ＯＪＬ報告書

トレースログ可視化ツール (TLV) に対する アプリログ機能追加とプロファイリング

350802254　**柳澤 大祐**

名古屋大学 大学院情報科学研究科

情報システム学専攻

2010 **年** 1 **月**

目次

第 1 章

はじめに

近年，PC，サーバ，組込みシステム等，用途を問わずマルチプロセッサの利用が進んでいる．その背景として，シングルプロセッサの高クロック化による性能向上の限界や，消費電力・発熱の増大があげられる．マルチプロセッサシステムでは処理の並列性を高めることにより性能向上を実現するため，消費電力の増加を抑えられる．組込みシステムにおいては，機械制御と GUI など要件の異なるサブシステム毎にプロセッサを使用する例があるなど，従来から複数のプロセッサを用いるマルチプロセッサシステムが存在していたが，部品点数の増加によるコスト増を招くため避けらていた．しかし，近年は，1 つのプロセッサ上に複数の実行コアを搭載したマルチコアプロセッサの登場により低コストで利用することが可能になり，低消費電力要件の強い，組込みシステムでの利用が増加している．

マルチプロセッサ環境では，処理の並列性からプログラムの挙動が非決定的になり，タイミングによってプログラムの挙動が異なる．そのため，ブレークポイントやステップ実行を用いたシングルプロセッサ環境で用いられているデバッグ手法を用いることができない．

そのため，マルチプロセッサ環境では，プログラム実行履歴であるトレースログの解析によるデバッグ手法が主に用いられる．トレースログを解析することで，各プロセスが，いつ，どのプロセッサで，どのような動作したかというプログラムのデバッグに必要な情報が全て記録される．しかし，膨大な量となるトレースログから特定の情報を探し出すのが困難である．さらに，各プロセッサのログが時系列に分散して記録されるため，逐次的にトレースログを解析することも困難である．そのため，開発者が直接トレースログを解析するのは効率が悪い．

トレースログの解析を支援するために，多くのトレースログ可視化ツールが開発されている．組込みシステム向けデバッガソフトウェアや統合開発環境の一部，Unix 系 OS のトレースログプロファイラなどが存在する．しかし，これら既存のツールが扱うトレースログは，OS やデバッグハードウェアごとに異なるため，可視化対象が限定されおり，汎用性に乏しい．さらに，可視化表示項目が提供されているものに限られ，追加や変更が容易ではなく，拡張性に乏しい．

そこで後藤ら [1, 2] によって汎用性と拡張性を備えたトレースログ可視化ツール **TraceLogVisualizer (TLV)** が開発された．TLV 内部でトレースログを抽象的に扱えるよう，トレースログを一般化した標準形式トレースログを定め，任意の形式のトレースログを標準形式トレースログに変換する仕組みを変換ルールとして形式化した．標準形式トレースログから図形データを生成する仕組みを抽象化し，可視化ルールとして形式化した．TLV では，変換ルールと可視化ルールを外部ファイルとして与えることで，汎用性と拡張性を実現している．

本 OJL では，TLV のリリースを複数回行ない，要求の収集を行なった．収集した要求のうち “CPU ごと

のタスク表示” に対する要求が強かったため，これらの実現を行なった．また，ユーザアプリケーションのログ出力を可視化するための機能追加を行った．特定のフォーマットに従ったログ出力を受理し，文字列やユーザ定義状態として可視化することで，ユーザアプリケーションの様々な出力を可視化できるようになった．特に，文字列の可視化は汎用性が高く，システムのメッセージや変数値の変化など様々な出力が可能である．

また，TLV の高速化に対する要求も多かったので，第一段階としてプロファイリングによりボトルネックを調査した．今回は標準形式トレースログへの変換部分を対象とし，どのクラスや処理に時間がかかるかを明らかにした．

最後に，本報告書の構成を述べる．2 章では，TLV の設計について述べる．3 章では，TLV の開発プロセスや発表実績などについて述べる．4 章では，ユーザアプリケーションからのログ出力を可視化する機能追加について述べる．5 章では，ボトルネックを調査したプロファイリングについて述べる．最後に 6 章で本論文のまとめと今後の展望と課題について述べる．

第 2 章

トレースログ可視化ツール (TLV)

2.1 目的

TLV は，汎用性と拡張性を実現することを目標としている．

汎用性とは，可視化表示したいトレースログの形式を制限しないことであり，可視化表示の仕組みをトレースログの形式に依存させないことによって実現する．具体的には，まず，トレースログを抽象的に扱えるように，トレースログを一般化した標準形式トレースログを定義する．そして，任意の形式のトレースログを標準形式トレースログに変換する仕組みを，変換ルールとして形式化する．変換ルールの記述で任意のトレースログが標準形式トレースログに変換することができるため，あらゆるトレースログの可視化に対応することが可能となる．

拡張性とは，トレースログに対応する可視化表現をユーザレベルで拡張できることを表し，トレースログから可視化表示を行う仕組みを抽象化し，それを可視化ルールとして形式化して定義することで実現する．可視化ルールを記述することにより，トレースログ内の任意の情報を自由な表現方法で可視化することが可能になる．

2.2 TraceLogVisualizer の設計

TLV の主機能は，2 つの主たるプロセスと 6 種の外部ファイルによって実現される．図 2.1 に TLV の全体像を示す．

2 つの主たるプロセスとは，標準形式への変換と，図形データの生成である．標準形式への変換は，任意の形式をもつトレースログを標準形式トレースログに変換する処理である．この処理には，外部ファイルとして変換元のトレースログファイル，リソースを定義したリソースファイル，リソースタイプを定義したリソースヘッダファイル，標準形式トレースログへの変換ルールを定義した変換ルールファイルが読み込まれる．

また，図形データの生成は，変換した標準形式トレースログに対して可視化ルールを適用し図形データを生成する処理である．この処理には外部ファイルとして可視化ルールファイルが読み込まれる．可視化ルールファイルとは図形と可視化ルールの定義を記述したファイルである．

TLV は，トレースログとリソースファイルを読み込み，トレースログの対象に対応したリソースヘッダファイル，変換ルールファイルの定義に従い，標準形式トレースログを生成する．生成された標準形式トレースログに可視化ルールファイルで定義される可視化ルールを適用し図形データを生成した後，画面に表示する．

生成された標準形式トレースログと図形データは，TLV データとしてまとめられ可視化表示の元データと

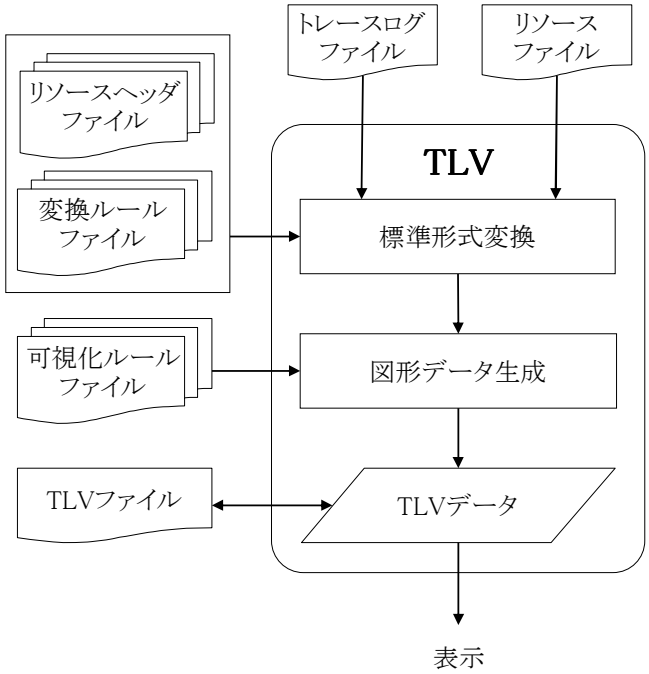

図 2.1　TLV の全体像

して用いられる．TLV データは TLV ファイルとして外部ファイルに保存することが可能であり，TLV ファイルを読み込むことで，標準形式変換と図形データ生成の処理を行わなくても可視化表示できるようになる．

図 2.2 に，TLV のスクリーンショットを示す．

2.2.1　標準形式トレースログへの変換

標準形式トレースログへの変換は，トレースログファイルを先頭から行単位で読み込み，変換ルールファイルで定義される置換ルールに従い標準形式トレースログに置換していくことで行われる．

リソース属性の遷移に伴うイベントや，元のトレースログに欠落している情報を補うイベントなど，元のトレースログの情報だけでは判断できないイベントを出力するには，特定時刻における特定リソースの有無やその数，特定リソースの属性の値などの条件で出力を制御できる必要がある．そのため，TLV の変換ルールでは，置換する条件の指定と，条件指定の際に用いる情報を置換マクロを用いて取得できる仕組みを提供している．

標準形式トレースログに含まれるリソースは，リソースファイルで定義されていなければならない．リソースファイルには，各リソースについて，その名前とリソースタイプ，必要であれば各属性の初期値を定義する．また，その際に使用されるリソースタイプはリソースヘッダファイルで定義されていなければならない．リソースヘッダファイルには各リソースタイプについて，その名前と属性，振る舞いの定義を記述する．

リソースヘッダ，変換ルール，可視化ルールは可視化するターゲット毎に用意する．その際のターゲットはリソースファイルに記述する．

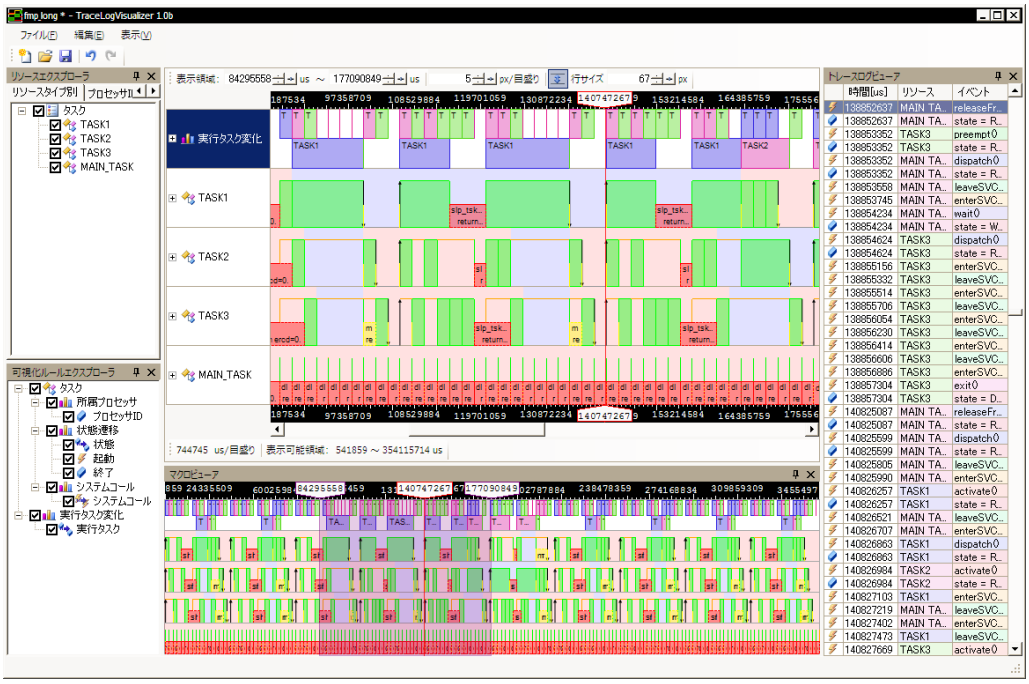

図 2.2　TLV のスクリーンショット

2.2.2　図形データの生成

標準形式変換プロセスを経て得られた標準形式トレースログは，可視化ルールを適用され図形データを生成する．ここで，図形データとは，ワールド変換が行われた全図形のデータを指す．可視化ルールは可視化ルールファイルとして与えられ，適用する可視化ルールはリソースファイルに記述する．

図形データの生成方法は，標準形式トレースログを一行ずつ可視化ルールのイベント期間と一致するか判断し，一致した場合にその可視化ルールの表示期間をワールド変換先の領域として採用しワールド変換することで行われる．

2.3　標準形式トレースログ

2.3.1　トレースログの抽象化

標準形式トレースログを提案するにあたり，トレースログの抽象化を行った．

トレースログを，時系列にイベントを記録したものである．イベントとはイベント発生源の属性の変化，イベント発生源の振る舞いと考えた．ここで，イベント発生源をリソースと呼称し，固有の識別子をもつものとする．つまり，リソースとは，イベントの発生源であり，名前を持ち，固有の属性をもつものと考えることができる．

リソースは型により属性，振る舞いを特徴付けられる．ここでリソースの型をリソースタイプと呼称する．

属性は，リソースが固有にもつ文字列，数値，真偽値で表されるスカラーデータとし，振る舞いはリソース

の行為であるとする．

リソースタイプとリソースの関係は，オブジェクト指向におけるクラスとオブジェクトの関係に類似しており，属性と振る舞いはメンバ変数とメソッドに類似している．ただし，振る舞いはリソースのなんらかの行為を表現しており，メソッドの，メンバ変数を操作するための関数や手続きを表す概念とは異なる．

主に，振る舞いは，属性の変化を伴わないイベントを表現するために用いる．振る舞いは任意の数のスカラーデータを引数として受け取ることができ，これは，図形描画の際の条件，あるいは描画材料として用いられることを想定している．

トレースログの抽象化を以下にまとめる．

トレースログ
: 時系列にイベントを記録したもの．

イベント
: リソースの属性の値の変化，リソースの振る舞い．

リソース
: イベントの発生源．固有の名前，属性をもつ．

リソースタイプ
: リソースの型．リソースの属性，振る舞いを特徴付ける．

属性
: リソースが固有にもつ情報．文字列，数値，真偽値のいずれかで表現されるスカラーデータで表される．

振る舞い
: リソースの行為．主に属性の値の変化を伴わない行為をイベントとして記録するために用いることを想定している．振る舞いは任意の数のスカラーデータを引数として受け取ることができる．

2.3.2 標準形式トレースログの定義

本小節では，前小節で抽象化したトレースログを，標準形式トレースログとして形式化する．標準形式トレースログの定義は，EBNF(Extended Backus Naur Form) および終端記号として正規表現を用いて行う．正規表現はスラッシュ記号 (/) で挟むものとする．

前小節によれば，トレースログは，時系列にイベントを記録したものであるので，1つのログには時刻とイベントが含まれるべきである．トレースログが記録されたファイルのデータを TraceLog，TraceLog を改行記号で区切った1行を TraceLogLine とすると，これらは次の EBNF で表現される．

```
TraceLog = { TraceLogLine,"\n" };
TraceLogLine = "[",Time,"]",Event;
```

TraceLogLine は”[”,”]”で時刻を囲み，その後ろにイベントを記述するものとする．

時刻は Time として定義され，次に示すように数値とアルファベットで構成するものとする．

```
Time = /[0-9a-Z]+/;
```

アルファベットが含まれるのは，10 進数以外の時刻を表現できるようにするためである．これは，時刻の単位として「秒」以外のもの，たとえば「実行命令数」などを表現できるように考慮したためである．この定

義から，時刻には，2 進数から 36 進数までを指定できることがわかる．

前小節にて，イベントを，リソースの属性の値の変化，リソースの振る舞いと抽象化した．そのため，イベントを次のように定義した．

```
Event = Resource,".",(AttributeChange|BehaviorHappen);
```

Resource はリソースを表し，AttributeChange は属性の値の変化イベント，BehaviorHappen は振る舞いイベントを表す．リソースはリソース名による直接指定，あるいはリソースタイプ名と属性条件による条件指定の 2 通りの指定方法を用意した．

リソースの定義を次に示す．

```
Resource = ResourceName
         | ResourceTypeName,"(",AttributeCondition,")";
ResourceName = Name;
ResourceTypeName = Name;
Name = /[0-9a-Z_]+/;
```

リソースとリソースタイプの名前は数値とアルファベット，アンダーバーで構成されるとした．AttributeCondition は属性条件指定記述である．これは次のように定義する．

```
AttributeCondition = BooleanExpression;
BooleanExpression = Boolean
   |ComparisonExpression
   |BooleanExpression,[{LogicalOpe,BooleanExpression}]
   |"(",BooleanExpression,")";
ComparisonExpression = AttributeName,ComparisonOpe,Value;
Boolean = "true"|"false";
LogicalOpe = "&&"|"||";
ComparisonOpe = "=="|"!="|"<"|">"|"<="|">=";
```

属性条件指定は，論理式で表され，命題として属性の値の条件式を，等価演算子や比較演算子を用いて記述できるとした．

AttributeName はリソースの名前であり，リソース名やリソースタイプ名と同様に，次のように定義する．

```
AttributeName = Name;
```

イベントの定義にて，AttributeChange は属性の値の変化を，BehaviorHappen は振る舞いを表現しているとした．これらは，リソースとドット”.”でつなげることでそのリソース固有のものであることを示す．リソースの属性の値の変化と振る舞いは次のように定義した．

```
AttributeChange = AttributeName,"=",Value;
Value = /[^"\\]+/;
BehaviorHappen =  BehaviorName,"(",Arguments,")";
BehaviorName = Name;
```

表 2.1　標準形式トレースログの例

```
1 [2403010]MAIN_TASK.leaveSVC(ena_tex,ercd=0)
2 [4496099]MAIN_TASK.state=READY
3 [4496802]TASK(state==RUNNING).state=READY
```

```
Arguments = [{Argument,[","]}];
Argument = /[^"\\]*/;
```

属性の変化イベントは，属性名と変化後の値を代入演算子でつなぐことで記述し，振る舞いイベントは，振る舞い名に続けてカンマで区切った引数を括弧”()”で囲み記述するとした．

2.3.3　標準形式トレースログの例

前小節の定義を元に記述した，標準形式トレースログの例を表 2.1 に示す．

1 行目がリソースの振る舞いイベントであり，2 行目，3 行目が属性の値の変化イベントである．1 行目の振る舞いイベントには引数が指定されている．

1 行目，2 行目はリソースを名前で直接指定しているが，3 行目はリソースタイプと属性の条件によってリソースを特定している．

2.4　図形データ

2.4.1　座標系

図形を定義する座標系と，表示する座標系は分離して考えるとする．これにより，図形を表示環境から独立して定義することが可能になる．図形を定義する座標系をローカル座標系，表示する座標系をデバイス座標系と呼称する．

また，TLV では，高さと時間を次元に持つ，ワールド座標系という座標系を導入した．ローカル座標系で定義された図形は，はじめに，ワールド座標系における，図形を表示すべき時間の領域にマッピングされ，これを表示環境に依存するデバイス座標系にマッピングすることで表示する．これにより，図形の表示領域を，抽象度の高い時刻で指定することが可能になる．ここで，ローカル座標系からワールド座標系へのマッピングをワールド変換と呼称する．また，表示する時間の領域を表示期間と呼称する．表示期間は開始時刻と終了時刻で表される時刻のペアである．

ローカル座標系において，図形の大きさと位置を定義する際は，pixel 単位による絶対指定か，ワールド座標系へのマッピング領域に対する割合を % で指定する相対指定かのいずれかを用いる．

図 2.3 に座標系の例を，図 2.4 にワールド変換の例を示す．

2.4.2　基本図形と図形，図形群

可視化表現は，複数の図形を組み合わせることで実現する．この際，基本となる図形の単位を基本図形と呼称する．

基本図形として扱える形状は楕円，多角形，四角形，線分，矢印，扇形，文字列の 7 種類とする．基本図形

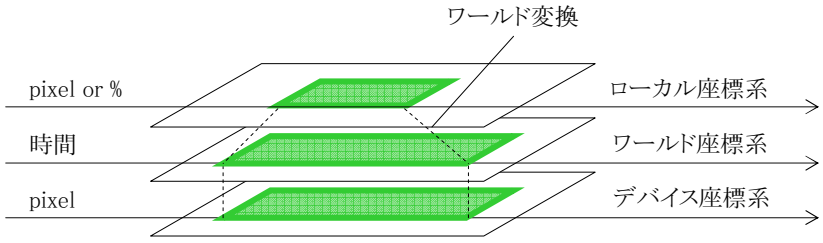

図 2.3　座標系

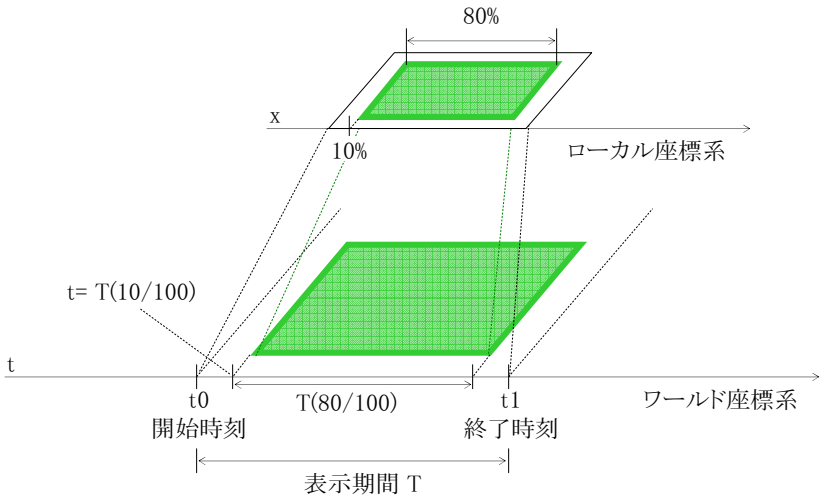

図 2.4　ワールド変換

は，形状や大きさ，位置，塗りつぶしの色，線の色，線種，透明度などの属性を指定して定義する．

複数の基本図形を仮想的に z 軸方向に階層的に重ねたものを，単に図形と呼称し，可視化表現の最小単位とする．図形は，構成する基本図形を順序付きで指定し，名前をつけて定義する．図形は名前を用いて参照することができ，その際に引数を与えることができるとする．この際，引数は，図形を構成する基本図形の，任意の属性に割り当てることを想定している．

複数の図形を仮想的に z 軸方向に階層的に重ねたものを図形群と呼称する．図 2.5 に図形と図形群の例を示す．

2.5　図形データとイベントの対応

本小節では，前小節で述べた可視化表現とトレースログのイベントをどのように対応付けるのかを述べる．

2.5.1　開始イベント，終了イベント，イベント期間

前小節において，可視化表現は，図形をワールド変換を経て表示期間にマッピングすることであることを説明した．ここで，表示期間の開始時刻，終了時刻を，イベントを用いて指定するとする．つまり，指定されたイベントが発生する時刻をトレースログより抽出することにより表示期間を決定する．このようにして，ト

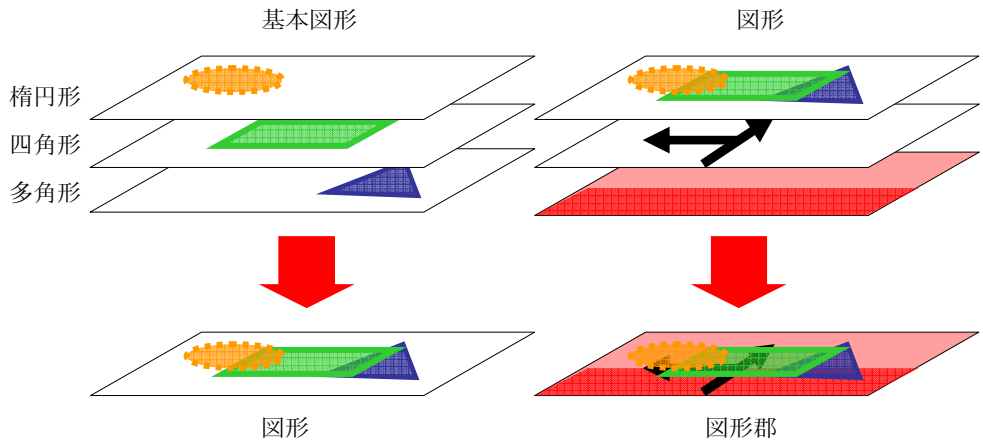

図 2.5　図形と図形群

表 2.2　イベント期間を抽出するトレースログ

```
1 [1000]MAIN_TASK.state=RUNNING
2 [1100]MAIN_TASK.state=WAITING
```

レースログのイベントと可視化表現を対応付ける．ここで，開始時刻に対応するイベントを開始イベント，終了時刻に対応するイベントを終了イベントと呼称し，表示期間をイベントで表現したものをイベント期間と呼称する．

2.5.2　可視化ルール

図形群と，そのマッピング対象であるイベント期間を構成要素としてもつ構造体を，可視化ルールと呼称する．図 2.6 に，標準形式トレースログを用いてイベント期間を定義した可視化ルールの例を示す．図 2.6 において，runningShape を，位置がローカル座標の原点，大きさがワールド座標系のマッピング領域に対して横幅 100%，縦幅 80% の長方形で色が緑色の図形とする．この図形を，開始イベント `MAIN_TASK.state=RUNNING`，終了イベント `MAIN_TASK.state` となるイベント期間で表示するよう定義したものが可視化ルール taskBecomeRunning である．開始イベント `MAIN_TASK.state=RUNNING` は，リソース `MAIN_TASK` の属性 `state` の値が `RUNNING` になったことを表し，終了イベント `MAIN_TASK.state` は，リソース `MAIN_TASK` の属性 `state` の値が単に変わったことを表している．

taskBecomeRunning を，表 2.2 に示すトレースログからイベントを抽出して表示期間の時刻を決定し，図形のワールド変換を行った結果が図 2.6 の右下に示すものである．

2.6　TraceLogVisualizer のその他の機能

本節では，標準形式変換と可視化データ生成の他に TLV が備える機能について詳述する．

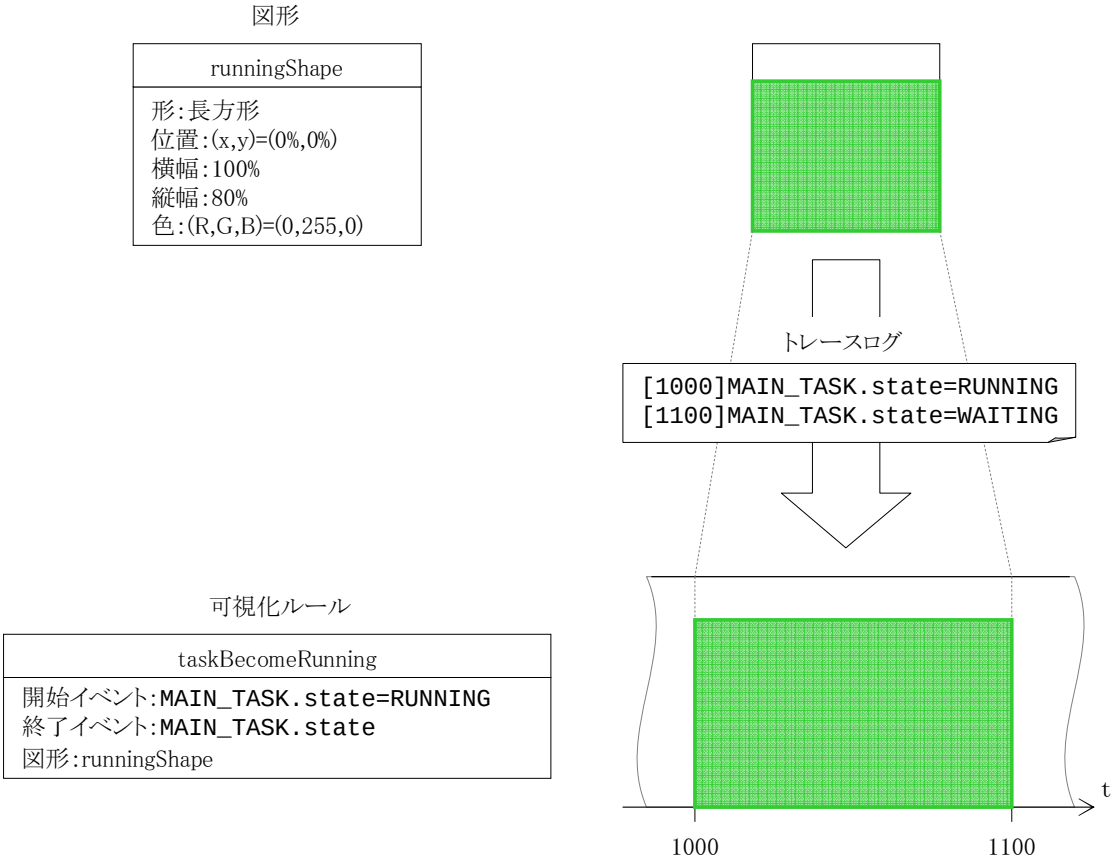

図 2.6　可視化ルール

2.6.1　マーカー

TLV では，可視化表示部にマーカーと呼ぶ印を指定の時刻に配置することができる注目するべきイベントの発生時刻にマーカーを配置することで，可視化表示内容の理解を補助することができる．

マーカーとマーカーの間には，その間の時間が表示されるので，ソフトウェアの計測を行うことができる．マーカーには名前を付けることができ，色の指定が可能である．また，マーカーはマウス操作で選択することができ，拡大縮小などの各種操作に利用される．マーカーは階層構造で管理し，階層ごとに表示の切り替え，マーカー間時間の表示などを行うことができる．

2.6.2　可視化表示部の制御

可視化表示ツールでは，可視化表示部を制御する操作性が使い勝手に大きく影響するため，TLV では目的や好みに合わせて様々な操作で制御を行えるようにした．TLV では，可視化表示部の制御として，表示領域の拡大縮小，移動を行うことができる．これらの操作方法として，キーボードによる操作，マウスによる操作，値の入力による操作の 3 つの方法を提供した．

マウスによる操作は，クリックによる操作，ホイールによる操作，選択による操作がある．クリックによる

操作はカーソルを虫眼鏡カーソルに変更してから行う．左クリックでカーソル位置を中心に拡大，右クリックで縮小を行う．また，左ダブルクリックを行うことでクリック箇所が中心になるように移動する．ホイールによる操作は，コントロールキーを押しながらホイールすることで移動を行い，シフトキーを押しながら上へホイールすることでカーソル位置を中心に拡大，下へスクロールすることで縮小する．選択による操作では，マウス操作により領域を選択し，その領域が表示領域になるように拡大する．

キーボードによる操作は，可視化表示部において方向キーを押すことで行い，微調整するのに適している．左キーで表示領域を左に移動し，右キーで右に移動する．上キーでカーソル位置，または選択されたマーカーを中心に表示領域を縮小し，下キーで拡大する．

値の入力による操作では，より詳細な制御を行うことができる．可視化表示部の上部にはツールバーが用意されており，そこで表示領域の開始時刻と終了時刻を直接入力することができる．

2.6.3 マクロ表示

TLV の要求分析を行った際，可視化表示部で拡大した場合に全体の内どの領域を表示しているのかを知りたいという要求があった．そのため，TLV では，マクロビューアというウィンドウを実装した．

マクロビューアでは，トレースログに含まれるイベントの最小時刻から最大時刻までを常に可視化表示しているウィンドウで，可視化表示部で表示している時間を半透明の色で塗りつぶして表示する．塗りつぶし領域のサイズをマウスで変更することができ，それに対応して可視化表示部の表示領域を変更することができる．

マクロビューアでは可視化表示部と同じように，キーボード，マウスにより拡大縮小，移動の制御を行うことができる．

2.6.4 トレースログのテキスト表示

TLV では，標準形式トレースログをテキストで表示するウィンドウを実装した．ここでは，トレースログの内容を確認することができる．

可視化表示部とテキスト表示ウィンドウは連携しており，テキスト表示ウィンドウでマウスを移動すると，カーソル位置にあるトレースログの時刻にあわせて可視化表示部のカーソルが表示されたり，ダブルクリックすることで対応する時刻に可視化表示部を移動することができる．また，可視化表示部でダブルクリックすることで，ダブルクリック位置にある図形に対応したトレースログが，テキスト表示ウィンドウの先頭に表示されるようになっている．

2.6.5 可視化表示項目の表示非表示切り替え

TLV では，可視化表示する項目を可視化ルールにより変更，追加することができるが，それらの表示を可視化ルールやリソースを単位で切り替えることができる．これらの操作は，リソースウィンドウと可視化ルールウィンドウで行う．リソースウィンドウではリソースファイルで定義されたリソースを，リソースタイプやグループ化可能な属性毎にツリービュー形式で表示しており，チェックの有無でリソース毎に表示の切り替えを行える．同じように，可視化ルールウィンドウでは，可視化ルールごとに表示の切り替えを行える．

2.7　本 OJL で追加した機能

本 OJL で追加したその他の機能について述べる．

2.7.1　CPU ごとのタスク表示機能

タスクがコアからコアへと移動することをマイグレーションと呼ぶ．OS 開発者からマイグレーションの可視化機能が要望として挙げられたため，本機能の実現を決定した．図 2.7 は本機能のスクリーンショットである．

2.7.2　バージョン情報表示機能

2 人で別々に開発を行っていた際，その TLV がどのバージョンであるかが分かりづらいことがあった．そのため，「ヘルプ」メニュー，「TLV について」からバージョン情報表示ウィンドウを呼び出せる機能を追加した．

2.7.3　エラーメッセージ詳細表示機能

元々表示されるエラーメッセージは比較的情報が少なく，利用者がどのような対処をすればよいかが分かりづらかった．そこで，エラーが発生した箇所に応じて，原因と思われるメッセージを追加し，ユーザに対処法をわかりやすく伝えるよう改良した．

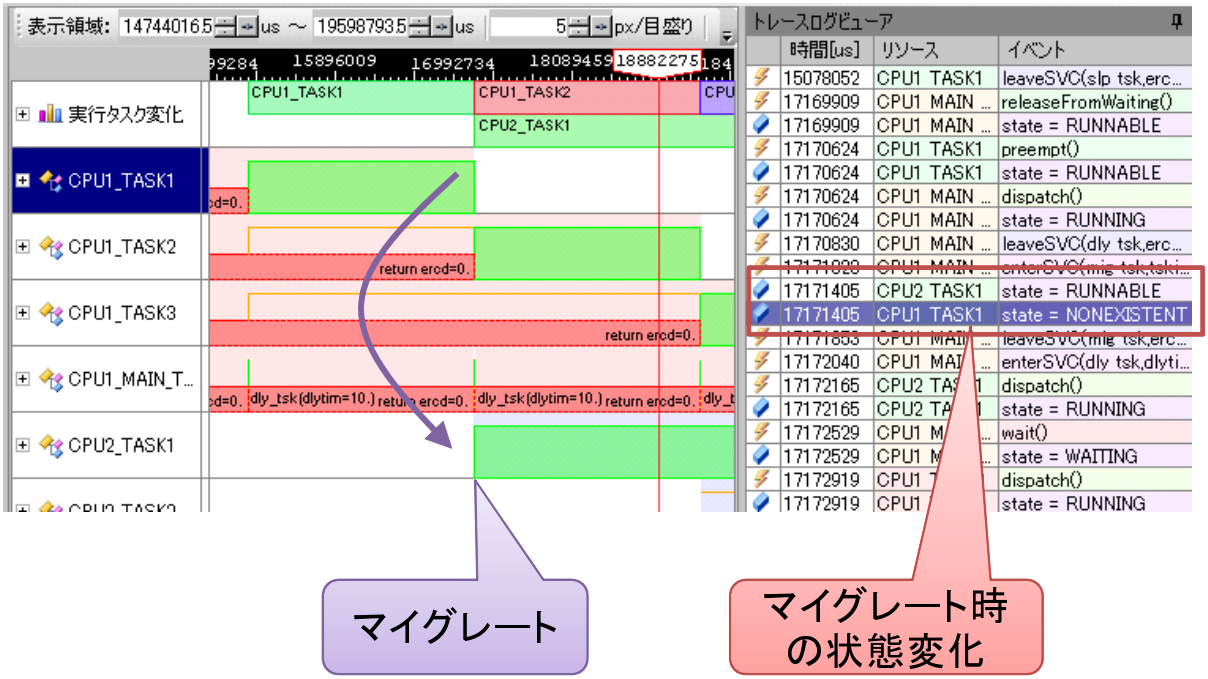

図 2.7　CPU ごとのタスク表示

第 3 章

実績

3.1 開発プロセス

TLV は，OJL(On the Job Learning) の開発テーマとして開発された．OJL とは，企業で行われているソフトウェア開発プロジェクトを教材とする実践教育であり，製品レベルの実システムの開発を通じて創造的な思考力を身につけるとともに，単なる例題にとどまらない現実の開発作業を担うことにより，納期，予算といった実社会の制約を踏まえたソフトウェア開発の実際について学ぶことを目的としている．

TLV はプロジェクトベースで開発が行われ，本年度は企業出身者 1 名と教員 1 名がプロジェクトマネージャを務め，筆者含む学生 4 名が開発実務を行った．進捗の報告は，週に 1 度のミーティングと週報の提出により行った．

3.1.1 フェーズ分割

TLV の開発は複数フェーズに分割して行われている．フェーズ 1,2,3 は主に後藤ら [1] によって実施された．本報告書ではフェーズ 3 以降について述べる．

各フェーズの内容は次に述べる通りである．

フェーズ 1,2,3

2007 年 5 月〜2008 年 3 月までに，一期生によって実施され，TLV の主要機能の実装が行われた [1].

フェーズ 3

■期間　2008 年度後期

■実施内容

- 新規に参加したため，前提となる C#,TOPPERS の学習を行なった．
- TLV 各クラスに対する単体テスト，システムテストを実行した．
- CPU ごとのタスク表示機能を実装した．
- リリース作業を担当した．

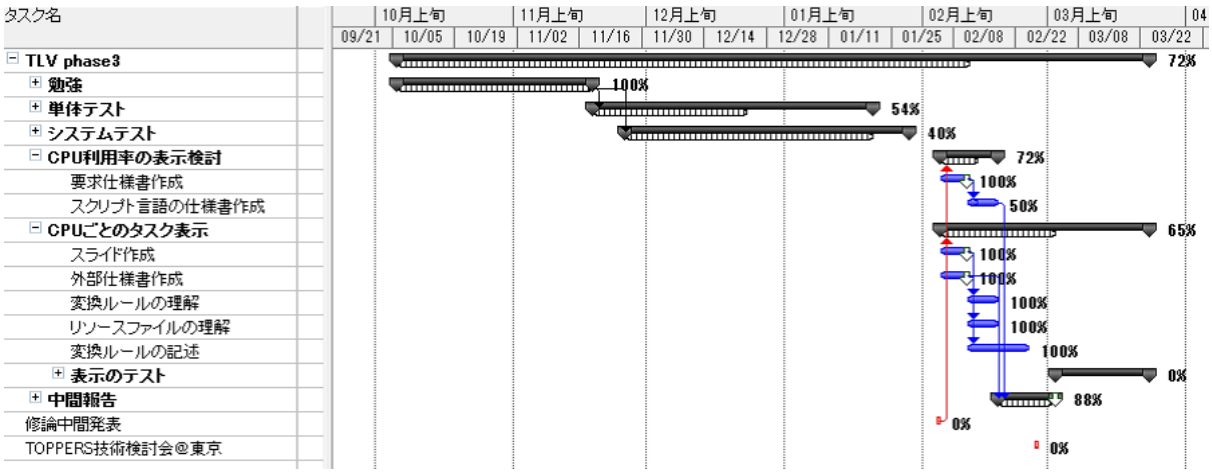

図 3.1　フェーズ 3 のスケジュール

■スケジュール　図　3.1 のように，10 月始めから 11 月中旬にかけて前提となる知識の学習を行ない，11 月中旬から 1 月末にかけて TLV に対するテストを行ない，1 月末から 2 月中旬にかけて CPU ごとのタスク表示機能を実装した．

■成果　単体テストは 328 個のメソッドのうち 162 個のメソッドに対して単体テストを作成した．システムテストでは不具合を発見できなかった．機能追加と並行してテストを実施したため，仕様が安定せず効率が悪かった．

また，3 月 25 日に TOPPERS 会員に向けに 1.0rc をリリースし，要求の収集を行なった．1.0rc のリリースによって 20 件のユーザインタフェースの改良案，追加の機能要求，可視化表現項目などの要求が得られた．

フェーズ 4

■期間　2009 年度前期

■実施内容

- 第 4 章で述べるアプリケーションログ機能追加を行なった．
- 前フェーズで収集した要求のうち，TLV の使いやすさを向上させる要求を実装した．
- TLV の仕様変更に伴い，前フェーズで実装した CPU ごとのタスク表示機能が正常に動作しなくなっていたため，再実装した．
- エラーメッセージをより詳細化した．
- リリース作業を担当し，リリースに伴う作業を自動化した

■スケジュール　図　3.2 のように，4 月中旬から 5 月上旬にかけて，CPU ごとのタスク表示機能再実装を行った．また，その後は 7 月上旬までアプリケーションログ機能を実装し，その後は 7 月下旬までエラーメッセージの改良を行った．

■成果　第 4 章で述べる機能を追加した．27 件の要望チケットを完了した．

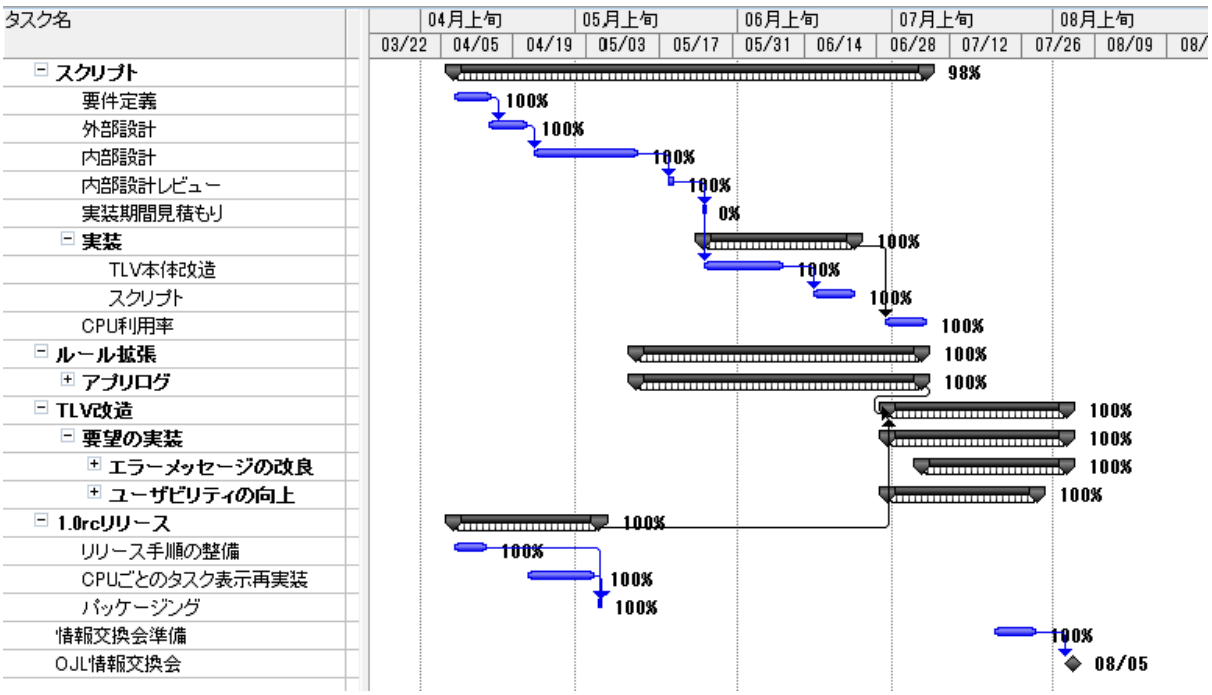

図 3.2　フェーズ 4 のスケジュール

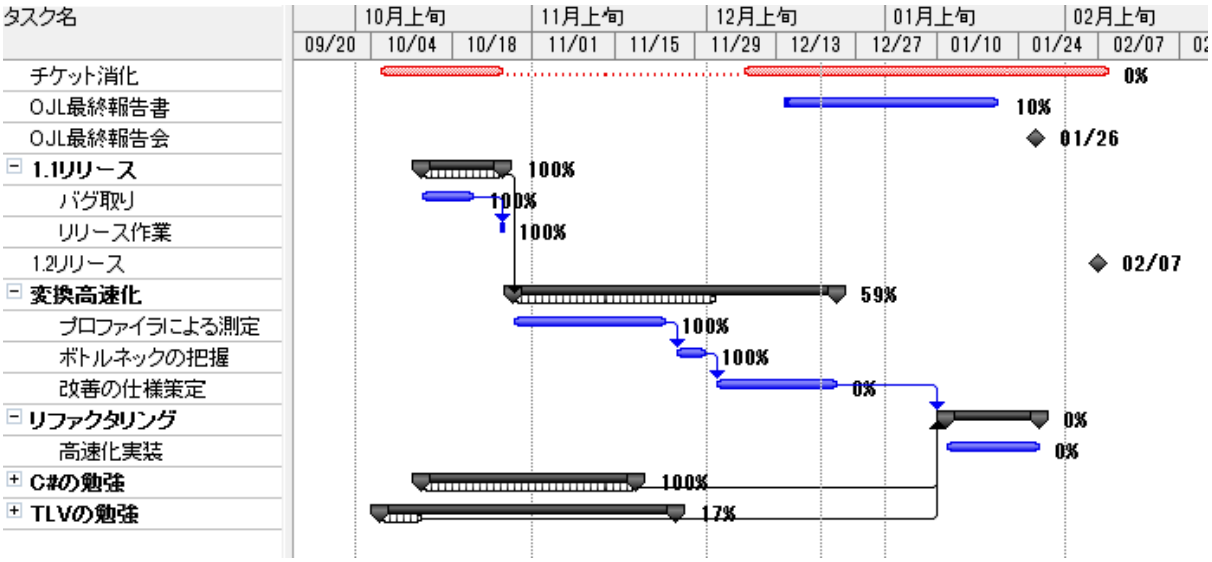

図 3.3　フェーズ 5 のスケジュール

5 月 13 日に，TOPPERS 会員に対して，フェーズ 3 までの内容をまとめた TLV 1.0 をリリースした．

フェーズ 5

■期間　2009 年度後期

■実施内容

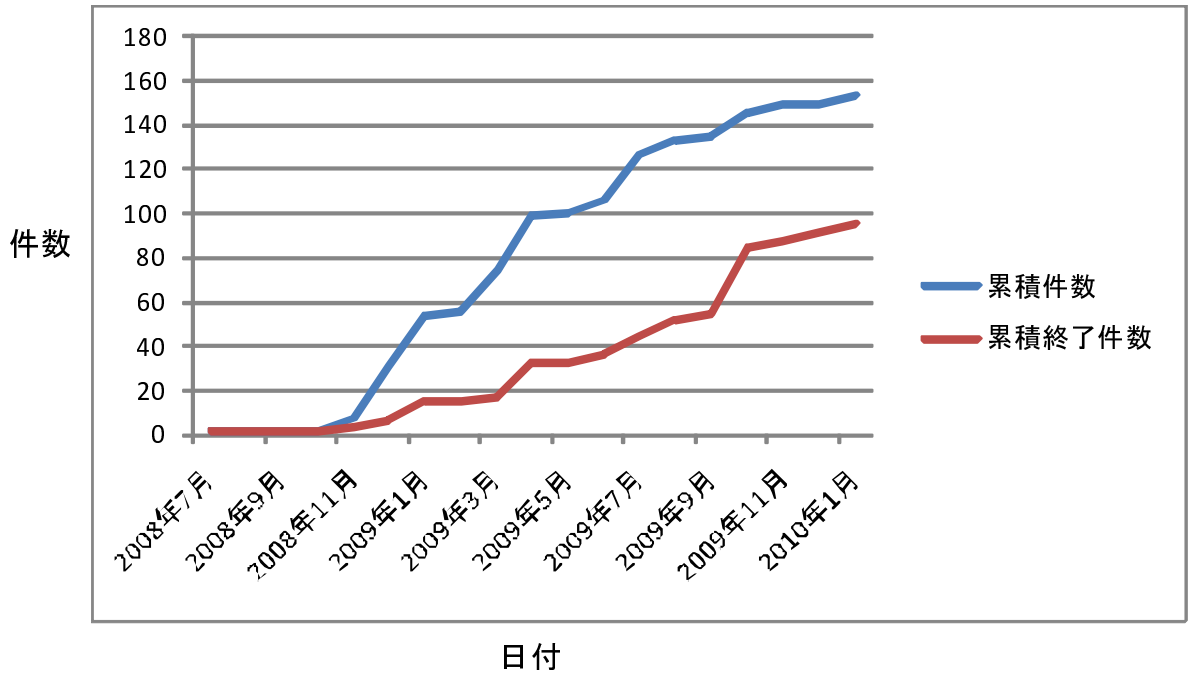

図 3.4　チケット消化率

- バージョン 1.1 をリリースした.
- 第 5 章で述べるようにプロファイリングを行った.
- 新たに三期生が参加したため，課題を与えるなどの指導を行なった.

■スケジュール　図 3.3 のように，10 月上旬に TLV1.1 リリース作業を行った．10 月下旬から 12 月中旬にかけて，プロファイリングによる測定，改善案の作成を行った.

■成果物　第 5 章で述べるプロファイリング結果をまとめた．また，TOPPERS 会員に対して RC 版をリリースし，その後一般向けに TLV1.1 をリリースした.

TLV 1.1rc1　10 月 1 日に TOPPERS 会員に対して公開.
TLV 1.1rc2　rc1 の不具合を修正し，10 月 26 日に TOPPERS 会員に対して公開.
TLV 1.1　11 月 9 日に一般向けに公開.

3.2　チケット消化率

累積チケット数の推移をグラフにまとめると図 3.4 のようになる．横軸が期間，縦軸が件数であり，青線が累積件数を，赤線が累積終了件数を表している．この図より，TLV プロジェクトでは一般的な要求曲線を描いており，プロジェクトは発展段階にあるととらえることができる.

3.3 発表実績

平成 21 年度 情報処理学会 第 139 回 システム LSI 設計技術 (SLDM) 研究会において発表 [2] を行ない，情報処理学会山下記念研究賞/学生奨励賞を受賞した．

Embedded Technology 2009(ET2009) [*1] において，ブース出展及び TLV の一般公開についてプレス発表を行なった．

3.4 活用事例

名古屋大学大学院情報科学研究科付属組込みシステム研究センター (NCES) [*2]内の 7 件のプロジェクトのうち，2 件のプロジェクトによって利用されている．また，同 NCES のコンソーシアム型共同研究によっても利用されている．

[*1] http://www.jasa.or.jp/et/

[*2] http://www.nces.is.nagoya-u.ac.jp/

第 4 章

アプリログ機能拡張

4.1　概要

■目的

これまでに寄せられた TLV への要望として、いわゆる printf デバッグを行いたいというものがあった。これが可能となると、一般的な printf デバッグによる出力を TLV のタイムライン上に表示させることによって、メッセージが出力されたときのタスク状態などが一目でわかるという利点がある。この要望を実現するものがアプリログ拡張である。また、関数の呼び出し関係や列挙型変数の値などを、タスク状態のように可視化したいという要望があったため、これに対してもアプリログ拡張で対応する。

本拡張は TLV ソースコードには触れず、リソースファイルやルールなどの記述のみで実現することとした。ソースコードに手を加えると、サブウィンドウやポップアップなどで情報を表示できるという利点があったが、拡張に時間がかかるという点と、TLV は変換ルール・可視化ルールによる汎用性・拡張性を特徴としているので、開発者自身がその効果を確認するという点が理由である。

■概要

本拡張では、次の 4 種類の可視化方法を提供する。

- 文字列の可視化
  タスク文字列可視化　タスクに関連する出力を、タスクの可視化行に重ねて表示
  文字列可視化　特定のタスクに関連しない出力を、独立の行に可視化表示
- ユーザ定義状態の可視化
  タスクユーザ定義状態　タスクに関連するユーザ定義状態を、タスクの可視化行に重ねて表示
  ユーザ定義状態可視化　特定のタスクに関連しない出力を、独立の行に可視化表示

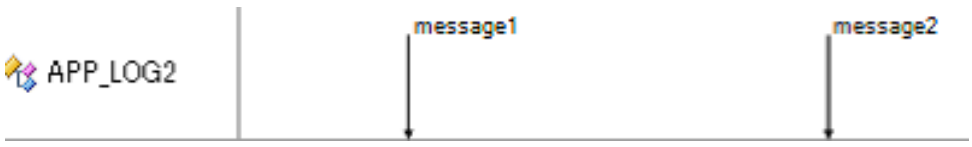

図 4.1　文字列の可視化

図 4.2　ユーザ定義状態の可視化

文字列の可視化とは、図 4.1 のように、指定時刻に指定文字列を表示するものである。これにより、ユーザはデバッグメッセージ、変数値を可視化できる。

また、ユーザ定義状態の可視化とは、図 4.2 のように、タスクやハンドラなどの状態とは別個の状態を表示するものである。ユーザは、それぞれのユーザ定義状態が何を意味するかをユーザ側で定義し、これを可視化できる。

■利用法

本機能をユーザアプリケーションから利用する場合は、以下の手順で行う。

1. リソースファイルに追記
   タスクに関連しない可視化のときのみ行う。新しく描画する行を追加するので、それに対応した記述をこちらで用意したテンプレートからリソースファイルに追記し、初期値や表示名などを記述する。
2. アプリケーションにログ出力関数の呼び出しを追加
   TOPPERS カーネルの場合、ユーザアプリケーションからのログ出力は syslog API により行う。この関数を用いて、本機能が受理するフォーマットでログを出力する。
3. アプリケーションを実行しログを取得
   ログの取得方法は、実機やエミュレータごとに異なるので、ここでは触れない。
4. TLV により可視化
   生成したログおよび追記したリソースファイルを TLV に入力し、可視化する。

4.2　設計

4.2.1　文字列可視化

リソースファイル

■タスクに関連しない出力

タスクに関連しない出力を行う際は、ユーザはテンプレートに沿ってリソースファイルに記述を追加しなければならない。文字列可視化は独立した行に描画する、すなわち一つのリソースを要求するためである。図 4.3 がそのテンプレートである。ユーザは、*ROWNAME* をリソースファイルの他の項目名と重複しないよう決定し、*RID* を他の同種の記述と重複しないよう決定し、リソースファイルに追加する必要がある。また、*RID* で設定した値は、後にログ出力の呼び出しをユーザアプリケーションに追加する際に参照する。

■タスクに関連する出力

タスクに関連する出力の際は、前述のようにリソースファイルへの追記は必要ない。これは、全体が参照するリソースヘッダファイルに、図 4.4 の定義がされているためである。この定義により、リソースファイルへ

```
"ROWNAME":{
    "Type":"ApplogString",
    "Attributes":{
        "id":"RID",
        "str":""
    }
}
```

ROWNAME　文字列可視化行の名前 `[^"]+`

RID　文字列可視化行の ID `[^":]+`

図 4.3　文字列可視化用テンプレート

```
"Task":{
    "DisplayName" :"タスク",
    "Attributes" :{
        "id":{
            ...
        },
        ...
        "applog_str":{
           "VariableType"  :"String",
           "DisplayName"   :"文字列",
           "AllocationType":"Dynamic",
           "CanGrouping"   :false,
           "Default"   :""
        },
        ...
    }
}
```

図 4.4　タスク文字列可視化用リソースヘッダ内記述

の追記なしにタスクに関する出力を受け付ける。

トレースログ形式

アプリケーションログ機能が認識できるログを出力する方法は 2 通りある。ひとつが、TOPPERS カーネルの syslog 関数を呼ぶ方法である。もうひとつが、本スクリプト拡張で用意する API を利用する方法である。関数の引数に出力したい内容を指定して呼び出して利用する。

図 4.5 はタスクに関連しない可視化の書式である。*RID* は先にリソースファイルで定義した値を参照する。*STR* には出力したい文字列を設定する。図 4.6 がタスクに関連する可視化の書式である。*TID* は関連するタスクの ID である。リソースファイルを参照し、関連づけたいタスクの *id* 属性の値を指定する。また、ユーザアプリケーションでログを出力すると、**出力ログ形式**に記述してある形式に従う。

利用シナリオ

タスクとは関係ない文字列可視化の具体例を示す。想定として、TOPPERS/ASP カーネル向けのユーザアプリケーションから “hoge” というメッセージを出力することを想定する。

syslog syslog(”applog str : ID %s : %s.”,*RID*,*STR*);
API void applog_str(const char **RID*, const char **STR*)
出力ログ形式 [*TIME*] applog str : ID *RID* : *STR*.
TIME 時刻 `[0-9a-zA-Z]+`
RID 表示行 ID `[^.:"]+`
STR 出力文字列 `[^.]*`

図 4.5 文字列可視化の書式

syslog syslog(”applog strtask : TASK %d : %s.”,*tid*,*str*);
API void applog_strtask(const int *tid*, const char **str*)
出力ログ形式 [*time*] applog strtask : TASK *tid* : *str*.
time 時刻 `[0-9a-zA-Z]+`
tid タスク ID `[0-9]+`
str 出力文字列 `[^.]*`

図 4.6 タスク文字列可視化の書式

```
    "TASK_TEX":{
      ...
    },
    "APP_LOG":{
      "Type":"ApplogString",
      "Attributes":
      {
        "id":"1"
      }
    }
  }
}
```

図 4.7 リソースファイル追加内容

1. リソースファイルへの追記
   /sampleFiles/asp/asp_short.res 末尾に、図 4.7 のように記述を追加する。
2. ユーザアプリケーションへのログ出力関数の呼び出しを追加
   アプリケーションへ、 図 4.8 のような記述を追加する。
3. アプリケーションを実行して、ログを取得する。その結果、`[123456789] applog str : ID 1 : hoge.` のような行を含むログが出力される。
4. TLV へ (1) で追記したリソースファイルと (3) で取得したログを入力し、図 4.1 のような可視化表示を得る。

また、タスクに関する文字列可視化の際は、次の作業を行うだけで、図 4.9 のような可視化表示が得られる。

```
int rid = 1;
char* str = "hoge";
syslog("applog str : ID %d : %s.", rid, str);
```

図 4.8　syslog 関数呼び出し

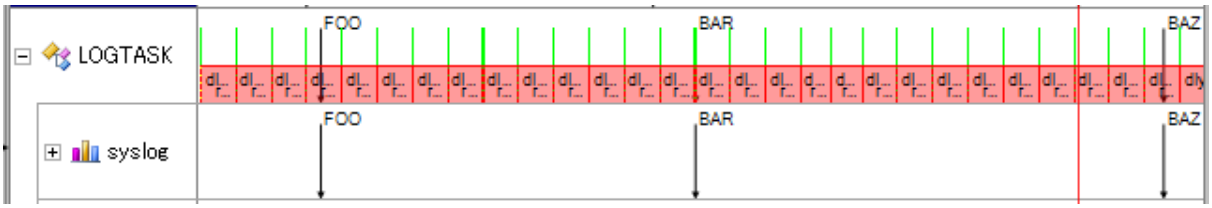

図 4.9　タスク文字列の可視化

1. ユーザアプリケーションにログ出力コードを追加
2. アプリケーションを実行してログ取得
3. TLV へ入力して可視化表示を得る

4.2.2　ユーザ定義状態可視化

TLV は、最大 8 個のユーザ定義状態が可視化できる方法を提供する。ユーザは、各自でどの状態が何を意味するか決定し、アプリケーションにログ出力コードを埋め込む。

利用の手順は文字列可視化と同様である。タスクに関係しない可視化では、リソースファイルの追加が必要な点も同様である。

リソースファイル

■タスクに関連しない出力

図 4.10 はタスクに関連しない可視化を行う際に追加が必要な記述である。*ROWNAME* および *RID* は文字列可視化の時と同じ意味であり、*SID* は、初期状態の ID である。

■タスクに関連する出力

文字列可視化同様、タスクに関する出力の場合にはリソースファイルへの追記は不要である。これは、リソースヘッダ内の図 4.11 の記述による。

トレースログ形式

図 4.12 はタスクに関連しない、図 4.13 はタスクに関連する可視化のためのトレースログの形式である。

```
"ROWNAME":{
    "Type":"ApplogState",
    "Attributes":{
        "id":"RID",
        "state":SID
    }
}
```

ROWNAME　文字列可視化行の名前 `[^"]+`
RID　ユーザ定義状態可視化行の ID `[^"]+`
SID　初期状態 `[0-9]+`

図 4.10　ユーザ定義状態可視化用テンプレート

```
"Task":{
    "DisplayName" :"タスク",
    "Attributes" :{
        "id":{
            ...
        },
        ...
        "applog_state":{
            "VariableType"  :"Number",
            "DisplayName"   :"ユーザ定義状態",
            "AllocationType":"Dynamic",
            "CanGrouping"   :false,
            "Default"       :0
        },
        ...
    }
}
```

図 4.11　タスクユーザ定義状態可視化用リソースヘッダ内記述

4.3　実装

実装方針は、先に述べたようにルール類のみに手を加えるというものであるため、まずはリソースファイル/リソースヘッダファイルへの記述を追加した。リソースファイルは、図 4.3 および図 4.10 として、使用前にユーザがリソースファイルへ追記するためのテンプレートを用意した。また、リソースヘッダファイルは、図 4.4 および図 4.11 として、拡張した。

次に変換ルールを実装した。図 4.14 は TOPPERS/ASP カーネル用に実装した変換ルールである。JSON で記述されており、**"入力されたトレースログに適用する正規表現" : "変換後の標準形式トレースログ"** という書式で書かれている。図中には 4 組のルールが記述されており、このそれぞれが 4 種類のアプリログ可視化方法に対応する。また、`(?<time>\d+)` という記法を用いると、正規表現`\d+` にマッチした文字列を`${time}`で取得することができる。`$RES_NAME{ApplogString(id==${rid})}`は、リソースヘッダファイルにて定義されるリソースの型が `ApplogString` であり、条件 `id==${rid}`を満たすリソースの名前へと置換される。これにより、例えば `[123456] applog str : ID 2 : message1.` というトレースログが、

syslog syslog(”applog state id %s %d.”,*RID*,*SID*);
API void applog_stt(const char **RID*, const int *SID*)
出力ログ形式 [*TIME*] applog state id *RID SID*.
TIME 時刻 `[0-9a-zA-Z]+`
RID 表示行 ID `[^\. ]+`
SID 状態 `[0-9]+`

図 4.12 ユーザ定義状態可視化の書式

syslog syslog(”applog state task %d %d.”,*TID*,*SID*);
API void applog_stttsk(const int *TID*, const int *SID*)
出力ログ形式 [*TIME*] applog state task *TID SID*.
TIME 時刻 `[0-9a-zA-Z]+`
TID タスク ID `[0-9]+`
SID 状態 `[0-9]+`

図 4.13 タスクユーザ定義状態可視化の書式

```
    ...
  ],
  "\[(?<time>\d+)\] applog str : ID (?<rid>[^: ]+) : (?<str>[^\.]+)\.?"  :
    "[${time}]$RES_NAME{ApplogString(id==${rid})}.str=${str}",
  "\[(?<time>\d+)\] applog strtask : TASK (?<tid>[^: ]+) : (?<str>[^\.]+)\.?"  :
    "[${time}]$RES_NAME{Task(id==${tid})}.applog_str=${str}",
  "\[(?<time>\d+)\] applog state : ID (?<rid>[^: ]+) : (?<state>\d+)\.?"  :
    "[${time}]$RES_NAME{ApplogState(id==${rid})}.state=${state}",
  "\[(?<time>\d+)\] applog statetask : TASK (?<tid>[^: ]+) : (?<state>\d+)\.?"  :
    "[${time}]$RES_NAME{Task(id==${tid})}.applog_state=${state}"
}
```

図 4.14 TOPPERS/ASP カーネル向け変換ルール

`[123456]APP_LOG2.str=message1` という標準形式トレースログへと変換される。

次は可視化ルールである。図 4.15 は TOPPERS カーネル向け可視化ルールの図形定義部分である。図形は、`SHAPENAME(arg0, arg1, ...)` という形で参照され、その引数が`${ARG0}`などの部分を置換するという特徴を持つ。この図形定義は文字列可視化用の *applogStateShapes* とユーザ定義状態用の *applogStateShapes* から構成される。*applogStateShapes* は 2 つの図形から構成されており、1 つめの図形では引数で受け取った文字列を画面上に配置し、2 つめの図形では時刻の位置に矢印マークを配置する。*applogStateShapes* は、引数で与えられた色をもつ長方形を配置する。

図 4.16 は TOPPERS カーネル向け可視化ルールの図形配置部分である。*applogString* は文字列可視化に対応し、*ApplogString* リソースの *str* 属性が更新されたとき、先に定義した *applogStringShapes* に引数として *str* を渡しつつ配置する。*applogStringTask* はタスク文字列可視化に対応し、*Task* リソースの *applog_str* 属性が更新されたとき、*applogStringShapes* に引数として *applog_str* を渡しつつ配置する。*applogState* は

```
  ...
],
"applogStringShapes":[
  {
    "Type":"Text",
    "Size":"100%,100%",
    "Font":{
      "Align":"TopLeft",
      "Size":7
    },
    "Text":"${ARG0}"
  },
  {
    "Type":"Arrow",
    "Points":["0,80%","0,0"],
    "Pen":{"Color":"ff000000","Width":1}
  }
],
"applogStateShapes":[
  {
    "Type":"Rectangle",
    "Location":"0,0",
    "Size":"100%,100%",
    "Pen":{"Color":"${ARG0}","Width":1,"Alpha":255},
    "Fill":"${ARG0}",
    "Alpha":100
  }
]
```

図 4.15　TOPPERS カーネル向け可視化ルール（図形定義）

ユーザ定義状態可視化に対応し、*ApplogState* リソースの *state* 属性が更新されたところから次の更新までの範囲に *applogStateShapes* を配置し、その引数を *state* の値（範囲 [0,7]）に応じた色とする。*applogStateTask* はタスクユーザ定義状態可視化に対応し、*Task* リソースの *applog_state* 属性が更新されたところから次の更新までの範囲に *applogStateShapes* を配置し、その引数に応じた色を指定する。

```
  ...
},
"applogString":{
  "DisplayName":"アプリログ（文字列）",
  "Target":"ApplogString",
  "Shapes":{
    "strLog":{
      "DisplayName":"文字列",
      "When":"${TARGET}.str",
      "Figures":"applogStringShapes(${VAL})"
    }
  }
},
"applogStringTask":{
  "DisplayName":"アプリログ（文字列）",
  "Target":"Task",
  "Shapes":{
    "strLogTask":{
      "DisplayName":"タスク文字列",
      "When":"${TARGET}.applog_str",
      "Figures":"applogStringShapes(${VAL})"
    }
  }
},
"applogState":{
  "DisplayName":"アプリログ（状態）",
  "Target":"ApplogState",
  "Shapes":{
    "stateLog":{
      "DisplayName":"ユーザ定義状態",
      "From":"${TARGET}.state",
      "To":"${TARGET}.state",
      "Figures":{
        "${FROM_VAL}==0":
          "applogStateShapes(ff0000)",
        ...
        "${FROM_VAL}==7":
          "applogStateShapes(ff00ff)"
      }
    }
  }
},
"applogStateTask":{
  "DisplayName":"アプリログ（状態）",
  "Target":"Task",
  "Shapes":{
    "stateLogTask":{
      "DisplayName":"ユーザ定義状態",
      "From":"${TARGET}.applog_state",
      "To":"${TARGET}.applog_state",
      "Figures":{
        "${FROM_VAL}==0":
          "applogStateShapes(ff0000)",
        ...
        "${FROM_VAL}==7":
          "applogStateShapes(ff00ff)"
      }
    }
  }
}
```

図 4.16　TOPPERS カーネル向け可視化ルール（図形配置）

第 5 章

プロファイリング

5.1 概要

■目的

TLV には，高速化をのぞむ要望が寄せられることが多い．ログの種類や長さに依存するものの，可視化表示が完了するまでに 100 分かかる事例も報告されている．しかしながら，高速化は難度の高い課題であるため，今まで着手することを見合わせていた．

今回，高速化に集中的に取り組むことになり，その第一段階として，ボトルネックを調査するために，プロファイリングを行うこととなった．

5.1.1 プロファイラの選定

プロファイリングを実施するにあたって，プロファイラを選定する必要があった．プロファイラは数多く存在するが，その中から入手性などを考慮して以下の 3 つを選択肢とした．

- NProf
- CLR プロファイラ
- Visual Studio 2008 Team Suite

■NProf

NProf [*1] は，.NET Framework に対応したオープンソースのプロファイラである．図 5.1 がスクリーンショットである．オープンソースソフトウェアなので入手しやすいという利点がある．しかし，次の欠点から採用を見送った．

- 機能が少ない．例えば，計算時間を割合でしか表示できない．
- 利用者が多くなく，情報収集しづらい．
- 開発が停止している．

[*1] http://code.google.com/p/nprof/

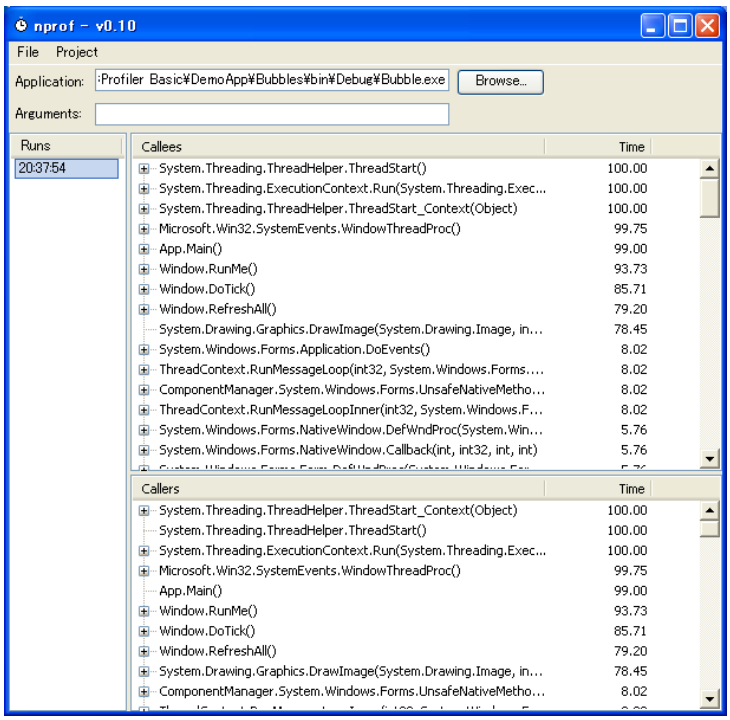

図 5.1　NProf のスクリーンショット

■CLR プロファイラ

CLR プロファイラ[*2] は，Microsoft が提供する.NET Framework 用のプロファイラである．ヒープメモリの状態やガベージコレクションを計測することができる．図 5.2 がスクリーンショットである．

無償で利用できるので入手しやすいが，メモリの状態表示が主で，関数の呼び出し回数などの時間に関する機能はあまり力が入っておらず，時間計測を目的とする今回の利用には向かないと判断した．

■Visual Studio 2008 Team Suite

Visual Studio 2008 Team Suite [*3] は, Microsoft の Visual Studio 2008 のバリエーションであり, Professional の機能に加えて大規模開発向けのさまざまな機能が搭載されている．

TLV の開発環境が Microsoft Visual Studio 2008 であり，また欠点である入手性も MSDN AA により解決できると判明したため，プロファイラとして Visual Studio 2008 Team Suite（以下 Team Suite）を用いることとした．

5.1.2　プロファイリングの種別

Team Suite のプロファイラには，サンプリングおよびインストルメントの 2 つの手法が存在する．

サンプリングは，プログラムを一時停止し，その時コールスタックの一番上にあるメソッドを記録する．これを一定時間ごとに繰り返し，プログラムが終了したらこれらを集計する．これにより，もっとも多くスタッ

[*2] http://msdn.microsoft.com/ja-jp/library/ms979205.aspx
[*3] http://msdn.microsoft.com/ja-jp/teamsystem/bb933734.aspx

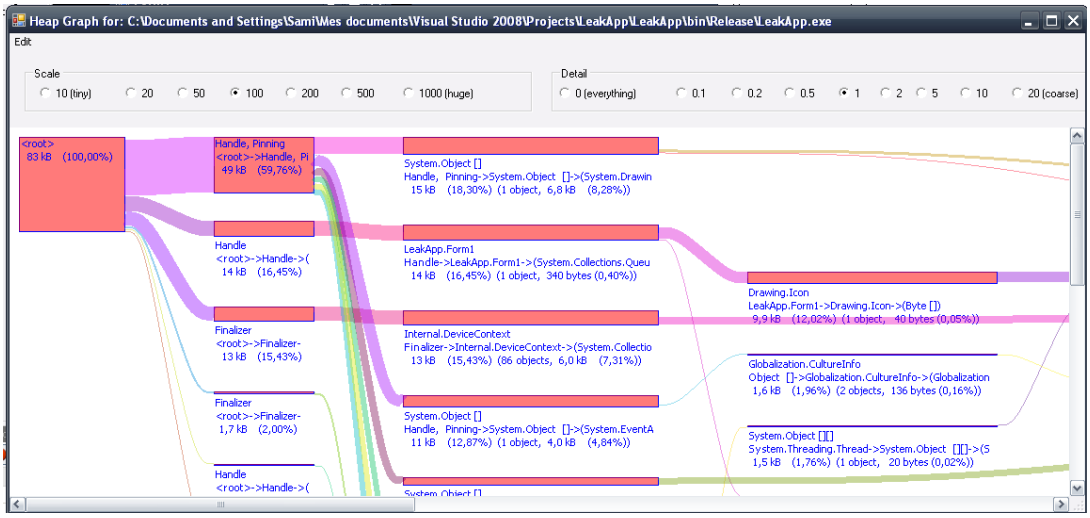

図 5.2　CLR プロファイラのスクリーンショット

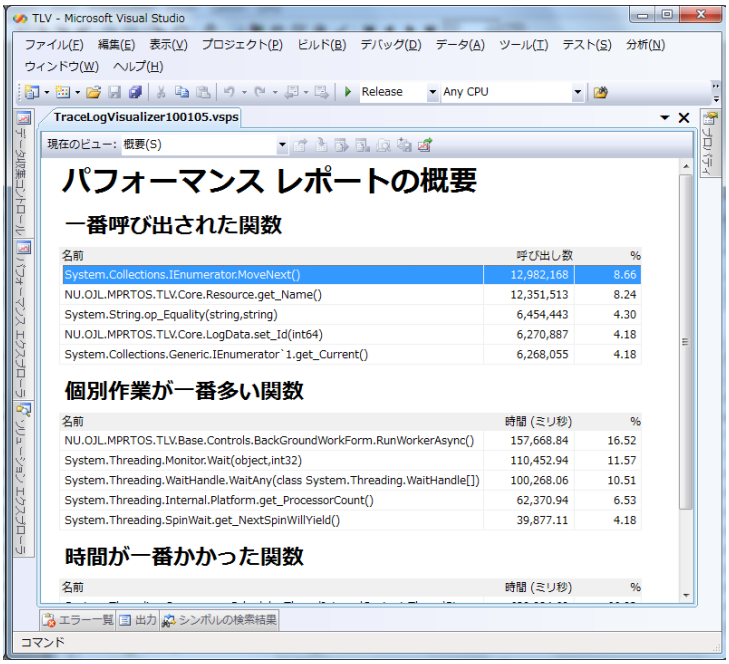

図 5.3　Visual Studio 2008 Team Suite のスクリーンショット

クトップに現れたメソッドがもっとも多くの時間を消費していることが分かる．プログラムの実行速度低下はインストルメントに比べて小さいという利点があるが，処理時間の少ないメソッドが大量に呼ばれ，そこがボトルネックとなっている場合はそれを検出できない場合がある．

インストルメントは，非常に小さいメソッドを除いた全てのメソッドの開始と終了にタイマを挿入し，プログラム終了後にこれらを集計する．これによりほぼ全メソッドの実行時間を測定することができるので，細かい違いが求められる際やサンプリングでは詳細が判明しない際に用いる．ただし，メソッドの開始および終了

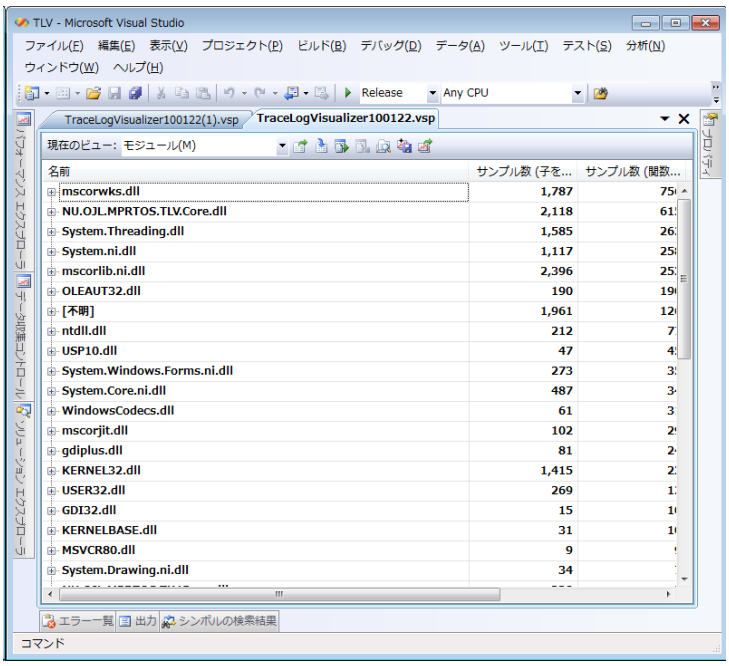

図 5.4　サンプリングでプロファイリングをしたスクリーンショット

に挿入されるコードによるオーバーヘッドがかなり大きく，通常のプログラムに比べて 10 倍以上時間がかかる点，また収集されるデータが膨大なため，集計にかかる時間もかなり大きい点が欠点である．

そこで，まずはサンプリングによりプロファイリングを試みた．その結果，mscoworks.dll，OLEAUT32.dll および CLRStubOrUnknownAddress といったシステムコールとおぼしき項目が並び（図 5.4），TLV のどの処理が時間を要しているのかが判断しづらかった．以上より，インストルメントによりプロファイリングを行うこととした．

5.1.3　測定方法

今回の測定では，図 2.1 にある標準形式変換を対象としている．もう一方の図形データ生成は対象としなかった．そのため，以下の手順でプロファイリングを行った．

1. Team Suite のウィザードにて初期設定を行った．この際，方式をインストルメント，対象を実行ファイルとした．実行ファイルは，Release ビルドされたものを指定した．
2. Team Suite にて，「プロファイルを一時停止して起動」ボタンから TLV を起動．
3. TLV にて，新規作成ウィザードを起動し，asp_short.res および asp_short.log を指定し，即座に変換処理が可能となるように準備した．
4. Team Suite にて，「収集の再開」ボタンからプロファイラを実行させた．
5. TLV にて，新規作成ウィザードの OK ボタンを押し，ログの処理を開始した．
6. 標準形式への変換処理が完了し，可視化処理へと移行した段階で Team Suite の「収集の一時停止」ボタンを押してプロファイラを停止させた．
7. 可視化処理が完了した段階で TLV を終了させた．

8. Team Suite の集計処理が実行され，プロファイル結果が出力された．

一時停止した状態で起動した理由は，GUI アプリケーションの場合，起動時からプロファイリングの情報収集をしていると実際に計測したい処理以外の部分が多く入ってしまうので，その影響を最小限にするためである．GUI アプリケーションの場合，起動時からプロファイリングの情報収集をしていると実際に計測したい処理以外の部分が多く入ってしまうためである．

5.2 プロファイリング結果

Team Suite の「プロセス」ビューからアプリケーションの実行時間を調べたところ，175 秒であった．一方，TLV の実行ファイルが消費したアプリケーション時間を「モジュール」ビューから調べたところ，49.7 秒だった．アプリケーション時間とは，プロファイリングしたコードの直接実行にかかった時間である．システムコールの呼び出しにかかった時間や，他のスレッドの実行の待機時間は除外される．ここから，差し引き 125.3 秒はこれらのオーバーヘッドである．

次に，「関数」ビューにて，「アプリケーション時間 %（関数のみ）」の列を降順にソートし，時間時間の大きいものを調査した．「関数のみ」とは，その関数のみの実行時間であり，外部関数の呼び出しにかかった時間を含まないことを意味する．この結果次に列挙するクラスに含まれるメソッドの実行時間が大半を占め，残りはごく少数であることが分かった．

- Core.TraceLogData
- System.Linq パッケージ
- System.Collections パッケージ
- Core.Time
- Core.LogData

これらのクラス群が占める実行時間をグラフにすると図 5.5 となる．グラフより，これら 5 群のみで全体の 56% を占めていることが分かる．特に，もっとも実行時間の長い Core.TraceLogData クラスは 18% を占める．次節にて，クラス群ごとに詳細とそれにたいする改善案を述べる．

5.3 改善案

5.3.1 Core.TraceLogData

TraceLogData クラスを詳細にみると，表 5.1 のようになる．上位の 2 メソッドで 13.71% を占める．これらメソッドは LINQ（後述）の検索条件を指定するラムダ式である．ラムダ式とは，特定の記法を用いることで匿名メソッドを簡単に記述できる記法である．これらラムダ式はそれぞれ短いのでこれ以上の短縮は不可能だが，ここでは一つのログを基準に，それより時刻が早いログを検索するために用いられている．改善案としては，LINQ は線形探索を行う点に注目し，ログの格納に O(logn) のデータ構造を利用するというアプローチが考えられる．

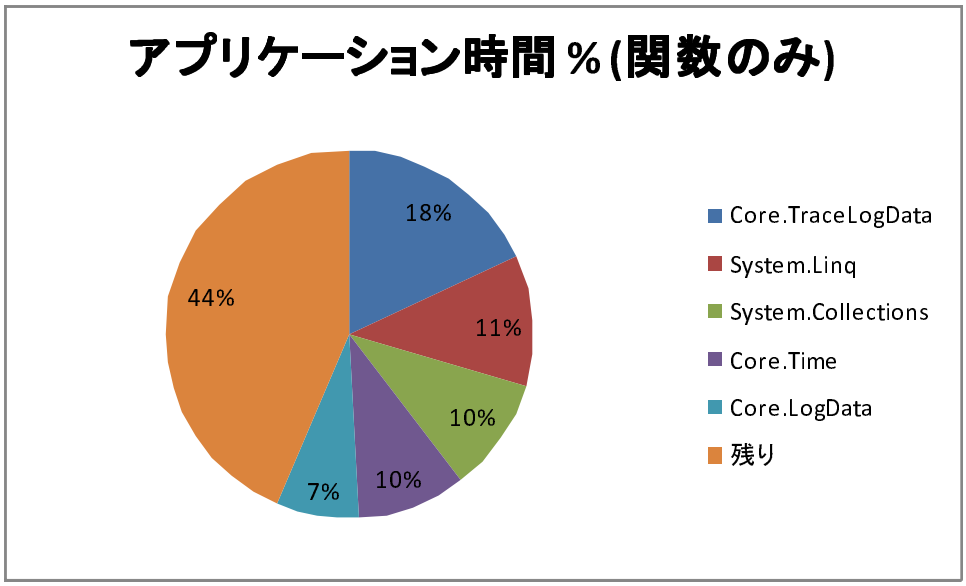

図 5.5　アプリケーション時間に占める割合

5.3.2　System.Linq および System.Collections

LINQ [*4] とは，System.Collections に含まれる各種コレクションに対して，SQL 文のような検索クエリを発行できる仕組みである．表 5.2 をみると，最上位のメソッドが 6.5% の実行時間を占めていることが分かる．

また，LINQ をはじめ各所で用いられている System.Collections パッケージの結果は表 5.3 に記述されている．これをみると，MoveNext() メソッドが 5.88% を占めている．

これらより，LINQ および Collections が使われている場所はいくつかあるが，これも先ほどと同様に効率のよいデータ構造を採用し，検索にかかる時間を低減することが有効だと考える．

5.3.3　Core.Time

Time クラスの場合は，Time2 つ同士の大小を比較するメソッドが 9.47% を占めている．図 5.4 は Time クラスの結果である．これらメソッドは大変小さなものなので，これ以上の高速化は望めない．しかし，呼び出し回数が 600 万回以上あるため，全体の一部を占めている．

これの対策は，Time クラスを利用しているクラスを突き止め，そのクラスで行われる Time の呼び出しの回数を減らすことが挙げられる．

[*4] http://msdn.microsoft.com/ja-jp/library/bb308959.aspx

表 5.1　TraceLogData クラスのプロファイル詳細結果

関数名	アプリケーション時間（子を含む）	アプリケーション時間（関数のみ）	アプリケーション時間 %（子を含む）	アプリケーション時間 %（関数のみ）	呼び出し数
<>c__DisplayClass16.<GetObject>b__a(class LogData)	11,084.51	3,872.50	22.31	7.79	3,544,294
<>c__DisplayClass4.<GetAttributeValue>b__1(class LogData)	8,393.55	2,941.68	16.89	5.92	2,682,256
<GetObject>d__18.MoveNext()	4,695.36	1,238.18	9.45	2.49	17,968
GetAttributeValue(class TraceLog)	3,590.89	863.47	7.23	1.74	1,748
<>c__DisplayClasse.<GetObject>b__8(class Resource)	38.26	19.86	0.08	0.04	65,987
GetAttributeValue(string)	3,686.68	2.57	7.42	0.01	1,748
Add(class TraceLog)	142.75	13.63	0.29	0.03	3,087
GetObject(string)	179.14	2.53	0.36	0.01	3,311
<GetObject>d__18..ctor(int32)	6.99	3.86	0.01	0.01	9,894
GetObject(class TraceLog)	8.39	2.68	0.02	0.01	8,146
.ctor(class ResourceData)	0.36	0	0	0	1
.ctor(class TraceLogList,class ResourceData)	0.36	0	0	0	1
<>c__DisplayClass16.<GetObject>b__9(class LogData)	30.72	15.89	0.06	0.03	24,430
<>c__DisplayClass4.<GetAttributeValue>b__0(class LogData)	21.47	11.14	0.04	0.02	17,075
<GetObject>d__18.<>m__Finally23()	2.44	1.31	0	0	6,703
<GetObject>d__18.IEnumerable<Resource>.GetEnumerator()	6.57	3.16	0.01	0.01	9,888
<GetObject>d__18.System.IDisposable.Dispose()	2.27	1.6	0	0	9,887
get_LogDataBase()	0.88	0.88	0	0	4,887
get_MaxTime()	0.95	0.95	0	0	6,700
get_MinTime()	0	0	0	0	1
set_Path(string)	0	0	0	0	1
総計		8,995.89		18.11	

表 5.2　System.Linq パッケージのプロファイル詳細結果

関数名	アプリケーション時間（子を含む）	アプリケーション時間（関数のみ）	アプリケーション時間 %（子を含む）
System.Linq.Parallel.WhereQueryOperator`1.WhereQueryOperatorEnumerator`1.MoveNext()	28,907.02	3,229.67	58.18
System.Linq.Parallel.PartitionedDataSource`1.IndexRangeEnumerator.MoveNext()	963.2	963.2	1.94
System.Linq.Parallel.AsynchronousChannelMergeEnumerator`1.MoveNext()	916.16	145.82	1.84
System.Linq.Parallel.SpoolingTask`1.SpoolingWork()	29,237.21	102.65	58.85
System.Linq.Parallel.AsynchronousChannelMergeEnumerator`1.IncrementChannelIndex()	127.82	84.95	0.26
System.Linq.Parallel.IAsynchronousChannel`1.TryDequeue(!0&)	223.67	81.91	0.45
System.Linq.Parallel.IChannel`1.TryEnqueue(!0,int32)	97.65	57.7	0.2
System.Linq.Parallel.AsynchronousOneToOneChannel`1.TryDequeue(!0&)	141.77	52.37	0.29
System.Linq.Parallel.QueryOperator`1.QueryOpeningEnumerator.MoveNext()	1,852.38	45.26	3.73
System.Linq.Parallel.QueryOperator`1.QueryOpeningEnumerator.get_Current()	114.08	44.7	0.23
System.Linq.Parallel.AsynchronousChannelMergeEnumerator`1.IncrementChannelIndex()	42.87	42.87	0.09
…			
総計		5661.96	

5.3.4　Core.LogData

図　5.5 は LogData クラスの結果である．LogData クラスは大変小さいクラスで，getter/setter しか存在しない．しかし，呼び出し回数が 600 万回以上あるため，全体の一部を占めている．

これの対策は，Time 同様に LogData の呼び出し箇所を調べ，呼び出し回数を減らせるようなら減らすことが挙げられる．

表 5.3　System.Collections パッケージのプロファイル詳細結果

関数名	アプリケーション時間（子を含む）	アプリケーション時間（関数のみ）	アプリケーション時間 %（子を含む）	アプリケーション時間 %（関数のみ）	呼び出し数
System.Collections.IEnumerator.MoveNext()	30,815.88	2,923.63	62.03	5.88	12,982,168
System.Collections.Generic.IEnumerator`1.get_Current()	1,164.70	1,164.70	2.34	2.34	6,268,055
System.Collections.Generic.IEnumerator`1.get_Current()	683.9	683.9	1.38	1.38	6,229,393
System.Collections.Generic.IEnumerator`1.get_Current()	42.51	42.51	0.09	0.09	163,528
System.Collections.Generic.IEnumerator`1.get_Current()	69.38	46.74	0.14	0.09	163,526
System.Collections.Generic.Dictionary`2.ContainsKey(!0)	19.52	19.52	0.04	0.04	27,988
System.Collections.Generic.Dictionary`2.ContainsKey(!0)	16.18	16.18	0.03	0.03	45,928
System.Collections.Generic.Dictionary`2.ContainsKey(!0)	12.07	12.07	0.02	0.02	41,552
System.Collections.Generic.Dictionary`2.get_Item(!0)	10.66	10.66	0.02	0.02	43,292
System.Collections.Generic.Dictionary`2.get_Item(!0)	7.75	7.75	0.02	0.02	26,196
System.Collections.Generic.Dictionary`2.get_Item(!0)	9.37	9.37	0.02	0.02	41,520
System.Collections.Generic.List`1.Add(!0)	10.25	10.25	0.02	0.02	41,505
System.Collections.Generic.List`1.get_Count()	9.55	9.55	0.02	0.02	87,201
…					
		5,038.20		10.09	

表 5.4　Time クラスのプロファイル詳細結果

関数名	アプリケーション時間（子を含む）	アプリケーション時間（関数のみ）	アプリケーション時間 %（子を含む）	アプリケーション時間 %（関数のみ）	呼び出し数
NU.OJL.MPRTOS.TLV.Core.Time.CompareTo(valuetype NU.O	5,030.06	2,783.40	10.12	5.6	6,232,724
NU.OJL.MPRTOS.TLV.Core.Time.op_LessThanOrEqual(valuety	6,947.04	1,920.68	13.98	3.87	6,226,550
NU.OJL.MPRTOS.TLV.Core.Time.MinTime(int32)	254.67	32.12	0.51	0.06	41,506
NU.OJL.MPRTOS.TLV.Core.Time..ctor(string,int32)	142.21	21.01	0.29	0.04	44,611
NU.OJL.MPRTOS.TLV.Core.Time..cctor()	0	0	0	0	1
NU.OJL.MPRTOS.TLV.Core.Time.MaxTime(int32)	0.2	0	0	0	1
NU.OJL.MPRTOS.TLV.Core.Time.op_Equality(valuetype NU.OJ	0	0	0	0	1
NU.OJL.MPRTOS.TLV.Core.Time.op_GreaterThan(valuetype N	3.08	0.76	0.01	0	3,087
NU.OJL.MPRTOS.TLV.Core.Time.op_LessThan(valuetype NU.C	2.02	0.64	0	0	3,087
NU.OJL.MPRTOS.TLV.Core.Time.op_Subtraction(valuetype NU	1.28	0.01	0	0	11
NU.OJL.MPRTOS.TLV.Core.Time.ToString()	0.18	0.01	0	0	14
総計		4758.63		9.57	

表 5.5　LogData クラスのプロファイル詳細結果

関数名	アプリケーション時間（子を含む）	アプリケーション時間（関数のみ）	アプリケーション時間 %（子を含む）	アプリケーション時間 %（関数のみ）	呼び出し数
NU.OJL.MPRTOS.TLV.Core.LogData.get_Time()	1,604.13	1,604.13	3.23	3.23	6,226,568
NU.OJL.MPRTOS.TLV.Core.LogData.get_Object()	984.26	984.26	1.98	1.98	6,174,251
NU.OJL.MPRTOS.TLV.Core.LogData.set_Id(int64)	852.17	852.17	1.72	1.72	6,270,887
NU.OJL.MPRTOS.TLV.Core.LogData.get_Id()	91.7	91.7	0.18	0.18	706,733
NU.OJL.MPRTOS.TLV.Core.LogData.get_Type()	74.69	74.69	0.15	0.15	433,889
NU.OJL.MPRTOS.TLV.Core.LogData..ctor(valuetype	14.79	9.78	0.03	0.02	44,592
NU.OJL.MPRTOS.TLV.Core.LogData.CheckAttribute(	2.12	2.09	0	0	19
NU.OJL.MPRTOS.TLV.Core.LogData.Equals(class NU	0.48	0.12	0	0	556
NU.OJL.MPRTOS.TLV.Core.LogData.Equals(class NU	0.36	0.22	0	0	556
総計		3619.16		7.28	

第 6 章

おわりに

6.1 まとめ

本 OJL では，トレースログ可視化ツールである TLV の開発を継続して行ない，TLV に対して機能追加とリファクタリングを行なった．

TLV を開発された背景として，組込みシステムにおいてもマルチコアプロセッサの利用が進んでおり，それに伴い従来のデバッグ方法が有効でなくなってきたことを述べた．これは，マルチコアプロセッサが各コアで並列処理を行うため，プログラムの挙動が非決定的になり，バグの再現が保証されず，従来のブレークポイントによるステップ実行ではバグを確実に捕らえることができないからである．

一方，マルチコアプロセッサ環境におけるデバッグで有効な方法として，実行中にデバッグを行うのではなく，実行後にトレースログを解析する手法がある．そして，開発者が直接トレースログを扱うのは効率が悪く，トレースログの解析を支援するツールが要求されており，その 1 つとして可視化表示ツールがある．

既存のトレースログ可視化ツールは，標準化されたトレースログを扱っていないため，利用できるトレースログの形式が限られており，汎用性に乏しい．また，可視化表示項目が提供されているものに限られ，変更や追加を行う仕組みも提供されておらず，拡張性に乏しい．

そこで後藤ら [1, 2] によって，汎用性と拡張性を備えたトレースログ可視化ツール TLV が開発された．TLV 内部でトレースログを抽象的に扱えるよう，トレースログを一般化した標準形式トレースログを定め，任意の形式のトレースログを標準形式トレースログに変換する仕組みを変換ルールとして形式化した．トレースログの可視化表現を指示する仕組みを抽象化し，可視化ルールとして形式化した．TLV では，変換ルールと可視化ルールを外部ファイルとして与えることで，汎用性と拡張性を実現している．

本 OJL では，TLV のリリースを複数回行ない，要求の収集を行なった．収集した要求のうち “CPU ごとのタスク表示” に対する要求が強かったため，これらの実現を行なった．また，ユーザアプリケーションのログ出力を可視化するための機能追加を行った．特定のフォーマットに従ったログ出力を受理し，文字列やユーザ定義状態として可視化することで，ユーザアプリケーションの様々な出力を可視化できるようになった．特に，文字列の可視化は汎用性が高く，システムのメッセージや変数値の変化など様々な出力が可能である．

また，TLV の高速化に対する要求も多かったので，第一段階としてプロファイリングによりボトルネックを調査した．今回は標準形式トレースログへの変換部分を対象とし，どのクラスや処理に時間がかかるかを明らかにした．

6.2 今後の課題

変換・可視化の高速化

TLV の変換・可視化の高速化を行なう必要がある．

第 5 章で述べたプロファイリング結果を参考に，同時期に行われたリファクタリング案に従って TLV を抜本的に更新する必要がある．

不安定な動作の修正

現在，TLV が実行中にエラーを出力して動作を中断するバグレポートがしばしば送られている．ソフトウェアが広く利用され，開発に貢献するためには，機能が優れているだけでなく，道具として信頼できる必要がある．そのため，不安定な動作をする箇所を洗い出し，それらに対処する必要がある．

謝辞

TLV を開発するにあたり，ご指導を頂きました名古屋大学大学院情報科学研究科情報システム学専攻の高田広章教授，愛知県立大学大学院情報科学研究科の山本晋一郎教授に深く感謝致します．また，開発プロジェクトマネージャとして日頃より多くのご助言を頂きました同研究室の本田晋也助教，企業出身者としての立場から実践的なご意見を頂きました同研究科付属組込みリアルタイム研究センターの長尾卓哉研究員に深く感謝致します．

参考文献

[1] 後藤隼弐 『OJL によるトレースログ可視化ツールの開発』修士論文, 名古屋大学,2009

[2] 後藤隼弐, 本田晋也, 長尾卓哉, 高田広章『トレースログ可視化ツールの開発』情報処理学会 第 139 回 システム LSI 設計技術 (SLDM) 研究会, 2009 年 2 月 26 日

[3] JTAG ICE PARTNER-Jet，http://www.kmckk.co.jp/jet/，最終アクセス 2009 年 1 月 14 日

[4] WatchPoint デバッガ，https://www.sophia-systems.co.jp/ice/products/watchpoint，最終アクセス 2009 年 1 月 14 日

[5] QNX Momentics Tool Suite, http://www.qnx.co.jp/products/tools/, 最終アクセス 2009 年 1 月 14 日

[6] eBinder，http://www.esol.co.jp/embedded/ebinder.html，最終アクセス 2009 年 1 月 14 日

[7] LKST（Linux Kernel State Tracer) - A tool that records traces of kernel state transition as events, http://oss.hitachi.co.jp/sdl/english/lkst.html，最終アクセス 2009 年 1 月 14 日

[8] Prasad, V., Cohen, W., Eigler, F. C., Hunt, M., Keniston, J. and Chen, B.: Locating system problems using dynamic instrumentation. Proc. of the Linux Symposium, Vol.2, pp.4964, 2005.

[9] Mathieu Desnoyers and Michel Dagenais.: The lttng tracer : A low impact performance and behavior monitor for gnu/linux. In OLS (Ottawa Linux Symposium) 2006, pp.209224, 2006.

[10] R. McDougall, J. Mauro, and B. Gregg.: Solaris(TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris. Pearson Professional, 2006.

[11] Mathieu Desnoyers and Michel Dagenais.: OS Tracing for Hardware, Driver and Binary Reverse Engineering in Linux. CodeBreakers Journal Article, vol.4, no.1, 2007.

[12] OpenSolaris Project: Chime Visualization Tool for DTrace, http://opensolaris.org/os/project/dtrace-chime/，最終アクセス 2009 年 1 月 14 日

[13] RFC3164 The BSD syslog Protocol, http://www.ietf.org/rfc/rfc3164.txt，最終アクセス 2009 年 1 月 14 日

[14] TOPPERS Project，http://www.toppers.jp/，最終アクセス 2009 年 1 月 14 日

[15] Takuya Azumi and Masanari Yamamoto and Yasuo Kominami and Nobuhisa Takagi and Hiroshi Oyama and Hiroaki Takada.:A New Specification of Software Components for Embedded Systems. Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2007), pp.45-50, 2007