システムソフトウェアと
オペレーティング・システム 70－4
（１９９５．８．２２）

世界のDAGを利用した投機的makeの実現

根路銘 崇 <nero@ocean.ie.u-ryukyu.ac.jp>
新城 靖 <yas@ie.u-ryukyu.ac.jp>
當眞 聡 <ha@ocean.ie.u-ryukyu.ac.jp>
溝淵 雅也 <miz@ocean.ie.u-ryukyu.ac.jp>
喜屋武 盛基 <kyan@ie.u-ryukyu.ac.jp>
翁長 健治 <onaga@ie.u-ryukyu.ac.jp>

琉球大学　情報工学科
〒903-01　沖縄県西原町千原1番地
電話：098-895-2221　内線3266
Ｆａｘ：098-895-2688

概要　投機的処理とは，利用されるか利用されないか確定される前に実行を開始する処理である．余剰計算機資源を利用した投機的処理は，利用者や利用者プロセスにとっての見かけの処理時間を大幅に短縮することを可能にする．我々は，世界という概念を取り入れた投機的処理支援OSを実現している．世界は，DAGで表現される．我々は，投機的処理支援OS 上で動作するアプリケーションとして投機的makeを実現中である．本論文では，投機的makeがどのようにして世界のDAGを操作して投機的処理を行うかについて述べる．投機的make実現の特徴は，ファイルの依存関係を表わすDAGと相似の世界のDAGが作られる点にある．また，投機的makeのシミュレーションとその結果について述べ，投機的makeの性能予測を行う．

The implementation of Speculative Make using a DAG of Worlds

Takashi Nerome <nero@ocean.ie.u-ryukyu.ac.jp>
Yasushi Shinjo <yas@ie.u-ryukyu.ac.jp>
Hajime Toma <ha@ocean.ie.u-ryukyu.ac.jp>
Masaya Mizobuchi <miz@ocean.ie.u-ryukyu.ac.jp>
Seiki Kyan <kyan@ie.u-ryukyu.ac.jp>
Kenji Onaga <onaga@ie.u-ryukyu.ac.jp>

Department of Information Engineering
University of the Ryukyus
Nishihara, Okinawa 903-01, Japan
Phone: +81 98 895 2221 Ext.3266
Fax: +81 98 895 2688

Abstract　Speculative processing is processing that is started eagerly before it is known to be required. Speculative processing using surplus computer resources makes it possible to reduce apparent execution time for users. We are implementing an operating system that supports speculative processes. In the operating system, the idea of *worlds* is introduced. A world is a container of files and processes. Worlds construct a DAG. As an application of the operating system, Speculative Make is being implemented. In this paper, how Speculative Make manipulates worlds is shown. We made a simulation of Speculative Make , we show its result and we estimate performance of Speculative Make.

1 はじめに

近年，高性能計算機がネットワークに結合された環境が普及してきた．また，1000プロセッサ規模の高並列計算機が作られるようになってきた．このような環境では，多くの計算機資源が遊んでいる．前者の環境では，CPU資源は，1日に数パーセントしか使用されていないことがある．余剰計算機資源を利用することによって，個々のアプリケーションの実行速度が格段に向上する事が見込まれる．

投機的処理(speculative processing)とは，利用されるか利用されないか確定される前に実行が開始される処理である[1][5][7]．余剰計算機資源を利用して投機的処理を行なうことにより，利用者や利用者プロセスにとっての見かけの処理時間が短縮される．

我々は，投機的処理を支援するオペレーティング・システム(投機的処理支援OS)を実現している[1][2][4]．我々が扱う投機的処理は，事前実行[5]と値の書換え[1]を伴うものである．投機的処理の対象は，ファイルという記憶領域とそれを扱うプロセスやプロセスの集合である．

投機的処理支援OSは，**世界**という概念により，投機的処理を行なうアプリケーションを支援している．投機的処理を行なうアプリケーションからの要求を検討することによって，OSの資源割り当てやファイル・システムにおけるさまざまな問題を洗い出している．そのOS上で動く重要なアプリケーションは，投機的makeである．

makeは，一般に処理に時間がかかり，高速化が要求されている．makeに投機的処理機能を加えることで，既存のmakeに比べその見かけの処理時間が短くなることが予測される．投機的makeは，世界を利用することによって投機的処理を実現する．本論文では，世界を利用した投機的makeについて述べる．そして既存のシステム上で投機的makeのシミュレーションを行い，投機的makeの性能の予測を行なう．

我々は，これまで文献[1]において投機的処理を支援するオペレーティング・システムの基本構想を述べている．文献[1]では，多重プログラミング環境における，プロセス単位の投機的処理を支援するために解決すべき問題について議論している．また，文献[2]ではプロセスの操作と世界オブジェクトについて述べている．文献[4]では，投機的処理支援OSにおけるファイル・システムについて述べている．世界操作による投機的makeの基本的な実現方法を，文献[3]で述べている．文献[1]では，世界を入れ子構造で示した．その後，世界をDAGに拡張した．この論文では，DAGに拡張された世界を用いる投機的makeの実現について詳しく述べる．

2 投機的処理と世界

投機的処理(speculative processing)とは，利用されるか利用されないか確定される前に実行を開始する処理である．これに対して，投機的処理ではない処理は，**必須の処理**(mandatory processing)と呼ばれる．投機的処理は，将来使われずに無駄になる可能性のある処理であるのに対し，必須の処理は，要求されている処理で，無駄になることがない．

投機的処理の結果が要求された時，投機的処理は，必須の処理に変わる．投機的処理の結果が不要であると確定した時，投機的処理は，**的外れ**(irrelevant)に変わる．

2.1 投機的処理を実現するための技術

投機的処理を実現するためには，さまざまな問題点が生じてくる[1]．投機的処理を実現するためには，次の３つの技術が必要となる[1]．

（１）**隔離** ：投機的処理と必須の処理の相互作用を止める技術．投機的処理は，的外れになることがある．その影響によって投機的処理を行わなかったときの結果と違う結果を与えてはならない．よって，利用されることが確定されるまで，投機的処理の経過や結果を必須の処理から見えないようにしなければならない．

（２）**出現** ：投機的処理を必須の処理にみせる技術．投機的処理をいつまでも隔離していてはならない．投機的処理の結果が要求された時には，速やかにみせなければならない．

（３）**中止** ：的外れの処理を消す技術．的外れの処理を実行することは，資源を無駄にするものである．よって，的外れの処理を速やかに消さなければならない．

我々が扱う投機的処理は，事前実行[5]と値の書換え[1]を伴うものである．投機的処理の対象は，ファイルという記憶領域とそれを扱うプロセスやプロセスの集合である．

2.2 投機的処理支援OSと世界

我々が開発している投機的処理支援OSは，世界という概念を用いて投機的処理を支援する．世界とは，ファイルやプロセスを入れるための箱である．１つの世界は，単一のファイル・システムを持っている．世界は，親子関係を持つ．すべての世界は，DAGを形成する（図１）．プロセスは，自分の属する世界にあるファイルに加えて，その親の世界にあるファイルを参照することができる．プロセスは，子孫の世界のファイルを参照することはできない．

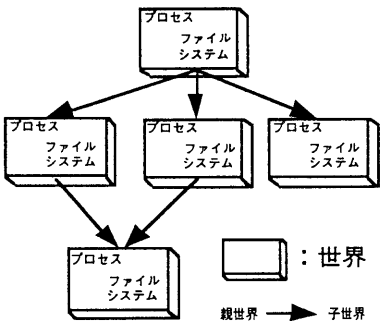


図１　世界のDAGの例

2.3 世界の操作

投機的処理支援OSが提供している，世界を操作するためのシステム・コールを，次に示す．

生成：wd=world_create(親の世界のリスト)

新たに別の世界を作る．返り値として新たに生成された世界の世界記述子wd(world descriptor)を返す．

融合：world_merge(世界１，世界２)

世界２を世界１に融合する．２つの世界に同じ名前のファイルが存在した場合，ファイルの最終更新時刻の新しいファイルが残され，古いファイルは消去される．世界２で実行中のプロセスも世界１に移される．今まで異なる世界に属していたため，遅延されていたプロセス間通信の遅延が，解除される．

削除：world_delete(世界)

目的の世界を世界の内容ごと削除する．

上記のシステム・コールに加えて，異なる世界の中でプロセスを生成するためのシステム・コールprocess_create()がある．返り値は，プロセスIDである．

表１　投機的処理に必要な技術と世界の操作の関係

投機的処理に必要な技術	→	世界の操作
隔離	→	生成
出現	→	融合
中止	→	削除

2.1節で述べた投機的処理を実現するために必要な技術は，世界の操作を用いて実現できる（表１）．

投機的処理は，子孫の世界の中で行われる．投機的処理を行なうために生成された世界を**投機の世界**と呼ぶ．

これに対して，利用者から見えるファイルやプロセスの属する世界を，**必須の世界**と呼ぶ．投機的処理を行うプロセスやその出力ファイルは，必須の世界の子孫である投機の世界につくられる．

投機的処理は，**管理プロセス**によって進められる．管理プロセスは，投機的処理を行なうプロセスを，新たに世界を生成して，そのなかで実行する．投機的処理を行うプロセスは，結果を投機の世界のファイルに出力する．投機的処理の結果が利用されることが確定された場合，管理プロセスは，必須の世界に投機の世界を融合する．すると，投機的処理の経過や結果が，利用者や利用者プロセスから見えるようになる．一方，投機的処理を行うプロセスやその結果が利用されないことが確定した場合，管理プロセスは，投機の世界を削除する．すると，結果のファイルや投機的処理を行うプロセスが削除される．

2.4 木からDAGへの拡張

文献[1]では，１つの投機の世界と世界の入れ子構造（木構造）を用いて，投機的処理を行う方法を提案した．その後，木構造をDAGへ拡張した．DAGの利点を以下にまとめる．

・１つの投機の世界を用いる場合との比較

この場合，投機の世界から，その中の一部のファイルやプロセスを必須の世界に移すことが必要になる．これを実現することは，非常に難しい．特に，ファイルの書き込み中のプロセスや，子プロセスを持つプロセスを移動することは，難しい．DAGの場合には，世界の融合を用いることで世界単位でファイルやプロセスを必須の世界に簡単に移動することができる．

・世界の入れ子構造を用いる場合との比較

入れ子構造の世界は，１つの親しか持たない．複数のファイルの更新を扱うために，深い入れ子を作る必要があった．DAGの場合には，３章で述べる投機的makeなどのアプリケーションでは，複数の親を持たせることで無駄な実行を減らすことができる．

3　投機的make

我々は，UNIXのmakeアプリケーションに投機的処理を実行する機能を組み込むことにした．アプリケーションとしてmakeを選んだ理由は，高速化が要求されているからである．既にmakeの並列化の研究は，盛んに行われている [6][8]．我々は，並列処理に投機的処理を加え，更に高速化をめざす．

3．1　makeにおける投機的処理可能部分

makeの主な目的は，実行形式のファイルを作成することである．その主な作業は，コンパイルとリンクである．makeは，実行形式のファイルを作成するときに，いくつもの関連ファイルをコンパイルして，その後リンクの処理を行う．これらの処理は，Makefileに記述されている規則と，予め定義されている規則を参照して進められる．

既存のmakeは，利用者からのシェルに対する"make"入力によりシェル・プロセスから起動される．例えば，ファイルの更新を検知してcc（Cコンパイラ）を実行する場合を考える．これは，シェルの動きを含めて次のif文で示される．

```
if(利用者からの"make"入力)
{
        if(ファイルの更新)
                cc;
}
```

投機的makeの実行は，次のif文で表現される．

```
if(ファイルの更新)
{
        if(利用者からの"make"入力)
                cc;
}
```

投機的makeは，内側のif文に対して投機的処理を行う．利用者からの"make"入力に先立ち，隔離された場所でccを実行する．そして後に利用者からの"make"入力があったときに出現させる．こうすることにより，利用者には短い時間で処理が行われているかのように見せかけることができる．

図２に，それぞれ同じコンパイル作業を行う従来のmakeと投機的makeの処理を示す．図２aの従来のmakeでは，利用者が"make"と入力するとMakefileに記載された処理を開始する．図２bの投機的makeにおける処理の実行では，ファイルの変更を検知してMakefileに記載された処理を開始する．この処理は，利用者が"make"と入力する以前に終了している．

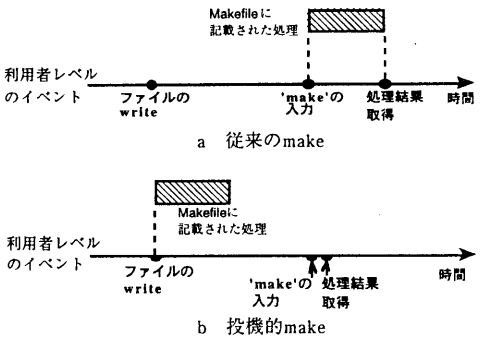


図２　従来のmakeと投機的makeの処理

利用者が"make"と入力後，従来のmakeに比べて短い時間でその処理の結果を知ることができる．

3.2 世界を用いた投機的makeの実現

投機的makeは，子世界を生成して投機的処理を行なう．投機的makeは，投機的処理支援OSの提供する，世界の生成，世界の削除，世界の融合の機能を用いる．投機的makeは，世界の操作に関して，2.3節で述べた管理プロセスとなる．

投機的make実現の基本方針は，２つある．１つは，更新するファイルごとに，投機の世界を生成することである．もう１つは，ファイルの依存関係に対応させて，世界の親子関係をつくることである．

3.3 投機的makeの構成

投機的makeの内部の構成を図３に示す．投機的makeは，常駐型のプロセスである．makeコマンドは，利用者が"make"と入力したという信号を常駐型プロセスへ送るだけである．

DAG生成モジュールは，Makefileから処理の対象のファイル，ファイルの依存関係，及びコマンド群の情報を入手する．DAG生成モジュールは，ファイルの依存関係を基にファイルのDAGを生成する．ファイルの依存関係を示すDAGの例を図４に示す．図４において，各ノードは，Makefileに記述され，最終更新時刻を調べるべきファイルに対応する．ここで，ファイルa.cに対応しているノードは，「a.cのノード」と表現する．各枝は，ファイルの存関係を

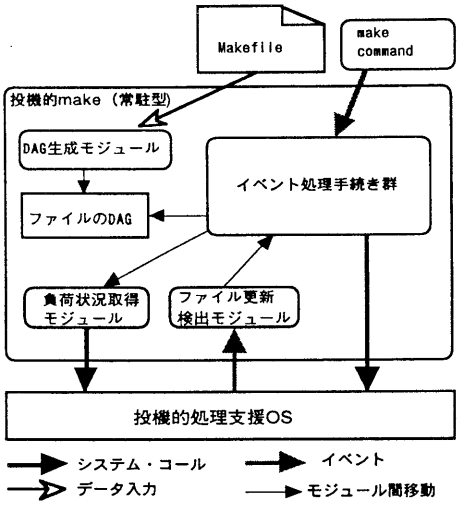


図３　投機的makeの内部の様子

示している．枝において，依存しているノードに対して，矢印が向いている．図４のa.c，b.c，c.c，d.c，e.h，f.hは，C言語のソース・ファイルである．a.o，b.o，c.o，d.oは，オブジェクト・ファイルであり，a.outは，実行ファイルである．

各ノードには，次の情報が含まれる．

(1)ファイル名

(2)そのファイルを生成する処理

(3)投機の世界の有無（投機の世界があればその世界IDで示す）

a.oのノードは，{a.o，cc -c a.c，no}という情報を持つ．このノードの情報より，a.oのファイルは，cc -c a.cの処理によって生成されるということが判る．a.oのノードは，投機の世界を持っていないということも判る．

図３の負荷状況取得モジュールは，投機的処理支援OSから負荷情報を取得する．ファイル更新検出モジュールは，投機的処理支援OSからファイルの更新の情報を受ける．

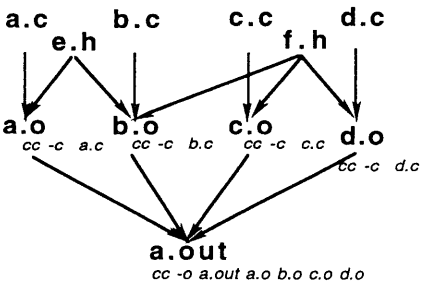


図４　ファイルのDAGの例

イベント処理手続き群は，次の２つの重要なイベントにより起動される手続きを含む．

(1)ファイル更新イベント処理手続き

ファイル更新イベントを受け付けると，ファイルのDAGから，変更されたファイルに対応するノードを検索する．そして，そのファイルに依存しているファイルのノードのリストを得る．そして各ノードについて子世界の有無を調べる．子世界があれば削除する[world_delete()]．そして，各ノードについて子世界を生成し[world_create()]，その中で必要な処理を行う[process_create()]．処理を行う時には，負荷状況取得モジュールに負荷の状況を問い合わせて，その処理の実行を開始すべきかどうかを判断する．依存関係のある処理の場合，その終了を待つ．

(2)利用者が'make'と入力した

世界のDAGを参照して必要な子世界を必須の世界へ融合する[world_merge()]．

３．４　投機的 make の実行例

図４に示されたファイルのDAGを例に，投機的makeの動きを示す．最初は，各ファイルの最終更新時刻には矛盾がないものとする．この場合，各ファイルに対する投機の世界はない．負荷状況取得モジュールは，定期的に資源の負荷状況をオペレーティング・システムから入手する．

ファイルの更新がない時には，図５に示すように必須の世界だけが存在する．必須の世界には，全てのファイルが存在する．図４においてf.hのファイルの更新が起きた場合を考える．すると，図５の状態から，図６に示すような世界のDAGを以下のようにして作成する．

投機的makeの内部では，ファイル更新イベント処理手続きが起動される．そして，負荷状況により投機的処理を実行するかどうかを判断する．実行しないと判断した場合は，何もしない．実行すると判断した場合，ファイル更新イベント処理手続きは，まず，以下のように必要な投機の世界を生成する．ファイル更新イベント処理手続きは，ファイルのDAGを参照し，更新されたファイルに対応したf.hのノードから，f.hに依存しているファイルを調べる．f.h

に依存しているファイルは，b.o，c.o，d.oである．次にb.o，c.o，d.oに対応しているそれぞれのノードから，それらに依存しているファイルを調べ，a.outを得る．a.outのノードの情報からは，a.outに依存しているファイルはないことが判る．この時，b.o，c.o，d.o，a.outの４つのノードについて投機の世界の有無を調べる．今の場合，各ノードには，投機の世界がない．次に，ファイル更新イベント処理手続きは，それぞれのファイルに対する子世界をworld_create()によって生成する．ここで，b.o，c.o，d.oに対応した各世界は，必須の世界の子世界となる．図６では，必須の世界の世界IDが0，子世界の世界IDはそれぞれ1,2,3である．a.outに対応した世界は，世界IDが1,2,3の世界の子世界として生成される．この世界の世界IDは4である．

投機の世界を生成すると，ファイル更新イベント処理手続きは，b.o,c.o,d.oを生成するためのコマンドをそれぞれのファイルに対応している世界の中で実行する．この時，2.3節で述べたprocess_create()システム・コールを発行する．例えば図４において，b.oを生成するためのコマンドは，[cc -c b.c]である．

ファイル更新イベント処理手続きは，b.o,c.o,d.oを生成するコマンドの実行の終了を確認する．全てのコマンドの実行終了後，a.outを生成するコマンドを，世界IDが４の世界の中で実行する．

a.outを生成するコマンドの実行中に，ファイル更新イベント処理手続きが新たにc.cのファイルの更新を検出した場合を考える．この手続きは図７に示すような世界のDAGを以下のようにして作る．

ファイル更新イベント処理手続きは，図４のファイルのDAGからc.cのノードを調べる．ファイル更新イベント処理手続きは，c.cに依存関係のあるファイルc.oとa.outのそれぞれのノードを調べ，投機の世界があることを知る．すると，world_delete()システム・コールを発行し，それらの世界を削除する．図７で示すように，世界 IDが4の世界を削除し，世界IDが2の世界を削除する．そして新たにc.o，a.outに対して，それぞれ世界を生成し処理を行う．このとき新たに生成された世界の世界IDは，それぞれ5,6である．

その後，利用者が"make"と入力したとする．すると，図８に示すような世界のDAGを以下のようにつくる．

投機的makeは，子孫の世界を必須の世界に融合するため，世界IDの0と6を引き数にしてworld_merge()システム・コールを発行する．これによって投機的makeは，処理の経過や結果を利用者に示すことができる．

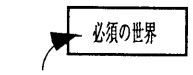


図５ 投機的makeの世界操作1
（投機の世界がない）

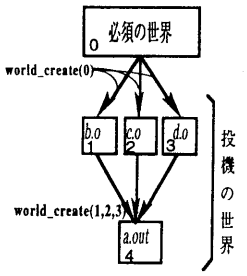


図６ 投機的makeの世界操作2
（図５の状態でf.hが更新した）

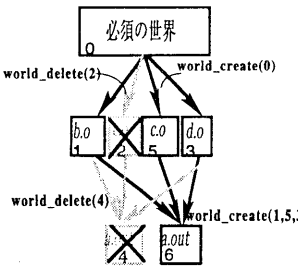


図７ 投機的makeの世界操作３
（図６の状態でc.cが更新した）

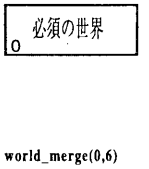


図８ 投機的makeの世界操作4
（利用者の"make"入力後）

4 投機的makeのシミュレーション

投機的makeの実現に先立ち，投機的makeシミュレータを作成した．これは，既存のmakeやエディタのトレースから投機的makeの動きをシミュレートする．投機的makeシミュレータによって，利用者にとっての見かけの処理時間がどれだけ短くなるかを予測することができる．この章では，シミュレーションの方法と，投機的makeシミュレータのアルゴリズムについて述べる．そして，投機的makeシミュレータによって得られた結果を基に，投機的makeの性能を予測する．

4.1 シミュレーションの方法

既存のmakeを改変して，利用者の動き（makeと入力した時刻）とmakeから起動されたコマンドの，実行開始時刻と終了時刻をログ・ファイルに書き出すようにした．そしてエディタを改変して，ファイルの更新時刻とそのファイル名をログ・ファイルに書き出すようにした．投機的makeシミュレータは，

そのログ・ファイルを入力し，投機的makeの動きをシミュレートしたものを出力する．

4.2 投機的makeシミュレータの入力と出力

投機的makeシミュレータの入力は，既存のmakeの動きと利用者の動きをトレースしたログである．このログには，以下の情報が含まれている．

・ファイルの更新時刻とそのファイル名
・利用者が“make”と打った時刻
・makeがMakefileに記載された処理の実行を開始した時刻と，それらの処理が終了した時刻

図9に投機的makeシミュレータの実行例を示す．

図9の入力では，利用者が，時刻19:23:26.0にファイルselect.cを書き換えている．その後，利用者は同じファイルを19:23:35.0に書き換えている．利用者は,時刻19:23:40.5に"make"と入力している．図9の入力において4行と6行，5行と7行は，makeがMakefileに記載された処理を開始した時刻と終了した時刻である．

投機的makeシミュレータの出力は，投機的makeが実際に動作したことを想定したログである．図9の出力の1行目で，利用者が時刻19:23:26.0にファイルselect.cを書き換えたことを検出している．これに対応して2行目でworld_create()システム・コールを発行している．3行目は，時刻19:23:35.0に同じファイルselect.cの書き換えを検出している．これに対応して，4行目と5行目でworld_delete()とworld_create()を発行している．6行目は，時刻19:23:40.5に利用者が"make"と入力したことを示している．7行と8行，9行と10行は，処理の実行時間を計算している．11行目では，world_merge()システム・コールを発行している．Run-timeは，利用者が"make"と入力した時刻を基準とした処理終了時刻である．-3.4とは，利用者が"make"と入力する3.4秒前に，処理が終了していることを示している．この時間は，出力のログの3行目の時刻に8行目と10行目で得られた処理の実行時間を加え，6行目の時刻を引く事によって得られる．

```
                :
1: 19:23:26.0 /home/h9/rei/NNTP/select.c
2: 19:23:35.0 /home/h9/rei/NNTP/select.c
3: 19:23:40.5  make
4: 19:23:40.7   ->cc -c -g -DD select.c
5: 19:23:42.0   -<cc -c -g -DD select.c
6: 19:23:42.0   ->cc article.o client.o disp.o
7: 19:23:42.8   -<cc article.o client.o disp.o
                select.o group.o -o group
                :
```

入力（既存のmakeと利用者の動き）

投機的makeシミュレータ

```
                :
1: 19:23:26.0   file-change select.c
2: ---:world_create(0)
3: 19:23:35.0   file-change select.c
4: ---:world_delete
5: ---:world_create(0)
6: 19:23:40.5  make
7:        ->cc -c -g -DD select.c
8:        -<cc -c -g -DD select.c,time=1.3
9:        ->cc article.o client.o disp.o
            select.o group.o -o group
10:       -<cc article.o client.o disp.o
            select.o group.o -o group,time=0.8
11:---:world_merge Run-time= -3.4
                :
```

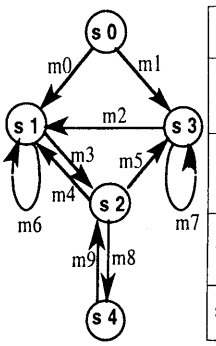
出力（投機的makeの動きと利用者の動き）

図9 投機的makeシミュレータの実行例

4.3 投機的make シミュレータのアルゴリズム

投機的makeシミュレータのアルゴリズムを，状態遷移図を用いて表現したものを図10に示す．

これは，状態s 0を初期状態として終了状態がない状態遷移図である．例えば，ログからファイルの



s0: 初期状態

s1: ファイルの更新のログを読み込んだ状態

s2: "make"入力のログを読み込んだ状態

s3: ログからファイル更新を待っている状態

s4: 処理の終了を待っている状態

m0:ファイルの更新／ファイルの更新時刻とファイル名の記録，world_create()の表示
m1:利用者からの"make"入力／ϕ
m2:ファイルの更新／ファイルの更新時刻とファイル名を記録，world_create()の表示
m3:利用者からの"make"入力／"make"の入力時刻の 記録
m4:ファイルの更新／"make"の入力時刻の 表示，world_merge()とRun-timeの表示，world_create()の表示，
m5:利用者からの"make"入力／"make"の入力時刻の表示，world_merge()とRun-timeの表示，"make"の入力時刻の 記録
m6:ファイルの更新／world_delete()とworld_create()の表示，ファイルの更新時刻とファイル名を記録
m7:利用者からの"make"入力／ϕ
m8:処理の開始／処理の開始時刻の記録
m9:処理の終了／処理の実行時間と終了時間の記録

図10 投機的makeシミュレータの状態遷移図

書き換えの行を読み込むと，状態s0から状態s1に遷移する．利用者が"make"と入力した行を読み込むと，状態s1から状態s2に遷移する．

4.4 投機的makeのシミュレーションの結果

投機的makeシミュレータを使用して，投機的makeの性能を予測した．約45kbytesのソース・ファイルをコンパイルし，リンクする作業を行った時の結果を，図11に棒グラフで示す．図11は，95回のコンパイルの作業のログを入力とした時の結果である．横軸は，利用者が"make"と入力した時刻を0としたときの処理終了時刻である．縦軸は，1秒毎に時間を切ったときの回数である．この例では，ファイルの書き込みから，利用者の"make"入力まで平均5秒の時間がかかることがわかった．この時の処理終了時刻の中央値は-1.1秒となっている．
図11より，約95%の処理は，利用者が"make"と入力する前に終了していることがわかる．

別の利用者が約6 kbytesのソース・ファイルをコンパイルし，リンクする作業を行ったときのデータを調べた．その結果，約80%の処理が，利用者が"make"と入力する前に処理が終了していることがわかった．この時の処理終了時刻の中央値は，-2.6秒となっている．以上より，投機的makeの効果があることがわかる．

5 おわりに

本論文では，世界の概念を提供する投機的処理支援OSについて述べた．投機的処理支援OS上で動作するアプリケーションとして投機的makeを取り上げた．そして世界のDAGを利用した投機的makeの実現について述べた．投機的make実現の特徴は，ファイルのDAGと相似の世界のDAGが作られる点にある．また，本論文では，投機的make実現に先立ち，投機的makeのシミュレータを実現した．投機的makeのシミュレータは，既存のmakeの動きと利用者の動きのトレースを使用する．

今後は，投機的makeを実現し，その性能を評価していきたい．

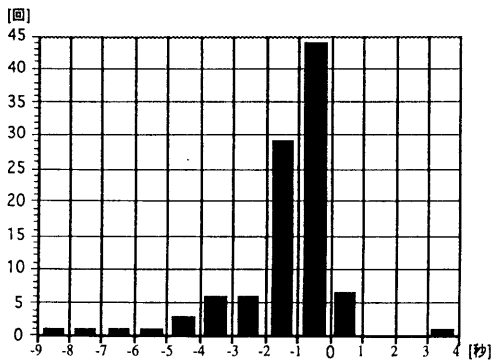


図11 利用者が"make"と入力した時刻を0としたときの処理終了時刻(約45kbytesのソースファイルに対するコンパイル・リンクの作業の場合)

参考文献

[1]根路銘，當眞，新城，喜屋武，翁長："粗粒度投機的並列処理を支援するオペレーティング・システムの構想"，SWoPP94，情報処理学会研究報告，94-ARC-107，pp.89-96(1994).

[2]溝淵，新城，當眞，根路銘，喜屋武，翁長："投機的処理を支援するオペレーティング・システムにおける世界とプロセスの操作"，95-OS-69，pp.103-108(1995).

[3]根路銘，當眞，新城，喜屋武，翁長："投機的処理支援OS上で動作する投機的makeの世界 操作"，1995年電子情報通信学会総合大会，D-168，P.176(1995).

[4]當眞，新城，根路銘，溝淵，喜屋武，翁長："粗粒度投機的処理を支援するオペレーティング・システムにおけるファイル・システム"，SWoPP95，情報処理学会研究報告，95-OS-70-3(1995).

[5]R.B.Osborne: "Speculative Computation in Multilisp"， Proc. US/Japan Workshop on Parallel Lisp ("Parallel Lisp: Languages and Systems", Springer-Verlag LNCS 441， pp.103-135， 1990).

[6]C.J.Fleckenstein and D.Hemmedinger: "Using a Global Name Space for Parallel Execution of UNIX Tools"， Communications of the ACM， Vol.32 No.9， pp.1085-1090 (1989).

[7]R.H.Halstead， Jr.: "New Ideas in Parallel Lisp: Language Design， Implementation， and Programming tools"， Proc. US/Japan Workshop on Parallel Lisp ("Parallel Lisp: Languages and Systems"， Springer-Verlag LNCS 441， pp.2-57 (1990)).

[8]S.Mullender， G.Rossum， A.Tanenbaum， R.Renesse and H.Staveren: "Amoeba: A Distributed Operating System for the 1990s"， IEEE Computer， Vol.23， No.5， pp.44-53 (1990).