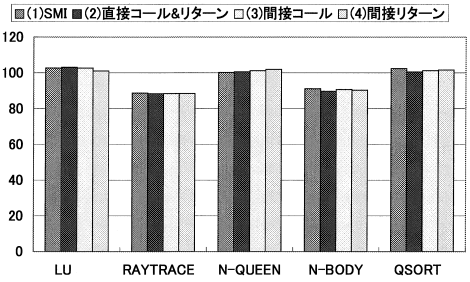

図 10 x86 での実行時間（無改造 MIC の場合を 100%とする）
Fig. 10 Elapsed time on x86 (relative to unmodified MIC).

が得られている．一方，x86 での単一メモリイメージのオーバヘッドも 26.1%から 84.7%，平均で 49.0%と大きなものとなっているが，無改造の MIC の場合と比べて速くなっているものもある．

次に，単一メモリイメージのオーバヘッドの内訳を調べるために単一メモリイメージの機能を部分的に有効にしたコードを用いて実行時間を測定した．**図 9** と **図 10** がそれぞれのプログラムの実行時間の結果である．数値は無改造の MIC の変換器を用いて変換したコードの実行時間を 100%としたときのそれぞれの実行時間の増減の割合である．4 種類の値のうちで最も左の（1）は共通のデータ表現形式，共通のスタックフレームのレイアウト，間接コールおよび間接リターンをすべて用いた場合の実行時間である．（2）は共通のデータ表現形式，共通のスタックフレームのみを用いた場合，（3）は共通のデータ表現形式，共通のスタックフレームのレイアウトのほかに間接コールのみを用いた場合，（4）は共通のデータ表現形式，共通のスタックフレームのレイアウトのほかに間接リターンのみを用いた場合である．

SPARC では単一メモリイメージを実現することによるオーバヘッドは最大 13.4%，平均 7.8%となっている．間接コールと間接リターンを行わない場合にはほとんどオーバヘッドが存在せず，逆に若干速くなっている場合もある．これは，バイトオーダーを逆転してメモリアクセスする **lda/sta** などにかかる時間と普通のメモリアクセス命令にかかる時間にほとんど差がないためだと考えられる．一方，間接コールと間接リターンは，関数呼び出しが少ない **LU** 以外では，有意なオーバヘッドが出ている．間接コールは最大 8.0%，平均 3.1%のオーバヘッドになっている．間接リターンは最大 5.4%，平均 3.6%のオーバヘッドになっており，3 種類のオーバヘッドの内訳の中では最大になっている．

x86 の場合では，単一メモリイメージを実現することによるオーバヘッドは最大 2.4%であり，逆に 11.3%ほと速くなっているものもある．高速化の主な原因は，共通のスタックフレームを採用したことによって，より効率の良い関数呼び出し規約に変わったためであると考えられる．間接コールと間接リターンのオーバヘッドもそれぞれ 1%程度あるが，SPARC と比べると小さな値におさえられている．

SPARC と x86 のどちらの場合も，共通のデータ表現形式と共通のスタックフレームのレイアウトを採用することによるオーバヘッドはほとんどない．オーバヘッドの大半は間接コールと間接リターンにあり，間接リターンの方がより大きな値になっている．したがって，これらのオーバヘッドを削減できれば，単一メモリイメージのオーバヘッドをさらに減らすことが可能である．現在の実装では間接リターンを採用しているが，スレッドの移動時にスタック上のリターンアドレスの値を正しく変換する方法もある．この方法ではスレッド移動時以外にはオーバヘッドはない．間接コールに関しては，関数のポインタはプログラマに見せている値であるので，間接コールに類する抽象化がどうしても必要であり，その分のオーバヘッドは避けられないと考えられる．ただし，静的に関数のアドレスが分かっている場合にはスタブ関数を介さずに関数のコードに直接ジャンプする最適化も考えられる．しかし，この方法は動的に関数がバイトコードから実行コードに変換されるような環境では，あらかじめ関数コードのアドレスを知ることが不可能であるため，たとえ，プログラム上では静的なコールが行われていたとしても，スタブ関数を返す必要がでてくる．スタブ関数以外の関数ポインタの抽象化の方法としては，関数実行コードのアドレスをテーブルにしておき，関数テーブルのインデックスを関数ポインタとしてプログラマに見せる方法がある．関数呼び出しの際には，テーブルを引いて，実際のコードのアドレスを得るのである．我々はまだこの方法のオーバヘッドを測定し

表 6 **RAYTRACE** の実行時間（秒）
Table 6 Elapsed time of **RAYTRACE** (second).

DSM	SPARC	x86
0.894	1.268	1.142

ていないので，どちらの方法が良いかは現時点では分からない．

5.2 異機種間のソフトウェア DSM での実行結果

我々は予備的な実験として，単一メモリイメージがソフトウェア DSM 上で正しく動くことを確認するために，Eager Release Consistency（ERC）[3] に基づくソフトウェア DSM を UDP を用いて実装した．ERC ではスレッド間の同期命令を実行したときに共有メモリの一貫性の維持のための計算機間の通信が行われ，同期命令を実行した後では，共有メモリが正しく更新されていることが保証される．したがって，メモリアクセス時に一貫性の維持のための通信が行われる Sequential Consistency よりも性能が良いといわれている．

そして，我々は，UltraSPARC（168 MHz）と Pentium II（333 MHz）の 2 台の計算機を用いて **RAYTRACE**（問題サイズは 5.1 節とは異なる）を動かし，単一メモリイメージで正しく動作することを確認した．

表 6 は **RAYTRACE** を SPARC と x86 の 2 台の計算機間でソフトウェア DSM を用いて実行した場合の実行時間（DSM）と，SPARC および x86 の計算機を用いて単一プロセッサで実行した場合の実行時間（SPARC，x86）である．ソフトウェア DSM を用いた実行時間は，理論上の最適な実行時間よりも 40%程度大きい値となった．

6. 関 連 研 究

Mermaid [9] は異機種間ソフトウェア DSM を初期に実現したシステムである．このシステムでは，ヒープ上にデータを確保するときにプログラマが明示的にその型を指定する．そして，ページに 1 つの型のデータのみ持つようにし，各ページが持つ型をテーブルで管理しておく．そして，そのテーブルをもとにデータ通信時にページ上のデータの表現形式を変換している．しかし，この方法では型キャストを正しく扱うことができない．また，この枠組みでは様々な型のデータが存在するスタックを共有することができない．実際 Mermaid ではスレッドを実行の途中で移動させることはできない．

CVM [7] でも異機種間ソフトウェア DSM を実現している．その方法は，共有メモリにのせるデータを `int`，`float`，`double` の型のデータに制限し，データ通信時にネットワークバイトオーダーに変換してから通信を行うというものである．型はプログラマがヒープにデータを割り当てるときに明示的に指定する．CVM は異機種間のスレッドの移動も実現している．これはスレッドを移動できる場所をプログラムの中で 1 カ所に制限し，移動するときにスタック上にある変数はすべてプログラマが明示的に型を指定することで実現している．

Java/DSM [8] はソフトウェア DSM を使って分散した Java VM に共有メモリを提供するシステムである．任意のアドレスのデータの型を判別できるように Java VM を改造し，データの表現形式は通信時に行っている．このシステムでは Java を使うことによって，異機種性をプログラマから隠している．しかし，他の言語処理系と協調動作させるのは困難である．

7. まとめと課題

本論文では，ソフトウェア分散共有メモリを用いて異種計算機間のメモリ上のヒープとスタックの共有を特定の言語に依存せずに実現する枠組みである単一メモリイメージを提案した．単一メモリイメージは，機種ごとのデータ表現形式の違いを解決するために，メモリ上のデータを計算機固有のデータ表現形式ではなく，計算機間で共通のデータ表現形式を用いて表現する．そして，メモリアクセスのアセンブリ命令の型を基にデータ表現形式を変換するコードを各メモリアクセス命令に付加する．また，共通のメモリレイアウトとスタブ関数による間接コールによって，異機種間でのポインタの値の違いを解決している．さらに，異機種間でのスレッドの移動を可能とする異機種間のスタックの共有を機種非依存なスタックフレームのレイアウトとスタブ関数による間接リターンを用いて実現した．

我々は単一メモリイメージを SPARC と x86 プロセッサの計算機上に実装し，そのオーバヘッドを測定した．GCC が生成するコードと比較した場合，SPARC でのオーバヘッドは平均 14.7%程度，x86 では平均 49.0%程度であった．

今後は，単一メモリイメージによる異機種間でのスタック共有を利用して，異機種間のスレッドの移動を実現する予定である．計算機間のスレッド移動の利用例には，動的に計算機やネットワークの負荷を測定してスレッドを適切に移動させることで全体の性能を向上させることなどがある．また，MIC のコードもメ

モリ上に置いて計算機間で共有し，動的に機械語を生成する枠組みも提供する予定である．これらを実現することによって，より透過的な異機種間の分散計算環境を実現できる．

謝辞 多くの助言をしていただいた新情報処理開発機構の原田浩氏と高橋俊行氏に心から感謝いたします．また，論文に対し有益なコメントをしてくださった東京大学の田浦健次朗先生，増原英彦先生，遠藤敏夫氏，大山恵弘氏，田中義純氏にも深く感謝いたします．

参 考 文 献

1) Aridor, Y., Factor, M. and Teperman, A.: cJVM: A Single System Image of a JVM on Cluster, *International Conference on Parallel Processing 99* (*ICPP 99*) (1999).
2) IEEE Task Force on Cluster Computing: Cluster Computing White Paper, Section 4 (2000). http://www.dcs.port.ac.uk/~mab/tfcc/WhitePaper/
3) Keleher, P.: Lazy Release Consistency for Distributed Shared Memory, Ph.D. Thesis, Rice University (1995).
4) Li, K. and Hudak, P.: Memory Coherence in Shared Virtual Memory Systems, *ACM Trans. Comput. Syst.*, Vol.7, No.4, pp.321–359 (1989).
5) Scales, D.J., Gharachorloo, K. and Thekkath, C.A.: Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory, *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.174–185 (1996).
6) Sunderam, V.S.: PVM: A framework for parallel distributed computing, *Journal of Concurrency: Practice and Experience*, pp.315–339 (1990).
7) Thitikamol, K. and Keleher, P.: Thread Migration and Load Balancing in Heterogeneous Environments, *5th Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers* (2000).
8) Yu, W. and Cox, A.: Java/DSM: A Platform for Heterogeneous Computing, *ACM 1997 Workshop on Java for Science and Engineering Computation* (1997).
9) Zhou, S., Stumm, M., Li, M. and Wortman, D.: Heterogeneous distributed shared memory, *IEEE Trans. Parallel and Distributed Systems* (1991).
10) 関口龍郎，米澤明憲：異機種間モーバイル計算のためのコード表現とその実装，情報処理学会プログラミング研究会 (2000).

(平成 12 年 7 月 14 日受付)
(平成 13 年 1 月 22 日採録)

上田 陽平

1977 年生．2000 年東京大学理学部情報科学科卒業．現在東京大学大学院理学系研究科情報科学専攻修士課程に在学中．主に並列・分散プログラミングの研究に従事．

山本 泰宇

1974 年生．1999 年東京大学理学部情報科学科卒業．現在東京大学大学院理学系研究科情報科学専攻博士課程に在学中．主に並列・分散言語の研究に従事．

関口 龍郎

1970 年生．1998 年東京大学大学院理学系研究科博士課程修了．理学博士．1998 年より日本学術振興会未来開拓事業特別研究員．主にモーバイル計算等の研究に従事．

米澤 明憲（正会員）

1947 年生．1977 年 Ph.D. in Computer Science（MIT）．1989 年より東京大学理学部情報科学科教授．2000 年より東京大学大学院情報学環教授（兼任）．2000 年より国立情報学研究所客員教授．超並列・分散ソフトウェアアーキテクチャ，等に興味を持つ．編著書に「モデルと表現」（岩波書店),「ABCL」（MIT Press）等がある．ACM Transaction on Programming Languages and Systems 副編集長，日本ソフトウェア科学会理事長等歴任，ACM Fellow．