The Parallel Universe

intel® Software

Issue 14
2013 年 6 月

インテル® System Studio:

インテリジェント・システム向け総合開発ソリューション

Noah Clemons

Let performance thrive.

Intel® Technical Webinar Series

intel® Software

Free Intel Webinar for C, C++, and Fortran Developers

Join technical experts for our biannual webinar series, featuring techniques and tools that can help optimize your applications, and increase performance and reliability.

Look for a special, two-day webinar on the new Intel® Xeon Phi™ coprocessor, June 25-26, 2013. This coprocessor, based on Intel® Many Integrated Core (MIC) architecture, is enabling dramatic performance gains on demanding applications.

intel® inside™ Xeon® intel® inside™ Xeon Phi™

SEE UPCOMING TECHNICAL WEBINARS AND REGISTER NOW ›

Embedded and Mobile Systems Developers

Attend focused webinars with the insight that can help accelerate time to market and reliability.

SEE THE ABSTRACTS AND REGISTER ›

© 2013 Intel Corporation. 無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、Intel Atom、Intel Core、Intel Inside ロゴ、Cilk、VTune、Xeon、Xeon Inside、Intel Xeon Phi は、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

目次

編集者からのメッセージ
開発ツールの分岐点
JAMES REINDERS 5

インテル® System Studio: インテリジェント・システム向け総合開発ソリューション
NOAH CLEMONS および ROBERT MUELLER-ALBRECHT 7

ハードウェアとソフトウェアの深い洞察を提供する、新しい統合ソフトウェア開発スイートについて説明します。Wind River Linux* や Yocto Project* など、組込み Linux* を搭載するインテル® プロセッサー・ベースの組込みデバイスおよびインテリジェント・システムの開発、テスト、最適化を促進します。

ショックウェーブ! CloverLeaf とインテル® Xeon Phi™ コプロセッサーの巡り合い
VICTOR GAMAYUNOV および STEPHEN BLAIR-CHAPPELL 25

CloverLeaf ベンチマークを「インテル® Xeon Phi™ コプロセッサー対応」にして大幅なパフォーマンス向上を達成する手順を説明します。

インテル® Xeon Phi™ コプロセッサーとインテル® MPI ライブラリーのプログラミング・モデル
MICHAEL CHUVELEV、DMITRY DURNOV、SERGEY KAZAKOV、ALEXANDER SUPALOV、ANDREW TANANAKIN 29

既存のハイパフォーマンス・コンピューティング (HPC) アプリケーションのインテル® Xeon Phi™ コプロセッサー搭載プラットフォームへの移植を促進する、インテル® MPI ライブラリーと単純なユーザー・インターフェイスを組み合わせた新しいプログラミング・モデルについて説明します。

従来のシリアル処理から並列処理へ: インテル® Cilk™ Plus による実用アプリケーションの変換
CHRIS CHRULSKI 33

シリアル処理の最適化とインテル® Cilk™ Plus の並列処理を併用することにより、従来のシリアル・アプリケーションのコードを大幅に変更せずに、8 コアのシステムで重要なワークロードの実行速度を 25 倍も向上させた実例を紹介します。

©2013 Intel Corporation. 無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、Intel Atom、Intel Core、Intel Inside ロゴ、Cilk、VTune、Xeon、Xeon Inside、Intel Xeon Phi は、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。*その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

開発ツールの分岐点

編集者からのメッセージ

James Reinders インテル コーポレーションの並列プログラミング・エバンジェリスト兼ディレクター。Morgan Kaufmann から出版された 2 冊の新しい書籍、『Intel® Xeon Phi™ Coprocessor High Performance Programming』（2013）および『Structured Parallel Programming』（2012、日本語:『構造化並列プログラミング』）の共著者。このほかに、『Intel® Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism』（O' Reilly Media、2007、日本語:『インテル スレッディング・ビルディング・ブロック ―マルチコア時代の C++ 並列プログラミング』、中国語、韓国語の翻訳版があります）や『VTune™ Performance Analyzer Essentials』（Intel Press,2005）などの文献を発表しています。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

最近の開発ツールは複雑な組込みシステムやモバイルシステムに、先例のないクロスプラットフォーム開発、アプリケーション移植と移行、ソフトウェア革新をもたらしました。その潜在的なパフォーマンスと能力は非常に大きなものです。この号では、「どこでも動作する」アプリケーションを記述する場合でも、HPC アクセラレーションを行う場合でも、アプリケーションのパフォーマンスと能力を高める上で役立つさまざまな手法を紹介します。

特集記事である「インテリジェント・システム向け総合開発ソリューション: インテル® System Studio」では、組込みおよびモバイル Linux* を搭載するインテリジェント・システム向けの新しいクロス開発ツール・スイートについて解説します。このスイートは、最近の組込みデバイスやモバイルシステムで利用されている、高度に統合された複雑なチップセット向けに最適化されたカスタム・システム・ソフトウェア・スタックおよびアプリケーション・レイヤー・コンポーネントの開発を促進します。

「ショックウェーブ! CloverLeaf とインテル® Xeon Phi™ コプロセッサーの巡り合い」では、インテル® Xeon Phi™ コプロセッサーを最大限に活用するため、スタッガード・デカルト格子上で 2 次元の圧縮性オイラー方程式を解く小さな CloverLeaf ベンチマークをベクトル化します。このベクトル化により、パフォーマンスは約 80% も向上しました。

「インテル® Xeon Phi™ コプロセッサーとインテル® MPI ライブラリーのプログラミング・モデル」では、インテル® MPI ライブラリー Linux* 版を利用して既存のハイパフォーマンス・コンピューティング (HPC) アプリケーションをインテル® Xeon Phi™ コプロセッサー搭載プラットフォームへ移植する方法を説明します。

「従来のシリアル処理から並列処理へ: インテル® Cilk™ Plus による実用アプリケーションの変換」では、インテル® Cilk™ Plus を利用して従来の実用シリアル・アプリケーションを並列アプリケーションに変換する際の高水準ワークフローと変換結果について解説します。

新しいツールは、インテル® Xeon Phi™ コプロセッサーのようなクロスプラットフォームのアプリケーションを開発したり、ユーザーの期待を明確にする上で役立つでしょう。開発ツールの次の峠を越えて優れた能力と革新を手にすることができるように、今後も開発者の皆さんをサポートしてまいります。

James Reinders
2013 年 5 月

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

インテル® System Studio: インテリジェント・システム向け総合開発ソリューション

Noah Clemons
インテル コーポレーション
DPD エンベデッド・コンピューティング、デバッガー、
ライブラリー・グループ
テクニカル・コンサルティング・エンジニア

Robert Mueller-Albrecht
インテル コーポレーション
DPD エンベデッド・コンピューティング、デバッガー、
ライブラリー・グループ

インテル® System Studio は、組込みおよびモバイル Linux* を搭載するインテリジェント・システム向けのクロス開発ツール・スイートです。このスイートは、最近の組込みデバイスで利用されている高度に統合された複雑なチップセット向けに最適化されたカスタム・システム・ソフトウェア・スタックおよびアプリケーション・レイヤー・コンポーネントの開発を促進します。この記事では、最近の組込みソフトウェア開発者のニーズに焦点を当てた、この魅力的な新しい製品について説明します。インテル® System Studio を構成するコンポーネントの技術的な概要を紹介した後、それらのコンポーネントをインテリジェント・システムや従来の組込みデバイスに適用する方法をについて解説します。

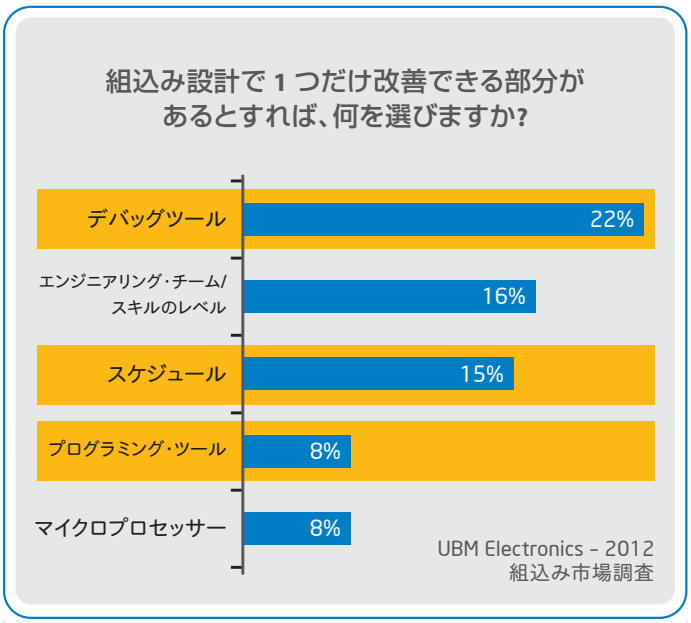


図 1. UBM Electronics – 2012 組込み市場調査

組込みツール・スイートの中で最も改善を希望する部分を組込み開発者に質問した最新の調査では、デバッグツールがトップにランクされました。インテル® System Studio は、基礎となるデバッグ環境を提供するツール・スイートであり、以下のツールが統合されています。

- システム・ソフトウェア・レイヤー向け JTAG デバッガー
- アプリケーション・デバッガー (GDB)
- 低オーバーヘッドの全 SoC データイベントのトレース
- SoC 全体の消費電力およびパフォーマンス・プロファイリング・ツール
- メモリーチェック/スレッド・プロファイリング・ツール
- 業界標準の C/C++ コンパイラー
- パフォーマンス・ライブラリー

インテル® System Studio のコンポーネントの多くは、Eclipse* 統合開発環境 (IDE) に統合されます。組込みシステム向けソフトウェアは、ハードウェア・ベンダーのプラットフォーム固有の反復処理 (制御) により動作が頻繁に制限されます。開発者は、対象ハードウェアまたはソフトウェア環境の制限のために移植性を放棄する傾向にあります。

インテル® System Studio を利用すると、インテル® 開発ツールの利用範囲が拡大され、開発者はこれらの制限を克服できます。組込み計算の開発者向けに、共通標準/オープン標準、業界標準、インテルの広範なエコシステムおよびコンピュート・コンティニュアムを利用するオプションが提供されます。

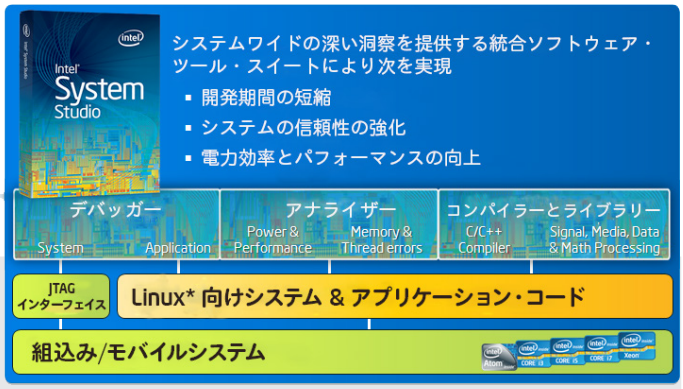


図 2. インテル® System Studio の概要

図 2 は、プラットフォーム・ソフトウェア・スタックにわたる組込みソフトウェア開発の主要な要素 (洞察、コンフィギュアビリティー、信頼性、パフォーマンスを含む) の中でインテル® System Studio が焦点を当てている要素を示しています。これらの要素をデバッグ・ソリューションとアナライザーで調べた後、組込みクロスビルド環境でコンパイラーとパフォーマンス・ライブラリーを使用してパフォーマンスをチェックします。

インテル® System Studio のインテリジェント・システム向けデバッガー

組込みクロス開発に焦点を当てたインテル® System Studio には、ホスト側のインストールだけでなく、リモートデバッグとデータ収集用に (コンポーネントをターゲットに配置する) `system_studio_target.tgz` パッケージも用意されています。

デバッグツールは次の 3 つのカテゴリーに分けられます。

1. メモリー/コード正当性検証ツール
2. 高水準言語アプリケーション・デバッグ・ツール
3. システム・ソフトウェア・デバッガー

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

1 番目と 2 番目は、ソフトウェア・スタック・コンポーネントのリアルタイム・デバッグを行います。デバッグツールの 3 番目の種類は、インストルメンテーション・ベースのイベント・トレース・ユーティリティーです。このユーティリティーは、複数の IP ブロックを備え、デバイス間で短時間でデータを渡す、ヘテロジニアス・マルチコア SoC チップセットの複雑な対話処理のデバッグに役立ちます。インテル® System Studio には、アプリケーション・デバッガーとして **GNU* プロジェクト・デバッガー**、システムデバッガーとして**インテル® JTAG デバッガー**が含まれています。また、ソフトウェア・デバッグ用の超低オーバーヘッド・インストルメンテーション手法である **SVEN (Software Visible Event Nexus)** も含まれています。

インテル® System Studio は、広範な Linux* ホストと次の JTAG デバイスをサポートします。

- Macraigor Systems* usb2Demon* デバイス
- インテル® ITP-XDP3

インテル® System Studio に含まれているインテル® JTAG デバッガー (図 3) は、Windows* ホストをサポートします。

問題を素早く検出する高度なシステム / アプリケーション・レベル・デバッガー	
インテル® JTAG デバッガーシステムとアプリケーション・デバッガー	全レジスターの内容を含む、詳細な CPV、SoC、チップセットのデバッグ
	OS カーネル・ソフトウェア、BIOS、UEFI、ファームウェア、ドライバーのソースレベルのデバッグ
	超高速ソフトウェア・イベント・トレーシング・インフラストラクチャー
	GDB* ベースの拡張アプリケーション・デバッガー

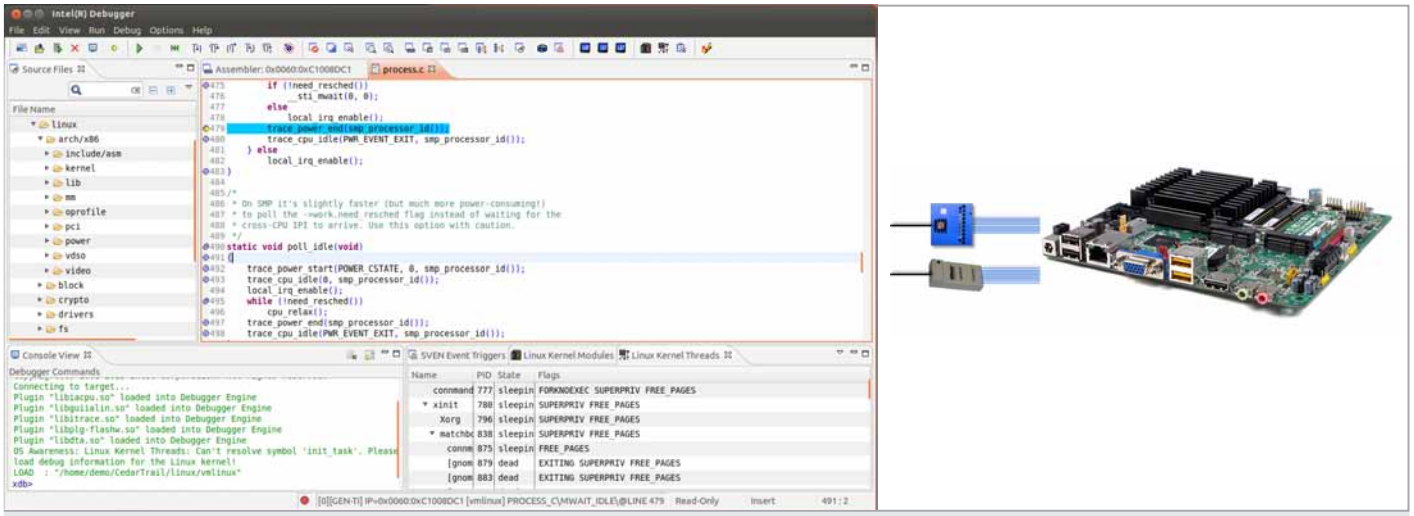


図 3. インテル® JTAG デバッガー

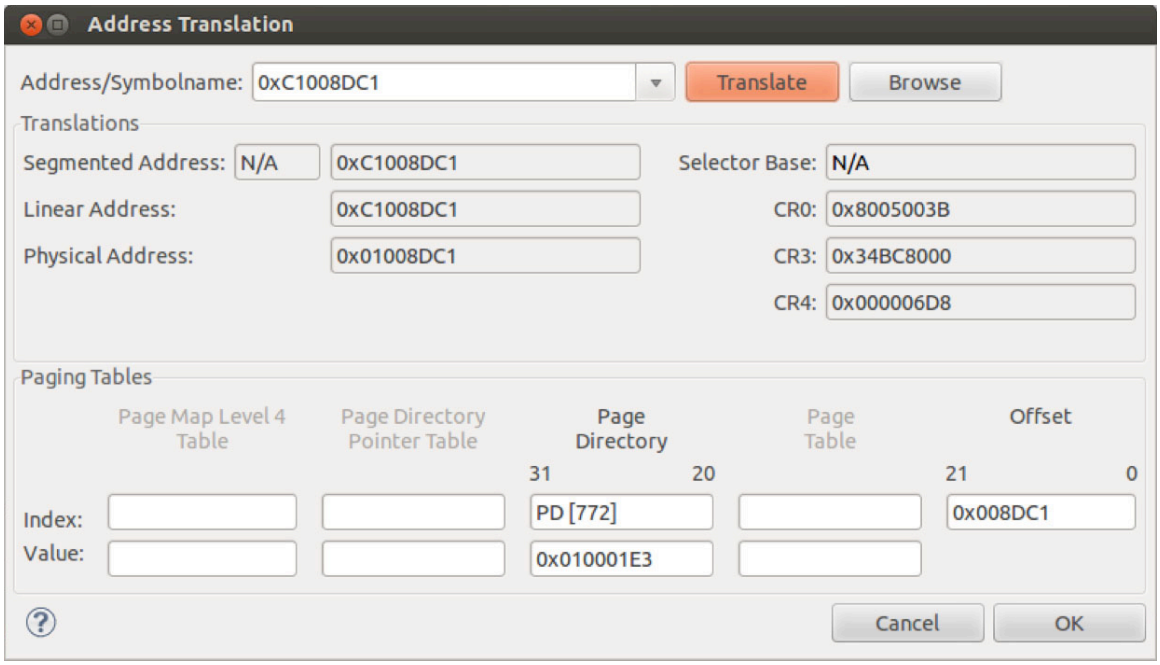
インテル® JTAG デバッガーによるファームウェア、ブートローダー、OS レイヤーのデバッグ

組込みソフトウェア・スタックの各レイヤーのデバッグには異なるアプローチが必要です。まず、各ソフトウェア・レイヤー (ファームウェアからアプリケーション・レイヤーまでのすべてのレイヤー)、一般的な課題、デバッグアプローチを見ていきましょう。

初期のハードウェア構成とボード向けに、インテル® System Studio は Eclipse* RCP ベースのユーザー・インターフェイスを利用した JTAG デバッグをサポートしています。このインターフェイスには、次のような機能が備わっています。

- デバイス制御、プロセッサー・ステータス、ソフトウェア・ステータスを管理するレジスター用のビットフィールド・エディター。このエディターは各ビットの状態を表示します (図4)。
- ページテーブルの視覚化。仮想アドレスが物理メモリーにどのようにマップされるかを示します。
- グローバル・ディスクリプター・テーブル (GDT) (メモリーセグメントの実行の可能性と書き込みの可能性を示す) およびローカル・ディスクリプター・テーブル (LDT) (特定プログラムのメモリーセグメントを保持) へのアクセス。
- OS メモリー管理と構成を考慮した不完全なデータ割り当ての識別。

ファームウェア、ブートローダー、OS、デバイスドライバーと、ブートプロセスに従ってデバッグを行います。次に、インテル® JTAG デバッガーによるファームウェアのデバッグの様子を示します (図 4)。



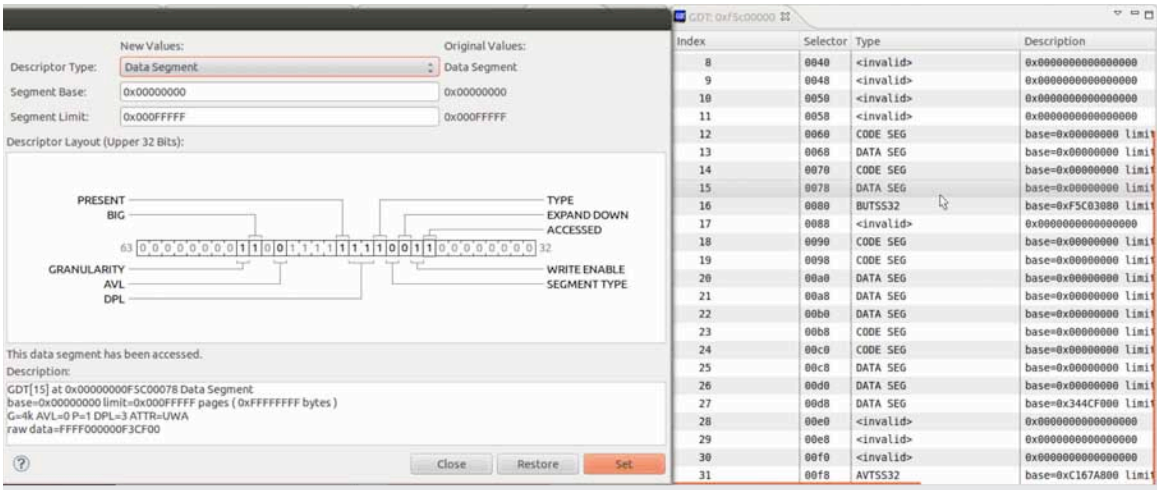


図 4. インテル® JTAG デバッガーでデバイスレジスターとメモリーマップを視覚化

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

UEFI (Unified Extensible Firmware Interface) のデバッグ

特に必要な周辺機器のみをサポートするように、デバイスのファームウェアをカスタマイズしてブート時間を高速化するには、インテル® Firmware Support Package やインテル® Bootloader Development Kit のような UEFI 互換のファームウェア・ソリューションを利用します。UEFI 環境では再配置可能なコードモジュールが使用されており、これらのモジュールのアドレスは通常エンドユーザーには分からないため、ファームウェアの問題のデバッグは大きな課題です。UEFI ファームウェアは Microsoft* C/C++ コンパイラーまたはインテル® C/C++ コンパイラーでコンパイルすることができます。ファームウェアのデバッグは、解決されソースコードにマップされる PDB シンボル情報に依存するため、ファームウェアのデバッグには (インテル® System Studio の一部である) Windows* ホストバージョンのインテル® JTAG デバッガーを利用する必要があります。

インテル® JTAG デバッガーでは、メモリーのコードモジュールを特定する 2 つの一般的なメソッドが提供されます。

1. 特定のメモリーアドレス (例えば、命令ポインター) に位置する特定のモジュールを識別します。
2. EFI ランタイムで既知のモジュールをすべてリストし、シンボルをロードする特定のモジュールを選択します。

最初のメソッドは、特定のモジュールにのみ依存します。デバッガーは指定されたアドレスに近いメモリーをスキャンし、シンボルをロードするのに十分なデータを含むモジュールヘッダーを探します。

特定のアドレスのシンボルをロードするには、EFI “LOADTHIS” コマンドを指定します。このコマンドは、指定されたアドレスのモジュールを識別して、そのシンボルをロードします。アドレスが指定されていない場合、現在の命令ポインターが使用されます。

```
xdb> efi “loadthis”

INFO: Software debugger set to: efi64 - EFI/PI compliant BIOS (64-bit mode)

INFO: Using DRAM search semantics, align=0x00001000 range=0x00100000

INFO: Searching backwards from 0x00000000809FB6C3 to 0x00000000808FB6C3 for PE/COFF header

INFO: Found PE/COFF module at 0x00000000809FB000 - 0x00000000809FF1C0 (size: 16832 bytes)

INFO: Loading debug symbols found at:

e:\dev.efi\work\Build\MdeModule\NOOPT_VS2008x86\X64\MdeModulePkg\Application\xdbefiutil\xdbefiutil\

DEBUG\xdbefiutil.efi
```

2 番目のメソッドは、メモリー (またはフラッシュ) で EFI データ構造を特定し、そこで示されたロード済みモジュールのリストを列挙します。この処理は、EFI システム・テーブル・ポインターやフラッシュボリュームのような特定のデータ構造の経験的な知識に依存します。

デバイスドライバー

多くの場合、デバイスドライバーはタイミングに注意する必要があるため、デバイスドライバーのデバッグには別の課題が発生します。例えば、ドライバーコードにインストルメンテーションを追加すると、動作が変わることがあります。この課題に取り組むため、インテル® System Studio では、インストルメントなしでデバッグできるように、OS およびドライバーを考慮したカーネルモジュールをターゲットデバイスにロードします。デバイスドライバーのロード時に、このカーネルモジュールは JTAG インターフェイス経由でドライバーの初期化と破棄のメモリー位置をホストにエクスポートします。デバイスドライバーのシンボル情報をロードし、情報にステップインして、コードを修正またはインストルメントすることなく実行フローをデバッグできるため、ドライバーコードのタイミング動作を変更するリスクを回避できます。

さらに、ビットフィールド・エディターは、パブリック・デバイス・レジスターにアクセスして、デバイスドライバーのデバッグ中にデバイス構成レジスターのエントリーをモニターすることができます。

命令トレース

ソースのエラーをトレースする能力はデバッグの基本です。インテル® System Studio は、実行フローをアンロールしてランタイム問題の根本的原因を特定する、高度な命令トレース機能を提供します。具体的には、最終分岐レコード (LBR) を検査してコードを逆アセンブルし、プログラムフローを再作成します。次に、アセンブリー命令と関連するソースコード (組込み Linux* の場合は ELF Dwarf 実行ファイルから取得) をペアにして、生成されたトレースをデバッガーに表示します (図 5)。

トレースはリアルタイムのパフォーマンスに影響を与えないため、スタック・オーバーフローやセグメンテーション・フォルトのような決定性のある再現可能なエラーを追跡する強力なツールです。次のような操作を行えます。

> OS シグナル・イベント・ハンドラー内のブレークポイントを設定する (セグメンテーション・フォルトでのブレークなど)。
> イベントの地点まで実行フローをアンロールする。
> 予想に反した地点まで実行を遡る。
> その地点まで再実行してメモリーアクセスを解析する。

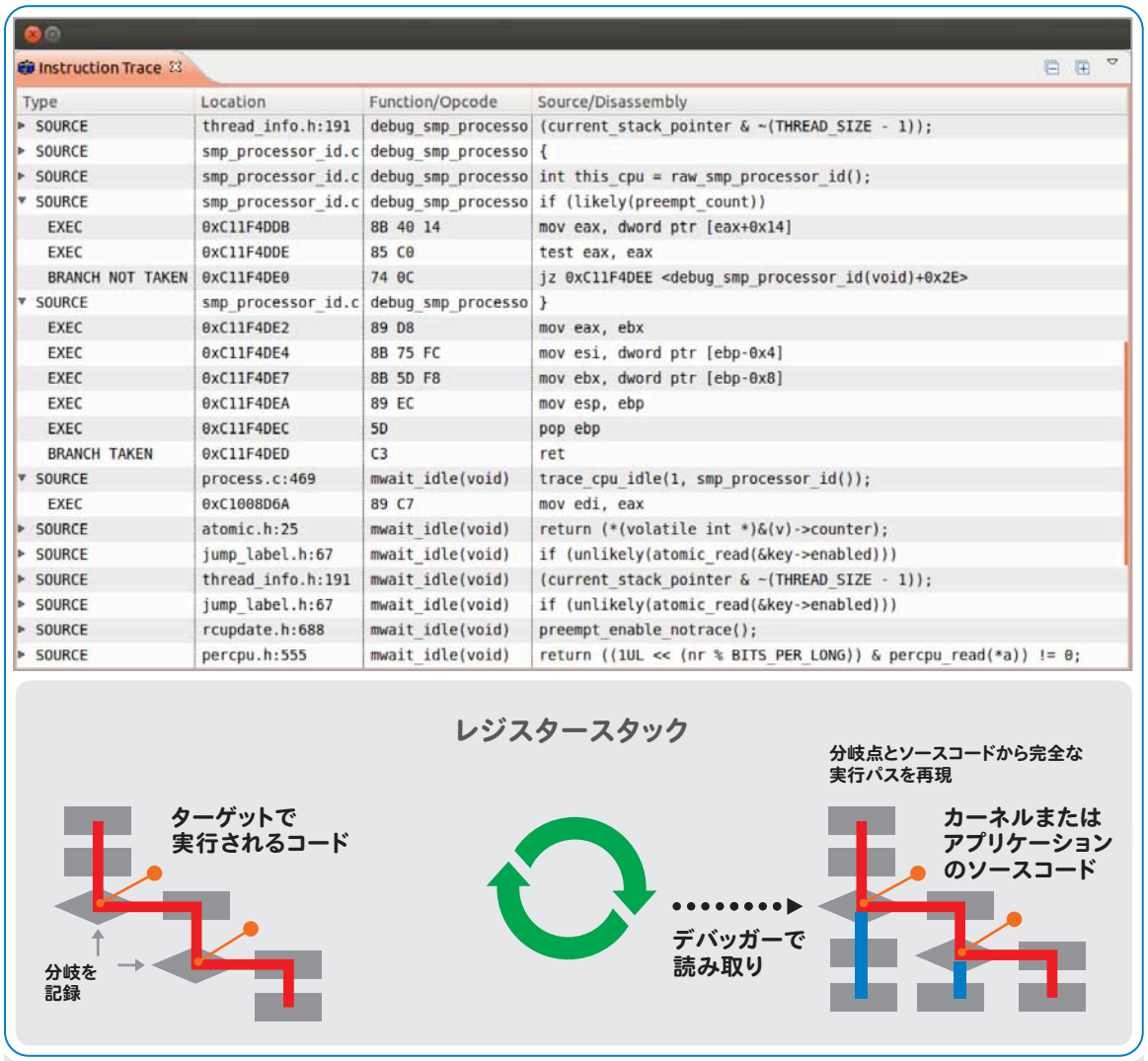


図 5. 命令をトレースして問題フローを再作成

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

SVEN による非決定性問題と再現が困難な問題のデバッグ

最近のインテリジェント・システムは、極度にスレッド化されている場合、またはソフトウェア・モジュール間のメッセージ/データ・パッシング・イベントに依存している場合、正確なタイミングに非常に敏感です。このようなシステムでは、再現が困難な非決定性問題がよく発生するため、デバッグは困難です。コードのデバッグがソフトウェア・スタックのタイミングに影響を与えたり、アプリケーションの動作を変更することがあります。また、デバッグセッション中に問題が再現できないこともあります。この特異な問題は「ハイゼンバグ (heisenbug) 」と呼ばれます。実際にデバイスが (容易にアクセスできない場所に) 配備されているときにのみ問題が発生する場合は、特にデバッグが困難でしょう。

多くのハイゼンバグ問題に対するソリューションとして、インテル® System Studio には SVEN ソフトウェア開発キット (SDK) テクノロジー・プレビューと呼ばれる、新しいコンポーネントが用意されています。SVEN は、イベント ID、タイムスタンプ、イベントタイプ、ペイロードを含む小さなインストルメンテーション構造としてデータイベントを定義します。プログラム実行フローのあらゆる種類の変数伝播やメッセージパッシングを SVEN イベントと関連付けて追跡することができます。SVEN は、スタティック・コードのインストルメンテーションとタイミング・オーバーヘッドが 5μs 以下の小さな DRAM バッファー[1] を利用するため、ハイゼンバグが発生する可能性を最小限に抑えます。SVEN により、従来の方法では見つけられない、タイミングに依存するランタイムの問題を発見することができます。製品版コードにインストルメンテーション・コードを残しても、ロギングを有効にしなければ実行に影響しないため、SVEN はアクセスが困難な場所に配置されたアプリケーションのオフラインデバッグに最適です。

当初、インテル® Atom™ プロセッサー CExxxx ベースのプラットフォーム向けに開発された SVEN は、現場で実証されたテクノロジーであり、現在はすべてのインテル® アーキテクチャーで利用できます。SVEN はチップセット全体にわたって非同期メッセージとデータイベントを追跡できます。高度に設定可能で、システム・ソフトウェアとアプリケーションの両方のインストルメンテーションに使用できます。デバッグを開始するには、イベントタイミングと関連付けるための信頼できるクロック信号が必要です (図 6)。

インテル® System Studio には、オープンソース SDK 形式の SVEN フレームワークと、イベントやタイミングを容易に操作して不規則性を迅速に識別できるグラフィカル・トレース・ビューアーが含まれています。また、JTAG サポートにより、データ・ブレークポイントを用いてシステムレベルのデバッグが可能です。この機能は、任意の SVEN イベントでトリガーし、疑わしいイベントからステップ実行してデバッグできます。次のようなブレークポイントを含みます。

- USB ドライバーからのイベントでブレーク。
- "ERROR" で始まるデバッグ文字列でブレーク。
- レジスター X がアクセスされた場合のブレーク。
- レジスター X の特定ビットが特定値の場合のブレーク。

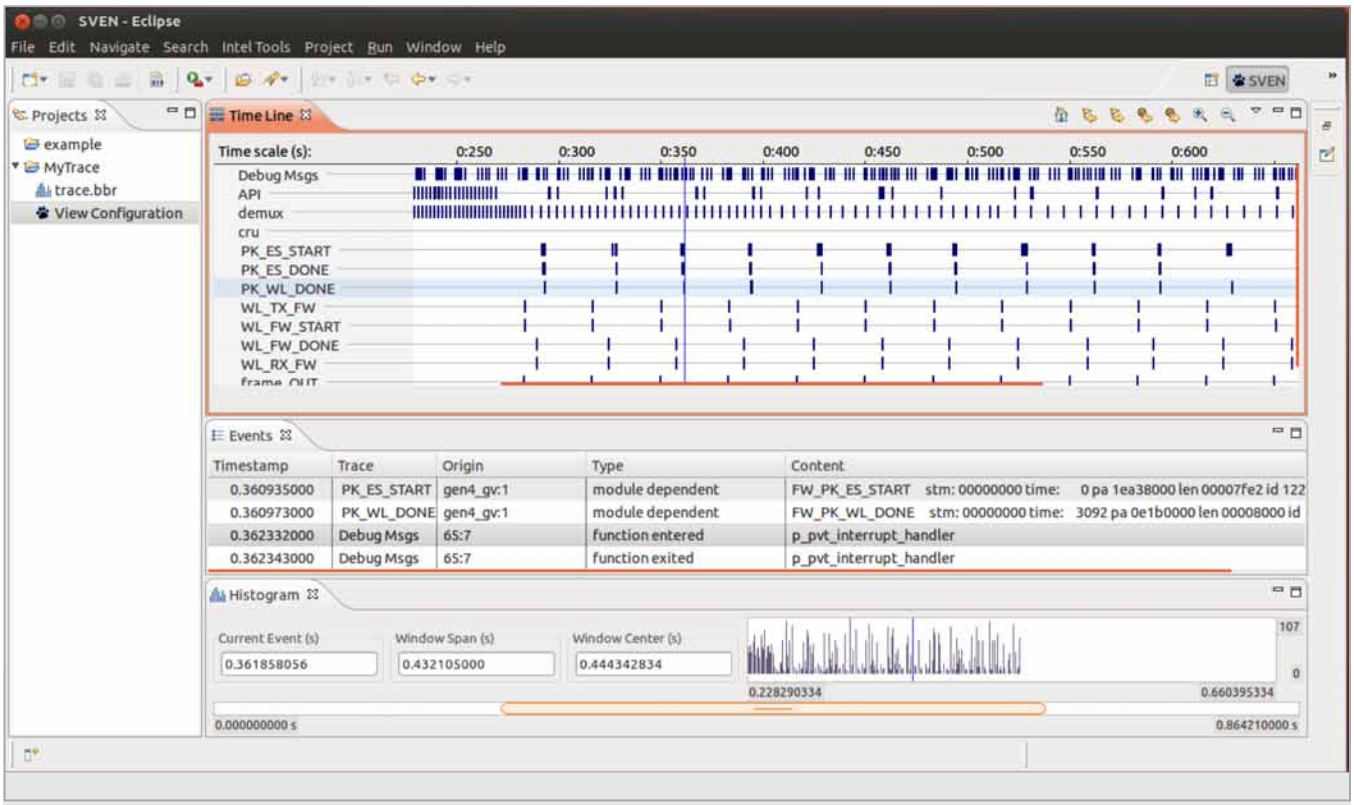


図 6. SVEN でソフトウェア・スタック間のイベントを視覚化

GDB によるアプリケーション・デバッグ

インテル® System Studio には、インテル® アーキテクチャー・ベース・プラットフォームのデータ競合検出機能が追加された独自の GNU プロジェクト・デバッガー (GDB) と、インテル® Atom プロセッサー・ベースの環境で BTS (Branch Trace Storage) ベースのプロセッサー固有命令フローをアンロールする機能が用意されています。この拡張 GDB を、Linux* ディストリビューションで提供されているデフォルトデバッガーの代わりに使用するには、次のデバッガー環境設定スクリプトを `source` コマンドで読み込みます。

```
<install-dir>/system_studio_2013.0.xxx/
debugger/gdb/bin/debuggervars.sh
```

標準 GDB と同様、この拡張 GDB は Eclipse* IDE に統合でき、リモート・システム・エクスプローラー (RSE) プラグイン (http://download.eclipse.org/tm/downloads/) をサポートします。

Eclipse* 内から gdbserver ベースのクロスデバッグを実行するには、次の操作を行います。

1. 製品のインストール時に提供される gdbserver (`<install-dir>/system_studio_2013.0.xxx/debugger/gdb/<arch>/<python>/bin/`) をターゲットシステムにコピーして、ターゲットの PATH 環境変数に追加します。
2. 正しい GDB 実行ファイルの場所を指すように Eclipse* を設定します。
   a. Eclipse* IDE のプルダウンメニューから、[Window (ウィンドウ)] > [Preferences (設定)] をクリックします。
   b. [Preferences (設定)] ダイアログが表示されたら、左のツリービューから [C++] > [Debug (デバッグ)] > [GDB] を選択します。
   c. [GDB debugger (GDB デバッガー)] テキストボックスを編集して GDB 実行ファイルを選択します。
3. `<install-dir>/system_studio_2013.0.xxx/debugger/gdb/<arch>/<python>/bin/` を指定します。アーキテクチャーおよび Python* のバージョンに応じて、`<arch>` には `ia32` または `intel64`、`<python>` には `py24`、`py26`、`py27` のいずれかを指定します。

「インテル® System Studio は開発、テスト、最適化を促進するハードウェアとソフトウェアの深い洞察を提供します。」

GDB を使用した組込みデバイスにおけるアプリケーションのデバッグ

このデバッグに関する詳細は、「GDB: The GNU* Project Debugger」 (http://www.gnu.org/software/gdb/) (英語) を参照してください。GDB には、クロス開発用の gdbserver と呼ばれるリモート・デバッグ・エージェントが含まれます。このデバッグ・エージェントは起動するデバッグターゲットにインストールして、開発ホストからデバッガーにリモートにアタッチすることができます。開発に使用したホストと異なるターゲットホストでプログラムを実行しなければならない場合、特にターゲットのリソース (CPU およびメモリーのいずれか) が制限されている場合にエージェントは便利です。その状況では、次のようにデバッガーを使用します。

- ターゲットマシンで gdbserver を使用してプログラムを開始します。
- gdbserver はプログラムの実行をエントリーポイントで中断し、デバッガーがプログラムに接続するまで待ちます。
- 次のコマンドは、アプリケーションを開始し、ローカルホストのポート 2000 でデバッガーとの接続を待つように gdbserver に指示します。

```
$ gdbserver localhost:2000 program

Process program created; pid = 5685

Listening on port 2000
```

gdbserver が待ち受け状態になると、この gdbserver との接続を確立し、(あたかも GDB の管理下で、プログラムが同じホスト上で直接デバッグされているかのように) 同じデバッグセッションを開始することができます。

```
$ gdb program

     (gdb) target remote targethost:4444

     Remote debugging using targethost:4444

     0x00007f29936d0af0 in ?? ()
from /lib64/ld-linux-x86-64.so.

     (gdb) b foo.adb:3

     Breakpoint 1 at 0x401f0c:
file foo.adb, line 3.

     (gdb) continue

     Continuing.

Breakpoint 1, foo () at foo.adb:4

          4        end foo;
```


インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

すでに実行しているプログラムにアタッチして gdbserver を使用することもできます。その場合、デバッガーと gdbserver の接続が確立されるまで、プログラムの実行は中断されます。構文は次のようになります。

```
$ gdbserver localhost:2000 --attach 5685

to tell gdbserver to wait for GDB
to attempt a debug connection to the
running process with process ID 5685
```

データ競合の検出

インテル® System Studio の GDB 拡張ビルドは、データ共有イベントおよび可能性のあるデータ競合でリアルタイムにブレークすることができます。唯一必要なことは、インテル® C++ コンパイラーの -debug parallel オプションを指定してコードをコンパイルしておくことです。スマート検出フィルターを利用すると、データ競合になる可能性のあるスレッド化部分に注目して、エラーの正確な位置を特定できます。

データ競合は、複数のスレッドが同期なしでオーバーラップされたメモリーにアクセスするときに発生します。データ競合が問題ない場合や設計の一部である場合もありますが、データ競合は通常、バグとして見なされます。GDB は、インテル® コンパイラーの一部である並列デバッグ拡張 (PDBX) データ競合検出器のフロントエンドとして使用できます。PDBX データ競合検出器は、コンパイラー・インストルメンテーションとランタイム・サポート・ライブラリーで構成されます。どちらもインテル® コンパイラーにより提供されます。-debug parallel オプションでコンパイルすることにより、PDBX ランタイム・ライブラリーは、検出されたデータ競合と通信し、解析を設定するためのデバッガー・インターフェイスを提供します。

PDBX データ競合検出器は、すべてのスレッドのメモリーアクセスと同期をすべて記録し、メモリーアクセスごとにデータ競合を調べます。この方法は、パフォーマンスおよびメモリー消費の両方の点で非常にコストがかかります。コストを軽減するには、データ競合解析を微調整します。同期イベントのロギングは制御できませんが、メモリーアクセスのロギングは制御可能です。データ競合検出器は同期を常にロギングしますが、メモリーアクセスはデータ競合検出器が有効な場合のみロギングされるため、いつでも無効にして再度有効にすることができます。もちろん、検出器で検出できるのは既知のメモリーアクセスにおけるデータ競合のみです。さらに、データ競合検出器で収集したメモリーログを破棄することもできます。メモリーに制限があるときにプログラムの個別の部分のデータ競合をデバッグする際に効果的です。

アプリケーションの一部を無視するようにデータ競合検出器を設定すると、さまざまな使用状況で役立つでしょう。

- 誤検出を無視する。データ競合検出器は正しく同期されたアクセスをデータ競合として誤って報告することがあります。通常は、データ競合検出器にとって不明な同期構造が原因です。部分的にインストルメントされたアプリケーションが原因である場合もあります。
- 意図的なデータ競合や問題ないデータ競合を無視する。問題ないデータ競合や意図的なデータ競合もあります。例えば、すべてのスレッドが同じ値や等価な値を書き込むことが保証されている場合のデータ競合は問題ありません。一般に、このような問題のないデータ競合を無視するほうが、スレッドの同期を行うより効率的です。
- 無関係なデータ競合を無視する。データ競合検出器は現在調べているバグと無関係なデータ競合を報告することがあります。この種のデータ競合は現時点では無視できます。
- データ競合解析のパフォーマンスを向上させる。プログラムの一部を無視すると、解析のパフォーマンス・オーバーヘッドが小さくなることがあります。プログラム全体を解析するのではなく、プログラムの一部を繰り返し解析する場合に効果的です。
- データ競合解析のメモリー消費を抑える。プログラムの一部を無視すると、解析のメモリー消費が少なくなり、プログラム全体を解析できなかったケースでデータ競合を検出できることがあります。

分岐命令のトレース

インテル® Atom プロセッサーには、分岐のログを OS 提供のリングバッファーに格納する、BTS (Branch Trace Storage) と呼ばれる機能が用意されています。GNU/Linux* オペレーティング・システムは、perf_event インターフェイスの一部として、(バージョン 2.6.32 以降で) この機能をサポートしています。

分岐トレースの gdb 拡張は、ハードウェア BTS 機能をベースにしており、直ちにクラッシュしない問題のデバッグに便利です。特に、アンワインド機能に支障をきたすスタックの破壊のように、ほかのデバッガー機能では失敗するバグのデバッグに役立ちます。gdb 分岐トレースコマンドを使用して、次のようにプログラム制御フローを記録し、記録した分岐トレースを表示することができます。

- シーケンシャル実行ブロックのリスト (リストビュー)。
- リストされたブロックの 1 つの逆アセンブリー。

分岐トレースはリバースデバッグほど強力ではありませんが、非常に高速です。リストビューで現在の位置を素早く把握することもできます。バックトレース・コマンドと同等であるため、最適化の影響や OS シグナルハンドラーによる切り捨てでプログラムのコールスタックが壊れる場合、バックトレースの代わりに利用できます。

例: プログラムがクラッシュし、バックトレースが有効でない場合。

```
(gdb) run

Starting program:../gdb/trace/examples/function_pointer/stack64  Program
received signal SIGSEGV, Segmentation fault.0x000000000000002a in ?? ()
(gdb) bt
#0  0x000000000000002a in ?? ()
#1  0x0000000000000017 in ?? ()
#2  0x000000000040050e in fun_B (arg=0x4005be) at src/stack.c:32
#3  0x0000000000000000 in ?? ()
```

分岐トレースを確認します。

ブロックのリストは、最新のブロックから開始 (現在の pc で終了) して、ブロック n+1 からブロック n のように制御フローのより古いブロックに続いています。

```
(gdb) btrace list 1-7
in ?? ()
in fun_B () at src/stack.c:36-37
in fun_B () at src/stack.c:32-34
in main () at src/stack.c:57
in fun_A () at src/stack.c:22-25
in fun_A () at src/stack.c:18-20
in main () at src/stack.c:51-56

from main(), we called first fun_A() and then fun_B().The call to
fun_A() returned, and we crashed somewhere in fun_B().
```

`main()` から `fun _ B()` への呼び出しで始まる、オリジナルの制御フロー (つまり逆トレース) 順で最後の 3 つのブロックの逆アセンブリーを確認します。

```
/m interleaves source info
(gdb) btrace /m 1-3
src/stack.c:32  static long fun_B(void* arg) {
 0x000000000040050e <fun_B+1>:        mov    %rsp,%rbp
 0x0000000000400511 <fun_B+4>:        mov    %rdi,-0x18(%rbp)
src/stack.c:33      struct B_arg* myarg = arg;
 0x0000000000400515 <fun_B+8>:        mov    -0x18(%rbp),%rax
 0x0000000000400519 <fun_B+12>:       mov    %rax,-0x8(%rbp)
src/stack.c:34      if (!myarg) return -1;
 0x000000000040051d <fun_B+16>:       cmpq   $0x0,-0x8(%rbp)
 0x0000000000400522 <fun_B+21>:       jne    0x40052d <fun_B+32>
src/stack.c:36      return myarg->arg1 + myarg->arg2;
 0x000000000040052d <fun_B+32>:       mov    -0x8(%rbp),%rax
 0x0000000000400531 <fun_B+36>:       mov    (%rax),%rdx
 0x0000000000400534 <fun_B+39>:       mov    -0x8(%rbp),%rax
 0x0000000000400538 <fun_B+43>:       mov    0x8(%rax),%rax
 0x000000000040053c <fun_B+47>:       lea    (%rdx,%rax,1),%rax
src/stack.c:37  }
 0x0000000000400540 <fun_B+51>:       leaveq
 0x0000000000400541 <fun_B+52>:       retq
0x000000000000002a:Cannot access memory at address 0x2a

fun_B() is executed and returns to an invalid address suggesting a corrupted
stack. fun_B() leaves, but there was no corresponding push on entry to fun_B()
=> the function pointer comp that was called in main() was corrupted.
```


インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

インテル® System Studio のアナライザー

前述のデバッグツールは、組込みプラットフォームの信頼性と完成度を保証する 1 つのアプローチをもたらすものです。インテル® Inspector for Systems によるコード解析は、メモリー問題と平行性問題を特定する別の手法を提供します。インテル® VTune™ Amplifier for Systems は、組込みプラットフォームの信頼性だけでなく、応答性、パフォーマンス、消費電力管理にも焦点を当てています。どちらのツールも、組込みクロス開発パラダイムを積極的にサポートし、ターゲットリソースの制限を考慮したインテル® System Studio の手法に対応するように設計されています。

インテル® Inspector for Systems	
重大なメモリー / スレッド化のコーディング・エラーを正確に特定	
インテル® Inspector for Systems のメモリー / スレッド化アナライザー	メモリーリーク、無効なアクセス、データ競合とデッドロックを素早く発見
	高速で効率良いスタティック/ヒープ拡張解析を実行して致命的な問題を発見
	リモートデータ収集、デバッガー・ブレークポイント、選択したエラーでのブレークをサポート

インテル® Inspector for Systems の `system_studio_target.tgz` パッケージには、ユーザー・インターフェイスとメモリー制限のあるプラットフォームで使用可能なコマンドライン・バージョンが含まれています。データ解析は、ターゲット・プラットフォームのコマンドラインから実行することも、サンプリング・データを開発ホストにコピーして、インテル® Inspector のグラフィカル・ユーザー・インターフェイス (GUI) から行うこともできます。

組込みターゲットデバイスまたは仮想化プラットフォームでレベル 2 メモリーチェック解析を行うには、コマンドラインから次のコマンドでインテル® Inspector を実行します。

```
$ source /home/root/inspector_for_systems/inspxe-vars.sh
$ inspxe-cl --no-auto-finalize --collect mi2 ../<binary-name>
```

サンプリング結果はディレクトリー `../r000mi2` に作成されます。メモリーリークまたはスタック・オーバーフローの場所に関するソースレベルの情報を得るには、シンボル情報を有効にして (`-g`) 最適化コードをコンパイルする必要があります。

Linux* ホスト上のインテル® Inspector 2013 for Systems (図 7) で結果を表示するには、次の操作を行います。

a. ターゲットデバイスから開発ホストに結果ディレクトリーをコピーします。

b. `source /opt/intel/system_studio_2013.0.xxx/inspector_2013_for_systems/inspxe-vars.sh` コマンドを実行します。

c. 次のコマンドを実行します。

```
$ inspxe-gui ~/<results-location>/test/r000mi2
```

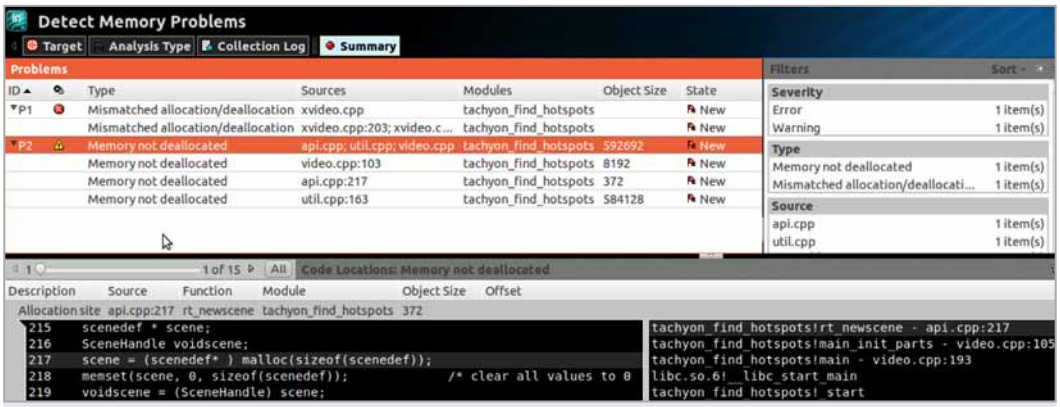


図 7. インテル® Inspector for Systems でメモリー割り当て問題を識別

インテル® VTune™ Amplifier for Systems

インテル® VTune™ Amplifier for Systems は、組込みデバイスで実行しているソフトウェア・コンポーネントの電力使用状況の解析とパフォーマンスの解析を行います。開発者は、最適化すべき hotspot と、アーキテクチャー上のリソース制限によるキャッシュミスや過度のメモリーアクセスを識別することができます。また、頻繁な I/O デバイスへのアクセスなどのアクティビティーによりウェイクアップ・イベントやプロセッサー周波数の変化を引き起こす、システムあるいはアプリケーション・コードの関数とコード行を特定します。`system_studio_target.tgz` パッケージには、データ収集用の 2 つのターゲット・コンポーネントが含まれています。「アーキテクチャー・イベントのみ」サンプリングには、インテル® VTune™ Amplifier の SEP (Sampling Enabling Product) を利用します。リモートデータ収集には、インテル® VTune™ Amplifier のコマンドライン・データ・コレクター amplxe-runss.py を利用します。図 8 は、SSH プロトコルを用いた amplxe-runss リモートデータ収集モデルの基本的なアーキテクチャーを示しています。

消費電力の最適化と効率化	
インテル® VTune™ Amplifier	CPU/GPU アクティビティーのイベントと解析に関する SoC 全体の詳細情報を提供
	ウェイクアップの原因、アプリケーションによってトリガーされたタイマー、ハードウェア割り込みレベルにマップされた割り込みを識別
	ソース・コード・レベルでプロセッサーをウェイクアップする CPU コア周波数とイベントを表示
パフォーマンスの最適化	
消費電力 / パフォーマンス・プロファイリング・ツール	システムイベントと SoC イベントを解析
	コールスタックとともにハードウェア・イベントを表示、低オーバーヘッドのサンプリングを提供、小さな関数の hotspot を検出
	インライン展開および並列化を決定する上で役立つデータとして統計コールカウントを提供
	ソースまたはアセンブリーで結果を表示 (インストルメンテーションの必要なし)

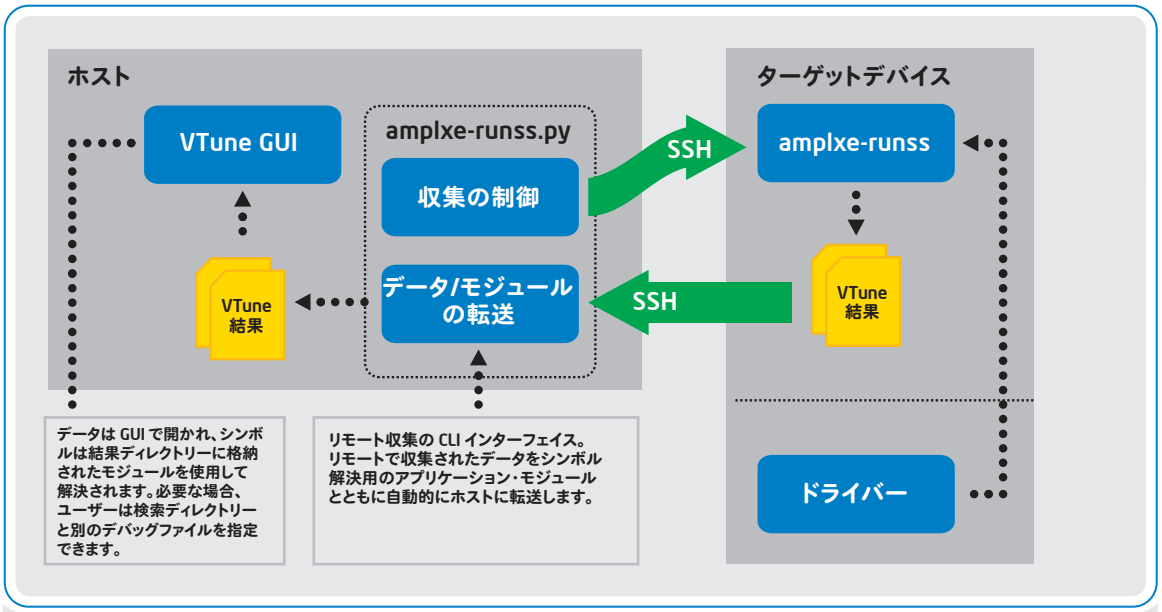


図 8. インテル® VTune™ Amplifier for Systems におけるホストとターゲットの関係

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

このリモートデータ収集モデルによりパフォーマンス・データを収集するには、インストールが完了した後、次の手順を行います。

パスワードなしでログインできるようにターゲットをセットアップします。

1. ルート・ディレクトリーに .ssh ディレクトリーが含まれており、読み取り/書き込み/実行権限が所有者のみに設定されていることを確認します。
2. `cat ~/.ssh/id_dsa.pub | ssh root@ip_target " cat >> /home/root/.ssh/authorized_keys"`
3. ターゲットで `chmod 600 /home/root/.ssh/authorized_keys` コマンドを実行します。
4. パスワードなしでログインできることを確認します。
   a. `ssh root@target_ip`

アプリケーションでインテル® VTune™ Amplifier 2013 for Systems のリモートコレクターを実行します。

1. ターゲットの環境変数を設定します。
   a. `export AMPLXE_TARGET_PRODUCT_DIR=/home/root/linux32`
      (リモート・コレクション・バイナリーをコピーした場所から)
   b. `export AMPLXE_TARGET_TMP_DIR=/tmp`
2. コレクションを実行します。
3. `amplxe-runss.py --target=user@target -r result@@@ --duration 10 --no-pmu-stack --event-config CPU_CLK_UNHALTED.CORE:,CPU_CLK_UNHALTED.REF`

ウェイクアップ・イベントと周波数変化データを収集するには、このコマンドを代わりに実行します。

```
amplxe-runss.py --target=root@ip_target  --pwr-config=sleep,frequency --result-dir r@@@ -- /tmp/hello
```

このコマンドは、ターゲットでリモート電力収集を行い、ターゲット・アプリケーション /tmp/hello を実行します。また、ホストで開くことができる結果ディレクトリー r000 を作成します。

Linux* ホストでインテル® VTune™ Amplifier 2013 for Systems の結果 (図 9) を表示するには、次のコマンドを実行します。

(`amplxe-gui  r000`)

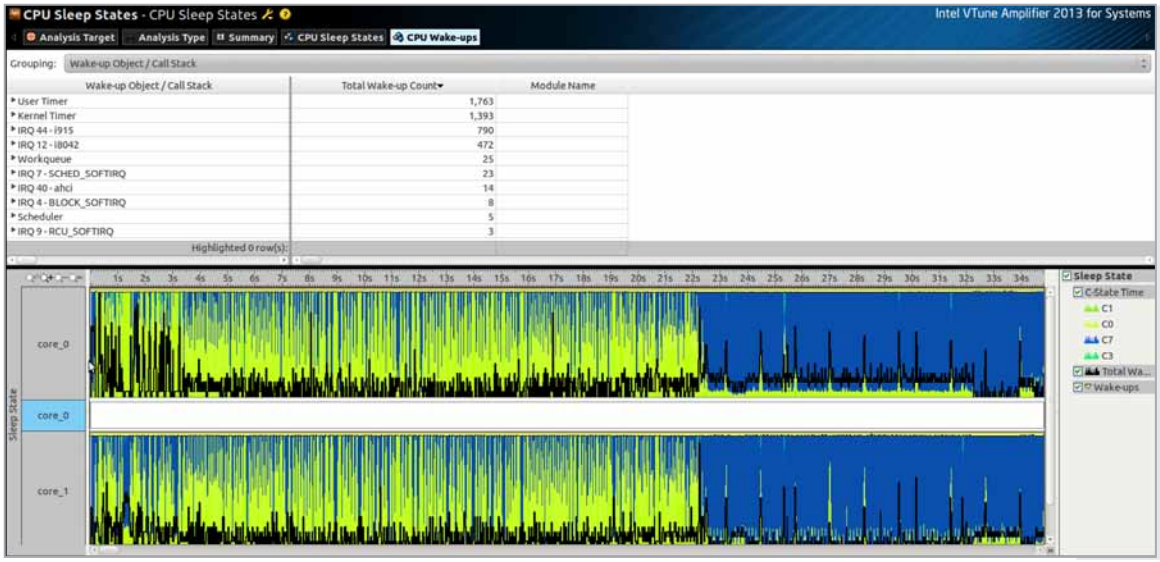


図 9. インテル® VTune™ Amplifier for Systems の電力状態解析

インテル® System Studio の コンパイラーとライブラリー

インテル® C++ コンパイラーは、Wind River* Linux*、Yocto Project*、その他のカスタム GNU* クロスビルド・ツール向けに使用される Poky-Linux* や OpenEmbedded* 互換の GNU* クロスツールとのクロスコンパイルおよび統合をサポートしています。インテル® System Studio には事前定義済みのコンパイラー環境ファイルが用意されており、-platform=yl13 オプション (Yocto Project* 1.3 がターゲットの場合)、または -platform=wrl50 オプション (Wind River Linux* 5.0 のアプリケーション開発ツールキットがターゲットの場合) を指定するだけでインテル® C++ コンパイラーをクロス開発に使用できます。

また、Eclipse* IDE への統合を自動化する eclipse_integration.sh スクリプトも用意されています。そのため、業界で広く使用されている IDE から、インテル® C++ コンパイラーやその他のコンポーネントの主な機能を簡単に利用できます。統合スクリプトを実行すると、新しく [Intel Tools] メニューが追加され、ガイド付き自動並列化、インテル® C++ コンパイラーの選択、インテル® C++ コンパイラーの環境ファイルエディターを利用できます。この環境ファイルエディター (図 10) はインテル® System Studio の機能の 1 つで、インテル® C++ コンパイラーの GNU* クロスビルド・ツールへの統合のカスタマイズを、コマンドライン・エディターだけでなく、IDE 内から行えます。

インテル® プロセッサーのパフォーマンスとスケーラビリティーを引き出す業界標準のハイパフォーマンス C++ コンパイラーおよびライブラリー	
インテル® C++ コンパイラー 13.0 とライブラリー	優れた速度最適化によって高速なコードを生成します。
	最短実行時間の低消費電力アプリケーションの開発が可能です。
	GNU* クロスビルド、Eclipse* CDT への統合、Yocto Project* アプリケーション開発ツールキットをサポートします。
	インテル® Cilk™ Plus の簡単な 3 つのキーワードによりマルチコアのパフォーマンスを向上させます。
	インテル® マス・カーネル・ライブラリー (インテル® MKL) は、インテル® Core™ プロセッサーおよびインテル® Xeon® プロセッサー向けに高度に最適化/スレッド化された数学ルーチンを提供します。
	インテル® インテグレーテッド・パフォーマンス・プリミティブ (インテル® IPP) は、要件の厳しい信号、データ、マルチメディア・アプリケーション向けに高度に最適化されたソフトウェア・ビルディング・ブロックの広範囲なライブラリーです。

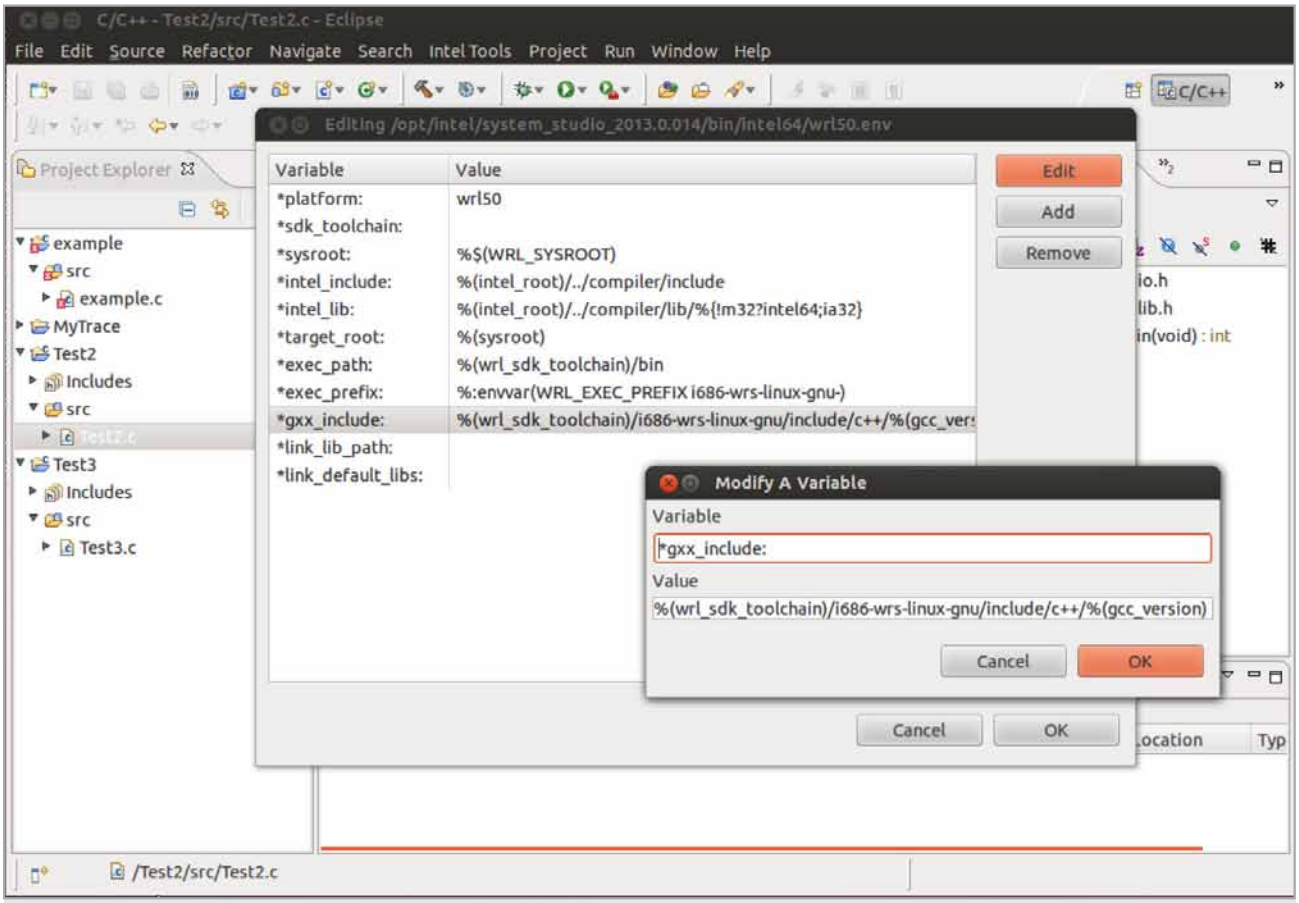


図 10. コンパイラーの Eclipse* IDE への統合と環境ファイルエディター

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

インテル® Cilk™ Plus は、制限されたワーカーのセット (スレッドのプール) に任意の数のタスクをマップするランタイム動的スケジューラーを利用して、C および C++ でフォワード・スケーリングなタスク並列処理を行います。呼び出しチェーン (入れ子の並列処理) の管理やリソースのオーバーサブスクライブなしにマルチコア並列処理を実装できる、構成可能な設計を考慮しています。インテル® Cilk™ Plus は、スケジュールされたタスクの数に従ってリソースを使う代わりに、プールのワーカーの数に従って並列処理することを保証します。この保証は、長期間スケーラブルな並列処理の実装とリソースを意識した組込み設計を行う際に特に効果的です (図 11)。

インテル® System Studio に含まれるインテル® MKL およびインテル® IPP は、インテリジェント・システムで高度に最適化されたデータ処理と信号処理を行うアプローチを提供します。

インテル® IPP は、広範なデータタイプをカバーするさまざまな分野の関数が含まれる C ライブラリーで、簡単に利用できます。キャッシュブロックのようなインコアの最適化に焦点を当てており、インテル® ストリーミング SIMD 拡張 (インテル® SSE) などの新しい命令セットの拡張を継続的に取り入れています。インテル® アドバンスト・ベクトル・エクステンション 2 (インテル® AVX 2) も今後サポートされる予定です。

```
for (size_t i = 0; i <= nsteps && (0 < stage[1].n || 0 == i); ++i) {
  cilk_spawn read_signal    (size,       x + stage[0].i, y + stage[0].i, stage[0].n);
  cilk_spawn process_signal(stage[1].n, x + stage[1].i, y + stage[1].i,
                                        x + stage[2].i, y + stage[2].i);
             print_signal   (stage[3].n, x + stage[3].i, y + stage[3].i, std::cout);
  cilk_sync;

  stage[2].n = stage[1].n;
  std::rotate(stage, stage + 4 - 1, stage + 4); // quad-buffering
}
```

図 11. インテル® Cilk™ Plus 構文の例。

すべてのインテル® 組込みプロセッサー向けに最適化された関数	
ライブラリー	インテル® MKL は、インテル® Core™ プロセッサーおよびインテル® Xeon® プロセッサー向けに高度に最適化/スレッド化された数学ルーチンを提供します。
	インテル® IPP は、最も要求の厳しい信号、データ、マルチメディア・アプリケーション向けに高度に最適化されたソフトウェア・ビルディング・ブロックの広範囲なライブラリーです。

信号処理 (1d)	画像処理 (2d)	カラー変換
> 変換 (ウェーブレットなど) > 畳み込み/相関 > フィルタリング (IIR、FIR など) > 統計	> 変換 (回転など) > (非) 線形フィルター (ノイズなど) > FFT、DFT、DCT > 統計	> カラー空間変換 > パターン (Bayer など) > 明るさ/コントラスト > リサンプリング
ベクトル/行列	**完全性圧縮/暗号化**	**その他のドメイン**
> 論理、シフト、変換 > 三角関数 > 分解、固有値 > 転置	> エラー訂正、リードソロモン > 圧縮 (エントロピー、ディクテーション) > MD5、T(DES)、RSA、DSA > 乱数ジェネレーター	> ビデオ、ピクチャー・コーディング > オーディオ (音声コーディングなど) > ストリング処理 > ユーティリティー

図 12. インテル® IPP でカバーされているさまざまなドメイン

インテル® IPP は、より小さなデータセット、レイテンシーに制約のあるアプリケーションの効率を高めるため、またアプリケーション・レベルのスレッド化をより適切に制御するため、シーケンシャル・ライブラリー[3] (ライブラリー名ポストフィックスは “linear” の “_l”) として提供されます。インテル® IPP はすべてスレッドセーフであり、マルチスレッド化をサポートします。インテル® IPP のメイン・サンプル・コレクション (別途ダウンロード) には、サンプルドライバーを含む「高度な使用法(advanced usage)」のサンプルと、このドライバーのサービスを要求するアプリケーションが含まれています。これらのサンプルは、カーネルモードで実行するコードを開発する上で役立ちます。インテル® IPP は、カーネルモード (ring 0) で実行するコードをサポートするため、非位置独立 (“nonpic”) のライブラリーを提供します。

インテル® IPP の関数名は、ライブラリー・ドメインを示すプリフィックス (例えば、信号処理は “ipps”) と、含まれるデータタイプを示すポストフィックス (例えば、単精度浮動小数点の場合は “f32”) からなる命名規則に従っています。C 言語の呼び出し規則とリンク (非修飾の関数名) では、ほぼすべてのプログラミング言語とコンパイラーからインテル® IPP 関数の呼び出しが可能です。

浮動小数点データ型については、インテル® IPP 関数のセットがインテル® MKL に含まれており、インテル® Xeon® プロセッサーおよびインテル® Core™ プロセッサーの能力をフルに引き出します。インテル® MKL は、シングルスレッドまたはマルチスレッド実装として利用可能です。

インテル® MKL には、高度に最適化およびスレッド化された関数が含まれており、64 ビット整数インデックス空間までの大量のワークロードがサポートされています (“ilp64”)。インテル® MKL はスループットの高さに焦点を当てていますが、アプリケーション・レベルのスレッド化はインテル® IPP に似たシーケンシャル実装によって行われます。複数のスレッドによるマルチスレッド・バージョンのインテル® MKL 関数の呼び出しはスレッドセーフです。

```
int main(int argc, char* argv[])
{
  ippInit();

  // インテル® IPP の使用を開始
}
```

図 13.最適なコードパスを選択するため、インテル® IPP のほかの関数を使用する前に ippInit を呼び出します。ダイナミック・リンク・ライブラリー (*.so、*.dll) は、ライブラリーの「ロードイベント」をキャプチャーして初期化を行います。一方、スタティック・リンクは、スタティック・ライブラリー (*.a, *.lib) から (呼び出しに従って) コードを取り出します。ただし、柔軟性のため、使用するリンクの種類 (動的または静的) に関係なく ippInit を呼び出します。

```
void process_signal_ipp(size_t size, const float xin[], const float yin[],
  float xout[], float yout[])
{
  std::copy(xin, xin + size, xout);
  ippsAutoCorr_32f(yin, static_cast<int>(size), yout, static_cast<int>(size));
}

void process_signal_ipp(size_t size, const double xin[], const double yin[],
  double xout[], double yout[])
{
  std::copy(xin, xin + size, xout);
  ippsAutoCorr_64f(yin, static_cast<int>(size), yout, static_cast<int>(size));
}
```

図 14. インテル® IPP は、さまざまなプログラミング言語で利用可能な C 関数ライブラリーです。上記の C++ コードは、型に依存しない手法で自己相関を実行するため、process_signal_ipp をオーバーロードしています。

まとめ

インテル® System Studio は、ハードウェアとソフトウェアの深い洞察を提供する新しい統合ソフトウェア開発スイートです。Wind River Linux* や Yocto Project* など、組込み Linux* を搭載するインテル® プロセッサー・ベースの組込みデバイスおよびインテリジェント・システムの開発、テスト、最適化を促進します。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

線形代数	高速フーリエ変換 (FFT)	ベクトル演算
> BLAS、スパース BLAS > LAPACK ソルバー > スパースソルバー (DSS、PARADISO) > 反復法ソルバー (RCI) > ScaLAPACK、PBLAS	> 多次元 > FFTW インターフェイス > クラスター FFT > 三角変換 > ポアソンソルバー > VSL による畳み込み	> 三角 > 双曲線 > 指数、対数 > 累乗 / 平方根
乱数ジェネレーター	**サマリー統計**	**データ適合**
> 線形合同法 > Wichmann-Hill > Mersenne Twister > Sobol > Neiderreiter > 非決定的	> 尖度 > 変化係数 > クォンタイル > 順序統計量 > 最小 / 最大 > 分散-共分散	> スプラインベース > 補間 > セル探索

図 15. 線形代数はインテル® MKL の主要ドメインの 1 つです。インテル® MKL の FFT は記述子をサポートしています (そのため、フォワード / バックワード関数のインターフェイスはインテル® IPP よりも単純です)。

```
void process_signal_mkl(size_t size, const float xin[], const float yin[],
  float xout[], float yout[])
{
  static MKL_INT mkl_size = static_cast<MKL_INT>(size);
  static VSLCorrTaskPtr task = 0;

if (mkl_size != size || 0 == task) {
    vslCorrDeleteTask(&task);
    mkl_size = static_cast<MKL_INT>(size);
    vslsCorrNewTask1D(&task, VSL_CORR_MODE_AUTO, mkl_size, mkl_size, mkl_size);
  }

  std::copy(xin, xin + size, xout);
  vslsCorrExec1D(task, yin, 1, yin, 1, yout, 1);
}
```

図 16. 相関 (および畳み込み) のインターフェイスはインテル® IPP のインターフェイスとは異なります。「フラットなオブジェクト指向インターフェイス」により、ハンドル (「相関タスク」) を作成したり編集できます。

関連資料 (英語)

製品の購入および更新: http://software.intel.com/en-us/articles/buy-or-renew/

評価ガイド: http://software.intel.com/en-us/articles/evaluation-guides/

サポート: すべての製品アップデートおよびインテル® プレミアサポート・サービスを 1 年間ご利用いただけます。インテル® プレミアサポートは、セキュアな Web ベースで技術者からのサポートを受けられます。このツールの詳細: http://software.intel.com/en-us/articles/intel-system-studio-articles

リリースノート: http://software.intel.com/en-us/articles/intel-system-studio-release-notes

フォーラム: インテルが運営するフォーラムでインテル® System Studio のほかのユーザーと意見を交換できます: http://software.intel.com/en-us/forums/intel-system-studio

ラーニングラボ: http://software.intel.com/en-us/articles/intel-learning-lab

コンテンツ・ライブラリー: さまざまな技術情報にアクセスできます。

1. インテル® Atom™ プロセッサー E660 1.6GHz およびインテル® Core™ i7 でインストルメント例を使用して測定
2. リモート・システム・エクスプローラー (RSE) は Eclipse* ターゲット管理プロジェクトの一部です。組込みクロス開発での使用方法の詳細は、eclipse.org のチュートリアルを参照してください: http://www.eclipse.org/tm/tutorial/index.php
3. 内部的にマルチスレッドを提供するダイナミック・リンク・ライブラリーも利用可能です。

ショック

CloverLeaf と インテル® Xeon Phi™ コプロセッサーの巡り合い

Victor Gamayunov
Stephen Blair-Chappell

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

ウェーブ!

インテル社の Victor Gamayunov と Stephen Blair-Chappell は最近、CloverLeaf ベンチマークの作者と協力して、ベンチマークを「インテル® Xeon Phi™ コプロセッサー対応」にする取り組みを行いました。

この記事では、Victor と Stephen が行った作業とその結果について説明します。

CloverLeaf ベンチマーク

CloverLeaf[1] はスタッガード・デカルト格子上で 2 次元の圧縮性オイラー方程式を解く小さなベンチマークです。コードは、Fortran フレームワークと、すべての実作業を行う多くの Fortran/ANSI-C 低水準ビルディング・ブロックから構成されます。

CloverLeaf で用いられる手法は、流星が地球の大気に与える影響をシミュレートする手法に似たもので、ショックウェーブ (衝撃波) の後方にある物体のフローと、ショックウェーブがどの程度遠くに移動するかを予測します。

240 以上のスレッドを処理するためにプログラミングで苦労した点

プログラミングで特に苦労したことはありませんでした。CloverLeaf では、シミュレーション空間は複数の領域に分割され、MPI により複数のコア上で実行されます。図 2 は、16 の領域に分割されたシミュレーション・メッシュです。インテル® Xeon Phi™ コプロセッサーでは、領域の数を 60 に増やし、各タスクで 4 つの OpenMP* スレッドを利用して、60 の MPI タスクを実行しました。

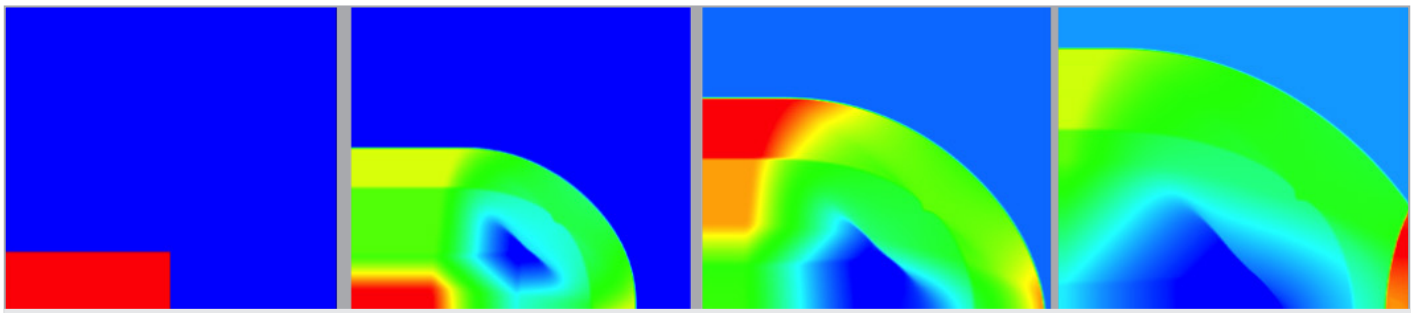


図 1. CloverLeaf シミュレーションのスナップショット

図 1 は、CloverLeaf シミュレーションの 4 つのスナップショットを示しています。この結果は、米国エネルギー省 (U.S. Department of Energy) のグラフィカル解析ツール VisIt で視覚化されました。[2] シミュレーションの最初に、高エネルギー高密度ガスの領域が低エネルギー低密度ガスの領域に取り込まれます。

シミュレーションの進行とともに、ショックウェーブがどのように移動するかが分かります。最後のショットは、境界 (正方形の箱の壁) から反射し、より複雑なパターンを形成したショックウェーブを示しています。

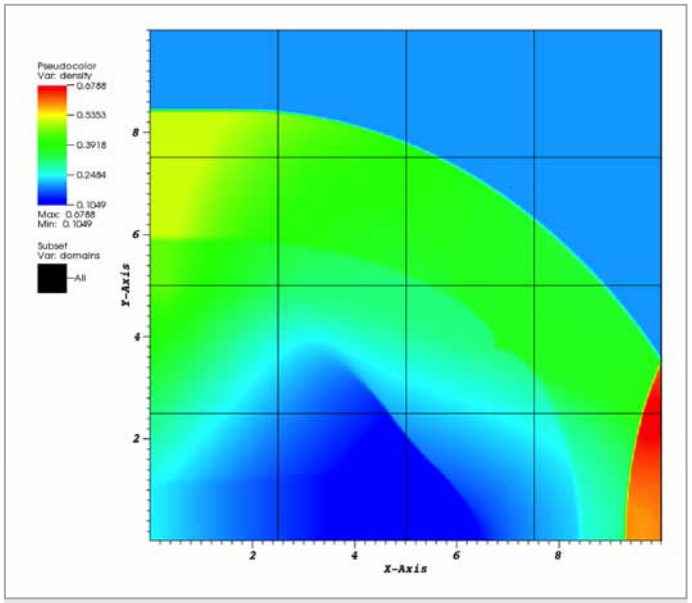


図 2. 区分化されたデータセット

「インテル® Xeon Phi™ コプロセッサー対応」にするために行った作業

当初、インテル® メニー・インテグレーテッド・コア (インテル® MIC) アーキテクチャー・ハードウェアがなかったため、すべての開発を通常の 8 コアのワークステーション PC 上で行いました。インテル® Xeon Phi™ コプロセッサーの能力を引き出すには、コードを適切に並列化およびベクトル化する必要があることは分かっていました。そのためには、CloverLeaf ですべてのコプロセッサー・コアと 512 ビット幅の SIMD レジスターを利用しなければなりません。

ベンチマークは MPI によりすでに並列化されていたため、コードのベクトル化に当初の労力の大部分を費やしました。我々のアプローチは、最初にコードの hotspot を特定し、これらの hotspot がコンパイラーによって自動ベクトル化されたものであることを確かめることでした。

インテル® Xeon® プロセッサーでは、自動ベクトル化によりベンチマークのパフォーマンスが 20% 向上しました。インテル® Xeon Phi™ コプロセッサーでは 80% も向上しました。

その後、OpenMP* を MPI コードに追加しました。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

プロジェクトで特に良かった点

インテル® Xeon Phi™ コプロセッサーで CloverLeaf の最初のバージョンを実行することが、予想よりもはるかに簡単だったので驚きました。`-mmic` コンパイラー・オプションで既存のコードを再コンパイルするだけで、実行ファイルをコプロセッサー上でネイティブに実行することができました。

最も苦労した点と解決方法

プロジェクトの開始時点では、コードは適切にベクトル化されるかどうかを考慮することなく記述されていました。インテル® コンパイラーのレポート機能を利用することで、一部のコードが自動的に並列化されなかった理由を知ることができました。

1 行のコード中に複数のベクトル化の可能性が存在するため、コンパイラーのメッセージを個別に見ていくだけでは、コードがベクトル化されかどうかを確認するのに不十分なこともありました。この場合は、アセンブリー・コードを見てパックド命令が存在するかどうかを調べました。

インテル® Xeon Phi™ コプロセッサー向けにプログラムを移植する開発者へのアドバイス

1. **インテル® Xeon Phi™ コプロセッサー・ベースのシステムが利用できるまで待つのではなく、すぐに最適化作業を開始する。**
   従来の PC やワークステーション上で行った最適化作業は、インテル® Xeon Phi™ コプロセッサー上で実行するコードでもほぼ同じ効果が得られます。実際に、我々が行った最適化作業はインテル以外のデバイスでコードを実行した場合でも効果がありました。
2. **コードが適切に並列化およびベクトル化されていることを確認する。**
   一旦コードが適切に並列化およびベクトル化されれば、インテル® Xeon Phi™ コプロセッサー上で効率良く動作することが分かりました。
3. **プログラムの並列性を測定する手法を見つける。**
   我々はインテル® VTune™ Amplifier を使ってコードの並列性を確認した後、コンパイラーのベクトル化レポートとインテル® VTune™ Amplifier を組み合わせてコードが適切にベクトル化されているかどうかを調べました。
4. **並列化に適したアルゴリズムにリファクタリングする。**
   CloveLeaf ではループの依存性をなくすことが特に重要でした。
5. **ベクトル化およびキャッシュに適したデータアクセスに再構成する。**
   連続しないメモリー位置からの読み取り/書き込みを最小限にして、連続するアドレスに書き込むようにコードを修正すると、パフォーマンスが大幅に向上することが分かりました。
6. **可能であれば、同期を回避する。**
   コードがシリアル化されるため、ループ内部で同期プリミティブの使用を避けます。変数のスコープを縮小することを考えましょう。可能な場合は、共有変数を使用しないでください。
7. **動的メモリー割り当てを回避する。**
   スレッドコードで従来のメモリー割り当て関数やその他のシステム・ライブラリーの使用を避けます。これらの関数の多くはシリアル化の原因となるロックを使います。CloverLeaf では、細粒度の動的メモリー割り当ての代わりに、プログラムの最初に 1 回だけメモリー割り当てを行うように変更しました。

最終的な結果

図 3 は、シングルソケット、8 コア・プロセッサーとインテル® Xeon Phi™ コプロセッサーで CloverLeaf を実行した結果です。メッシュサイズが小さい場合、シングル・プロセッサーの 1/3 程度のパフォーマンスしか得られません。**メッシュサイズが大きくなると、シングル・プロセッサーの 3 倍以上のパフォーマンスが得られます。**

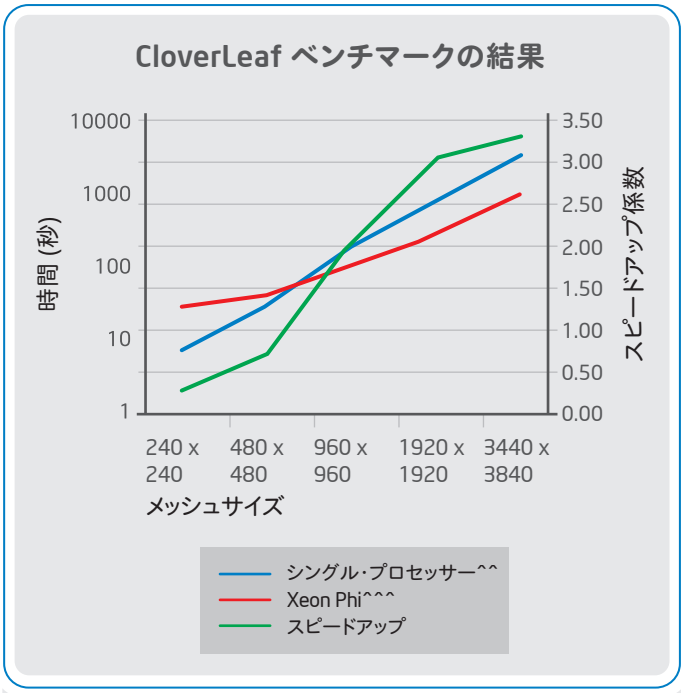


図 3. CloverLeaf の結果

参考資料 (英語)

1.http://warwick-pcav.github.com/CloverLeaf/
2.https://wci.llnl.gov/codes/visit/home.html

^^シングルソケット 8 コア インテル® Xeon® E5-2687W プロセッサー 3.1GHz、32GB DDR3 (1333Mhz) メモリー、ターボ・ブースト・テクノロジーおよびハイパースレッディング・テクノロジー有効。
^^^ インテル® Xeon Phi™ コプロセッサー 61 コア、1.09GHz、8GB GDDR5 (5.5 GT/s) メモリー。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、**http://software.intel.com/en-us/articles/optimization-notice/#opt-jp** を参照してください

インテル® Xeon Phi™ コプロセッサーとインテル® MPI ライブラリーの プログラミング・モデル

Michael Chuvelev、
Dmitry Durnov、
Sergey Kazakov、
Alexander Supalov、
Andrew Tananakin

インテル® Xeon Phi™ コプロセッサー[1] は、インテル® Xeon® プロセッサーで実行される、オフロードされて高度に並列化されたコードの処理に使用することができます。しかし、インテル® Xeon Phi™ コプロセッサーは独立した計算ノードと考えることもできます。つまり、メッセージ・パッシング・インターフェイス (MPI)[2] による分散メモリー型プログラミング・モデルも自然なアプローチと言えるでしょう。

インテル® MPI ライブラリー Linux* 版[3] は、バージョン 4.1 からインテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサーの両プラットフォームをサポートしています。その結果、既存のハイパフォーマンス・コンピューティング (HPC) アプリケーションをインテル® Xeon Phi™ コプロセッサー搭載プラットフォームに簡単に移植できるようになりました。

この記事では、移植プロセスを促進する、インテル® MPI ライブラリーと単純なユーザー・インターフェイスを組み合わせた新しいプログラミング・モデルについて説明します。(特定の実用アプリケーションの移植については、この後の記事をご覧ください。)

MPI プラットフォームとしてのインテル® MIC アーキテクチャー

インテル® メニー・インテグレーテッド・コア (インテル® MIC) アーキテクチャーに基づき、インテル® Xeon® プロセッサー・ファミリーを補足するために開発されたインテル® Xeon Phi™ コプロセッサーは、高度に並列化された HPC アプリケーションのパフォーマンスを劇的に向上させることができます。インテル® Xeon Phi™ コプロセッサーには、512 ビット幅のベクトルユニットを備えた計算コアが最大 61 個搭載されています。すべてのコアに完全にコヒーレントな 512KB L2 キャッシュが付属し、4 スレッドをサポートします。動作クロックは最大 1.05GHz で、最大 8GB (GDDR5) のメモリーを共有します。インテル® メニーコア・プラットフォーム・ソフトウェア・スタック (MPSS)、ユーザー・インターフェイス、ファイル・システム・ストレージ用のメモリーとしても利用されます。

ソフトウェア・スタックには組込み Linux* カーネルが含まれており、以下を含むフルセットのサービスを提供します。

- IP アドレス設定
- イーサネット・ブリッジ
- セキュアシェル (SSH) アクセス
- CCL-direct (コプロセッサー通信レイヤー) による InfiniBand* ホスト・チャネル・アダプター (HCA) アクセス
- ネットワーク・ファイル・システム (NFS)

これらの機能により、インテル® Xeon Phi™ コプロセッサーを独立したノードとして扱うことができる、高水準の標準インターフェイスが提供されます。このため、MPI は、インテル® Xeon Phi™ コプロセッサー向けのプログラミングにおいて、このアーキテクチャーでサポートされるプログラミング・モデルの幅を拡張した、非常に自然なアプローチをとることができます。

プログラミング・モデル

インテル® MIC では、オフロードと MPI モデルの 2 つの高水準プログラミング・モデルを利用できます (図 1 を参照)。

オフロードモードでは、インテル® Xeon Phi™ コプロセッサーまたはインテル® Xeon® プロセッサーのいずれかをアクセラレーターとして扱います。両方のオフロードモデルで共通なのは、インテル® MIC アーキテクチャー向けインテル® C コンパイラー、インテル® C++ コンパイラー、インテル® Fortran コンパイラーおよびインテル® マス・カーネル・ライブラリー (インテル® MKL) のオフロード機能を使用する点です。このモードでは、アプリケーションのオフロード機能について MPI ライブラリーが「何も知らない」ため、コードのオフロード部分の内部で MPI 呼び出しを行うことはできません。

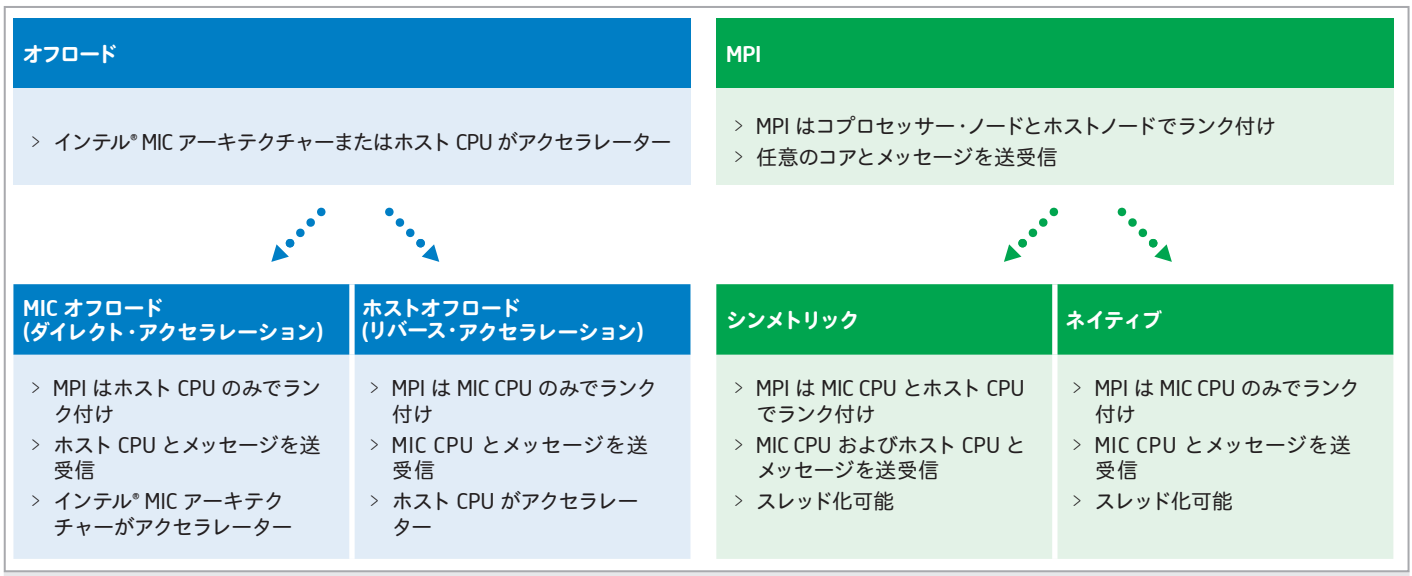


図 1. プログラミング・モデル

ダイレクト・アクセラレーションは、インテル® MPI の以前のバージョンやほかの MPI 実装ですでにサポートされているモデルです。リバース・アクセラレーションは、インテル® MPI バージョン 4.1 でサポートされたモデルで、前述の関連製品のオフロード機能に依存します。MPI モードでは、ホスト CPU (通常はインテル® Xeon® プロセッサー) およびインテル® Xeon Phi™ コプロセッサーをピアノードとして扱います。MPI プロセスは、ホスト CPU とインテル® Xeon Phi™ コプロセッサーの両方またはいずれかに存在します。主に次の 2 つの使用状況があります。

- シンメトリック・モデル: MPI プロセスはホストとインテル® Xeon Phi™ コプロセッサーの両方に存在します。ヘテロジニアス・クラスターの最も一般的な MPI ビューです。
- ネイティブモデル: すべての MPI プロセスはインテル® Xeon Phi™ コプロセッサーにのみ存在します。シンメトリック・モデルの特別なケースです。このモデルでは、ホスト CPU をオフロードに使用できるため、リバース・アクセラレーションと似ています。

ユーザー・インターフェイス

インテル® MPI は、インテル® Xeon Phi™ コプロセッサー向けに、Hydra プロセス・マネージャー (PM) `mpiexec.hydra` と、Hydra でビルドされたスクリプト `mpirun` を提供します。この PM およびスクリプトはインテル® Xeon® プロセッサー向けにも同様に提供されているため、前述のすべてのプログラミング・モデルで共通です。

各ノードと各コプロセッサーは、一意のシンボリック名または IP アドレスによって識別されます。インテル® MIC の構築には、インテル® MIC をサポートするインテル® MPI が必須です。ジョブ管理、リソース制御、クラスター監視などのサードパーティー製品で提供される関連サービスにもインテル® MIC のサポートが必要になります。

ホストおよびコプロセッサーは、次のように、長い (コロンなど) 表記で、`mpiexec.hydra` のコマンドラインにシンボリック名または IP アドレスによって指定できます。

```
(host)$ mpirun -n 4 -host myhost ./
hello :-n 4 -host mic0 ./hello.mic
```

より便利な短い表記にするには、シンボリック名または IP アドレスでノードとコプロセッサーを指定した hostfile または machinefile を作成します。ここでは、インテル® MPI で提供されるテストプログラム hello.c のアプリケーションでこの手法を使用します。

オフロードモデルのビルドと起動プロセスはインテル® Xeon® プロセッサー・クラスターの場合と同じです。

1. インテル® Composer XE およびインテル® MPI 環境を設定します。

```
(host)$ source <composer_installdir>/bin/
compilervars.sh intel64
(host)$ <installdir>/intel64/bin/mpivars.sh
```

2. アプリケーションをコンパイルします。

```
(host)$ mpiicc -o hello hello.c
```

3. 実行ファイルを起動します (例えば、hostfile でノードを定義します)。

```
(host)$ mpirun -n 64 -hostfile myhosts ./hello
```

myhosts にはノードの名前をリストします。

```
host1
host2
```

ネイティブモードでアプリケーションをビルドするには、ホストのインテル® Composer XE およびインテル® MPI 環境を上記のように設定します。次に、インテル® コンパイラーの呼び出しコマンドに `-mmic` オプションを追加して、インテル® MIC アーキテクチャー向けのバイナリー実行ファイルをビルドします。

```
(host)$ mpiicc -mmic -o hello.mic hello.c
```

コプロセッサーのアプリケーションを実行するには、環境変数 `I_MPI_MIC` を 1 に設定し、コプロセッサーの名前または IP アドレスを `hostfile` に指定します。

```
(host)$ mpirun -n 64 -hostfile <hostfile>
./hello.mic
```

`hostfile` で指定されたコプロセッサーは、クラスターノードで 1 つまたは複数のホスト CPU にアタッチすることができます。MPI は、ホストのみの通常の MPI アプリケーション実行と同じ規則に沿って、コプロセッサーの分配をランク付けします。

インテル® MPI ライブラリーは通常、「thin」バイナリー実行ファイルをターゲットにします。例えば、バイナリー実行ファイルはすべて、コンパイル時にホスト CPU またはインテル® MIC アーキテクチャーのいずれかをターゲットにします。バイナリー実行ファイルの命令セットとマシンタイプの非互換性により、ホストをターゲットとしたバイナリー実行ファイルはインテル® MIC アーキテクチャーでは実行できません。逆も同様です。このため、シンメトリック・モデルでは、次のようにインテル® Xeon® プロセッサー向けとインテル® Xeon Phi™ コプロセッサー・プラットフォーム向けの 2 つのアプリケーションをビルドします。

```
(host)$ mpiicc -o hello hello.c

(host)$ mpiicc -mmic -o hello.mic hello.c
```


インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

インテル® MPI は、次の規則を使用してプラットフォーム向けの適切な実行ファイルを選択します。

1. `mpiexec.hydra` コマンドラインで指定された実行ファイルの名前が、そのホスト用の実行ファイル名であると仮定します。
2. インテル® MIC の実行ファイルの名前は、ホストの実行ファイルの名前に `I_MPI_MIC_PREFIX` 環境変数の内容 (プリフィックス) と `I_MPI_MIC_POSTFIX` 環境変数の内容 (ポストフィックス) を追加して生成されます。どちらの変数もデフォルトは空です。

下記の例では、`I_MPI_MIC_POSTFIX=.mic` に設定しています。ヘテロジニアスの起動は通常の方法で行えます。

```
(host)$ mpirun -n 64 -hostfile <hostfile>
./hello
```

バイナリー実行ファイル `hello` および `hello.mic` は、`hostfile` の定義に従ってインテル® Xeon® プロセッサー・ノードまたはインテル® Xeon Phi™ コプロセッサー・ノードで起動されます。

まとめ

インテル® Xeon Phi™ コプロセッサー向けにインテル® MPI を使用するときに重要な点は「インテル® Xeon® プロセッサーと同じ」です。インテル® MPI は、既存の MPI アプリケーションをインテル® Xeon Phi™ コプロセッサーで実行する 2 つのプログラミング・モデルを提供します。これらのアプリケーションの移植には、ある程度の知識と労力が必要です。

次のステップは、インテル® Xeon Phi™ コプロセッサー・システムに移植された実用アプリケーションを使用してより詳細にこれらのプログラミング・モデルを調査し、提案したアプローチの長所を比較して、プログラミング・モデルの選択と最も適切な移植方法について実用的な推奨事項を提供することです。このステップについては、将来の号で紹介する予定です。

参考資料 (英語)

1.インテル® Xeon Phi™ コプロセッサー: http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

2.メッセージ・パッシング・インターフェイス (MPI): http://www.mpi-forum.org

3.インテル® MPI ライブラリー: http://www.intel.com/go/mpi

BLOG highlights

インテル® Xeon Phi™ コプロセッサー : どんなもので、なぜ知る必要があるのか

パート 1: チップのスーパーコンピューター

TAYLOR KIDD (インテル コーポレーション)

ここでは、インテル® メニー・インテグレーテッド・コア・アーキテクチャーの概要を紹介します。「インテル® Xeon Phi™ コプロセッサー: どんなもので、なぜ知る必要があるのか」というタイトルを付けたのはそのためです。

インテル® MIC アーキテクチャー (インテル® Xeon Phi™ コプロセッサー) を見て、カードのスーパーコンピューターと呼んだ人がいます。シリコンの 1 つのスラブにすべてが詰め込まれたこのコプロセッサーは、チップのスーパーコンピューターと呼んでもおかしくありません。

この単純な事実は、大きな制限であるとともに大きな利点でもあります。1 つのチップにすべてが詰め込まれていることにより、設計者はコプロセッサー全体を統合して扱えるため、データをシリコンの別の部分に移動してから戻さなければならないために発生するさまざまな制限を回避することができます。その結果、コプロセッサーの異なるコア間で、これまでにない連携と通信の帯域幅が実現されています。また、搭載されているコアの数もこれまでにないものです。SKU によって異なりますが、コプロセッサーには最大 61 個のコアが搭載されています。そして、シリコンの 1 つのスラブにすべてが詰め込まれているのです。

この記事の続きはこちら (英語) でご覧になれます。

Go-Parallel.com (英語) でご覧になれます。

Go Parallel では、この他にも次のような関連トピックのブログをご覧いただけます: Translating Multicore Power into Application Performance (マルチコアのパワーでアプリケーション・パフォーマンスを引き出そう)。

従来のシリアル処理から 並列処理へ:

インテル® Cilk™ Plus による実用アプリケーションの変換

Chris Chrulski

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、**http://software.intel.com/en-us/articles/optimization-notice/#opt-jp** を参照してください。

この記事では、インテル® Cilk™ Plus を利用して実際に使用されている従来のシリアル・アプリケーションを並列アプリケーションに変換したときの高水準ワークフローと結果を紹介します。ワークフローは次のようになります。

1. インテル® VTune™ Amplifier XE でオリジナル・アプリケーションを解析し hotspot を特定します。
2. 並列デザインパターンを適用してシリアル・アルゴリズムを並列化可能なアルゴリズムに変換します。
3. インテル® Cilk™ Plus の機能により並列化を実装します。
4. インテル® Inspector XE 2013 およびインテル® VTune™ Amplifier XE で、並列化したアプリケーションの正当性とパフォーマンスを解析します。

シリアル処理の最適化とインテル® Cilk™ Plus の並列処理を併用することにより、従来のシリアル・アプリケーションのコードを大幅に変更せずに、8 コアのシステムでアプリケーションの重要なワークロードの実行速度が 25 倍も向上しました。

使用したアプリケーション

ここで最適化したアプリケーションは、インテル® Composer XE 製品のコンポーネントである profmerge ユーティリティーです。profmerge は、プロファイルに基づく最適化 (PGO: Profile Guided Optimization) インストルメンテーションにより生成される中間ファイルをマージします。これらのファイルは、プロファイルに基づくフィードバック・コンパイル・ステップ中により高いレベルの最適化を行えるようにするため、またインテル® コードカバレッジ・ツールでレポートを生成するため、効率良く使用できる形式にマージされます。

PGO インストルメンテーションをオンにしてアプリケーションをビルドし実行すると、1 つ以上のダイナミック・データ結果 (.dyn) ファイルが生成されます。このファイルには、コードパスの頻度と頻繁な値に関する情報が含まれます。各データファイルには、アプリケーション内で実行されるインストルメント済みルーチンのデータレコードが含まれます。一般に、さまざまなコードパスをカバーするため、またテストカバレッジを解析するため、アプリケーションは異なるデータセットで複数回実行します。

profmerge アプリケーションは 90 年代の中頃に C で記述された後、追加機能をサポートするためにこの数年で拡張されました。基本的なアプリケーション構造は、トップレベルのグローバルデータ構造と変数を使用して、マージの現在の状態、実行したすべてのルーチンのカウンター、分岐パスの頻度、間接関数呼び出しの共通のターゲット、マージプロセスに関する統計を保持します。

ワークロードの選択

アプリケーションの最適化を準備するため、インテル® Composer XE 製品の試行実行で生成された 1,000 を超える .dyn ファイルから、製品にとって重要で動作に再現性がある代表的なワークロードが選択されました。

シリアル・アプリケーションの解析

profmerge は、デュアルコア インテル® Xeon® X5365 システム (3.0GHz、16GB メモリー、Windows* 7 Enterprise) でワークロードの処理に約 10 分かかっていました。インテル® VTune™ Amplifier XE 2013 を使用して、デバッグ情報を含むリリースビルド構成のアプリケーションの hotspot を解析しました。収集した時間とデータの量を適切な量に制限するため、25 の .dyn ファイルのサブセットが使用されました。

解析結果 (図 1) では、非常にホットな (ルーチン内で直接費やされたアプリケーション時間の割合を示す青の横棒が長い) 2 つの hotspot が示されました。通常、これらのホットなルーチンは並列化により高速化が見込める候補と見なすことができます。しかし、ここでは、これらのルーチンはデータファイルのロードに関連していたため、簡単に並列化できる対象ではありませんでした。代わりに、並列化に適したプログラムであることを確認するため、シリアルルーチンを最適化しました。シリアルルーチンを最適化しないと、並列バージョンのパフォーマンスが制限されるためです。シリアルルーチンを最適化した後、hotspot 解析が再度行われました。この解析では、シリアルバージョンでより最適化が必要な部分は見つかりませんでした。次のステップは、同時に複数のコアを利用するため、並列化に使用する手法を調査することでした。

インテル® VTune™ Amplifier XE を使用したアプリケーションの解析に関する追加情報およびチュートリアルは、Stephen Blair-Chappell および Andrew Stokes の「インテル® Parallel Studio XE を使用した並列プログラミング」[2] を参照してください。

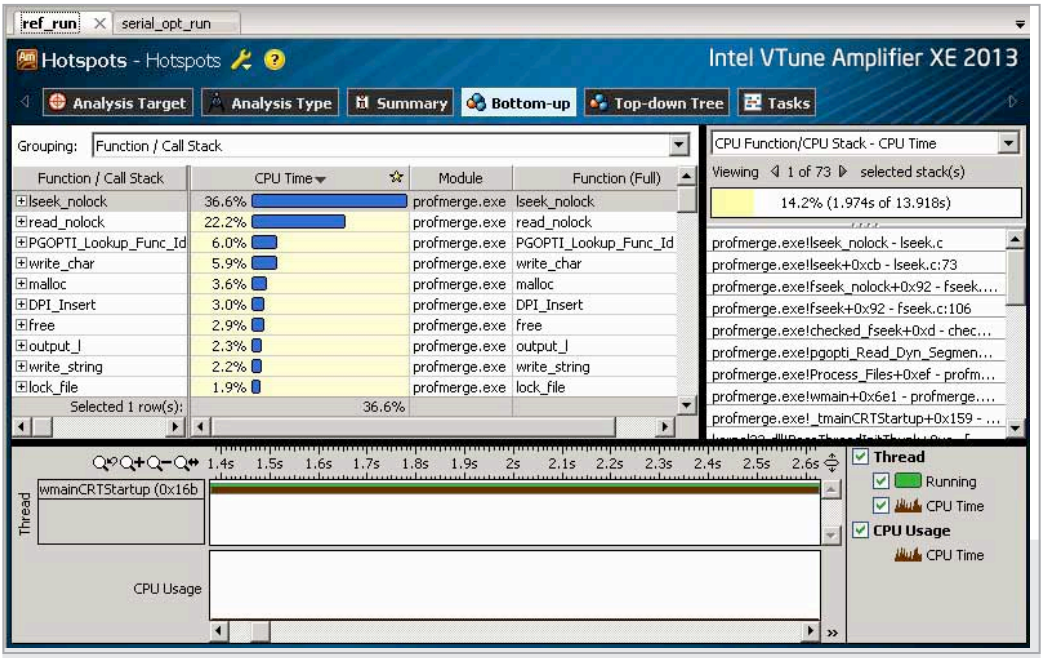


図 1. シリアル・ボトムアップ hostpot 解析

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

並列化デザインの適用

従来のシリアル・アプリケーションを並列アプリケーションに変換する最初のステップは、ワーカー間の通信のオーバーヘッドを最小限に抑えて、適切な粒度でワーカーが分割されるように、並列化する場所を探すことでした。

インテル® VTune™ Amplifier XE のトップダウン・ツリーを調べたところ、ほとんどの時間がアプリケーションの 3 つのメイン部分 (.dyn ファイルの読み取りとマージ、データの後処理、結果の書き込み) で費やされていることが分かりました。図 2 の青いバーは、そのルーチンが開始してから終了するまでに費やされた時間の量を表しています。ルーチン Process_Files の最初の 2 つのタスクが実行時間の約 80% を費やしていたため、これらのタスクを並列化の対象として選択しました。

マージして部分的な結果を生成する「分割統治」アルゴリズムを利用して実装されます。シリアルに処理したほうが効率的なサイズになるまで、セットは再帰的に分割されます。サブセットが完了すると、再起呼び出しは終了し、部分的な結果が互いにマージされます。

しかし、既存のアプリケーションを並列アプリケーションに直接変換するとき、マージ状態を示すのにグローバル変数を使っていることが問題になりました。そこで、各ワーカースレッドは新しいレコードを追加するか、既存のレコードの状態を更新することで、グローバルな状態を修正することができました。ロックでスレッドの安全性を確保することも考えましたが、得られるパフォーマンスが制限されるため、データ構造内のグローバル変数をカプセル化して、個々のスレッドが独自のデータ構造を持ち、変数をコピーできるようにしました。このカプセル化されたオブジェクトには、1 セットのファイルのデータをすべて含む 1 セットのレコードが含まれます。

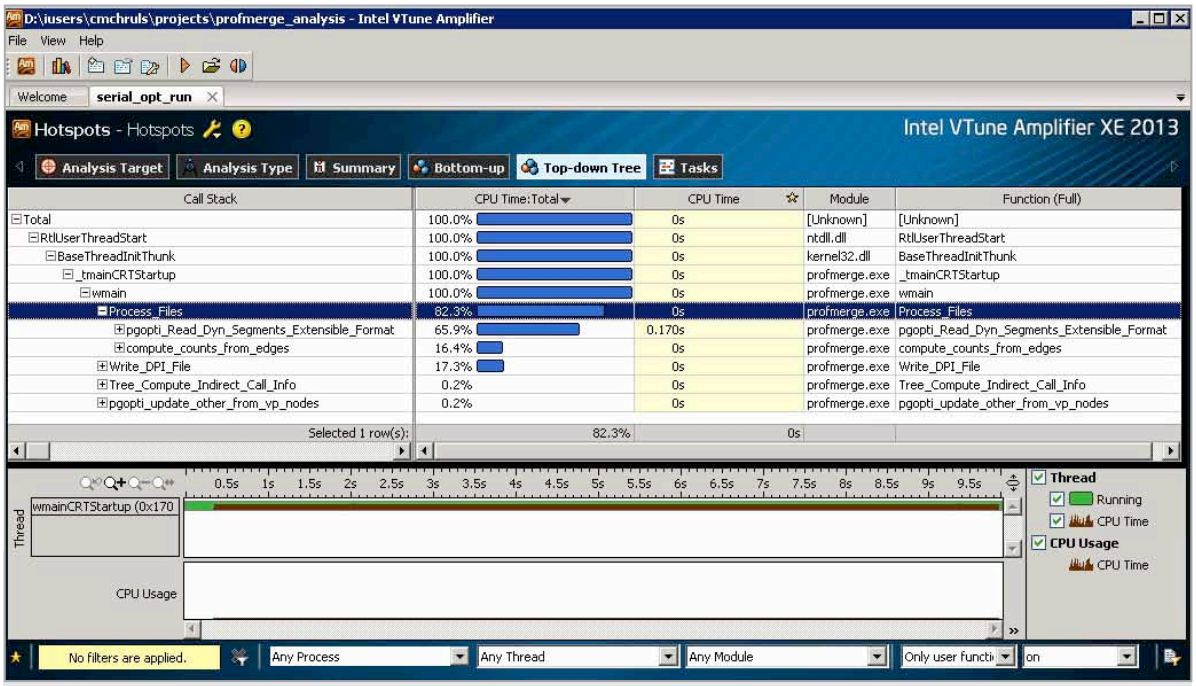


図 2. トップダウン・アプリケーション解析

オリジナルの実装では、ファイルはシーケンシャルに処理 (ファイルの先頭から最後までレコードをロード) し、集計データを更新するため、レコードの初期化処理を行っていました。レコードごとに費やされていた時間はわずかですが、ファイルをシーケンシャルに処理していたため、単一ファイル内で複数のワーカーを使用してもスケーラブルな結果を引き出す可能性はほとんどありませんでした。代わりに、異なるファイルを操作する別のワーカースレッドを使用する必要がありました。

並列アプリケーションに使用するデザインパターンが検討され、『Structured Parallel Programming (邦題: 構造化並列プログラミング)』(Michael McCool、ほか共著[3]) の第 8 章で説明されているフォーク・ジョイン (fork-join) デザインパターンが選択されました。このパターンは、ファイルのセットを半分に分割して処理した後、

状態をオブジェクトにカプセル化する修正には多少の労力が必要でした。グローバル変数にアクセスするか、グローバル変数を変更するすべての場所で状態オブジェクトを処理するように修正しなければなりませんでした。また、個別のオブジェクト・インスタンスのマージをサポートする新しいコードも記述する必要がありました。しかし、アプリケーション・ロジックとコードの大部分はほとんど手を加えられていません。アプリケーションによっては、手動でカプセル化する代わりに、インテル® Cilk™ Plus のハイパーオブジェクトを利用することができます。

```
// .dyn ファイルのサブセットのマージ。この関数が終了した後、'merger'
// オブジェクトは範囲 [lower _ bound..upper _ bound) の 'node _ vect' の
// ファイルを処理する。最小範囲に達するまで、このルーチンは範囲の
// オーバーラップしないサブセットで再帰的に呼び出される。
//
// パラメーター:
// node _ vect - マージするファイルの配列
// lower _ bound - マージする最初の要素のインデックス
// upper _ bound - マージする最後の要素の 1 つ先のインデックス
// merger - マージ結果を格納するデータ構造
static void merge _ nodes(
    FileNodeVect        *node _ vect,
    int                  lower _ bound,
    int                  upper _ bound,
    DpiMerger           *merger)
{
    // 処理する範囲が十分小さい場合、別のスレッドに分割せず
    // ファイルをすべてシリアルに処理
    if (upper _ bound - lower _ bound <= file _ count _ threshold) {
        for (int i = lower _ bound; i < upper _ bound; i++) {
            merger->merge _ file(*node _ vect[i]);
    }
    else {
        // 範囲を半分に分割して別々に処理
        int mid _ point = lower _ bound + (upper _ bound - lower _ bound) / 2;

        // インテル® Cilk™ Plus の継続スレッドで渡される merger
        // オブジェクトと同時に処理できるようにファイルの上半分の
        // 新しいマージ・オブジェクトを作成
        DpiMerger *high _ merge = pgopti _ create _ dpi _ merger();

        // [lower _ bound..mid _ point) をマージ
        cilk _ spawn merge _ nodes(node _ vect, lower _ bound, mid _ point, merger);

        // [mid _ point..upper _ bound) をマージ
        merge _ nodes(node _ vect, mid _ point, upper _ bound, high _ merge);
        // 両方の半分の結果が完了して呼び出し元の DpiMerger
        // オブジェクトにマージできるように、明示的にここで同期
        cilk _ sync;
        merge _ mergers(merger, high _ merge);
        pgopti _ delete _ dpi _ merger(high _ merge);
    }
}
```

図 3. cilk_spawn コードの構造

```
// 各レコードを処理し、エッジカウントに基づいて各ブロックの
// 実行カウントを格納
void compute_counts_from_edges(DpiNodeVec *nodes, size_t count)
{
     size_t i;
     cilk_for (i = 0; i < count; i++) {
        dpi_node_compute_counts_from_edges(nodes[i]);
    }
}
```

図 4. cilk_for コードの構造

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

インテル® Cilk™ Plus による並列処理の実装

図 3 は、分割統治アルゴリズムの実装を示しています。merge_nodes ルーチンには、処理するファイルの配列、入力として処理する要素の範囲のインデックス、結果を格納するデータ構造へのポインターを指定します。範囲を 1 つのセットとして処理するべきか、範囲をサブセットに分割するべきかを判断するチェックが行われます。範囲を分割するべきと判断されると、範囲を中間点で分割して、2 つの関数呼び出しが再帰的に行われます。インテル® Cilk™ Plus を利用したときにこれらの 2 つの呼び出しが別々に処理されるようにするため、これらの呼び出しの 1 つで使用される、カプセル化された状態オブジェクトの新しいインスタンスを作成しました。

修正後のアルゴリズムを実装してテストした後、図 3 の青で示したインテル® Cilk™ Plus のキーワードを追加して並列化しました。最初の再帰呼び出しに cilk_spawn 宣言子を追加し、2 つ目の再帰呼び出しの後に cilk_sync 宣言子を追加するだけで、処理を並列実行することができました。両方の範囲をマージする前に両方の範囲を完了する必要があるため、cilk_sync を追加しました。

並列実装に選択されたアプリケーションの 2 つ目の部分では、ほかのレコードと関係なくすべてのデータが各レコードにロードされた後、処理が行われます。コード配列のすべての要素を処理していた単純な ‘for’ ループは、ループ制御文で cilk_for キーワードにすることで、並列ループに変換されました (図 4)。

これらの変更と cilk.h ヘッダーファイルをインクルードしただけで、従来のシリアル・アプリケーションが並列アプリケーションへ変換され、エラーとパフォーマンスを解析する準備が整いました。ここまでの変更は、スレッドを手動で作成してスレッド間のデータ共有を管理するよりもはるかに簡単でした。

スレッドエラーの確認

インテル® Inspector XE 2013 を利用して、デバッグビルド構成を使った並列アプリケーションの潜在的な問題を確認しました。ワークロードから 25 の .dyn ファイルを使用して、最高レベルの解析モード「デッドロックとデータ競合の検出 (Locate Deadlocks and Data Races)」でスレッドエラー解析を実行しました (図 5)。収集するデータ量およびデータ収集のオーバーヘッドのため、より小さなワークロードを選択することを推奨します。

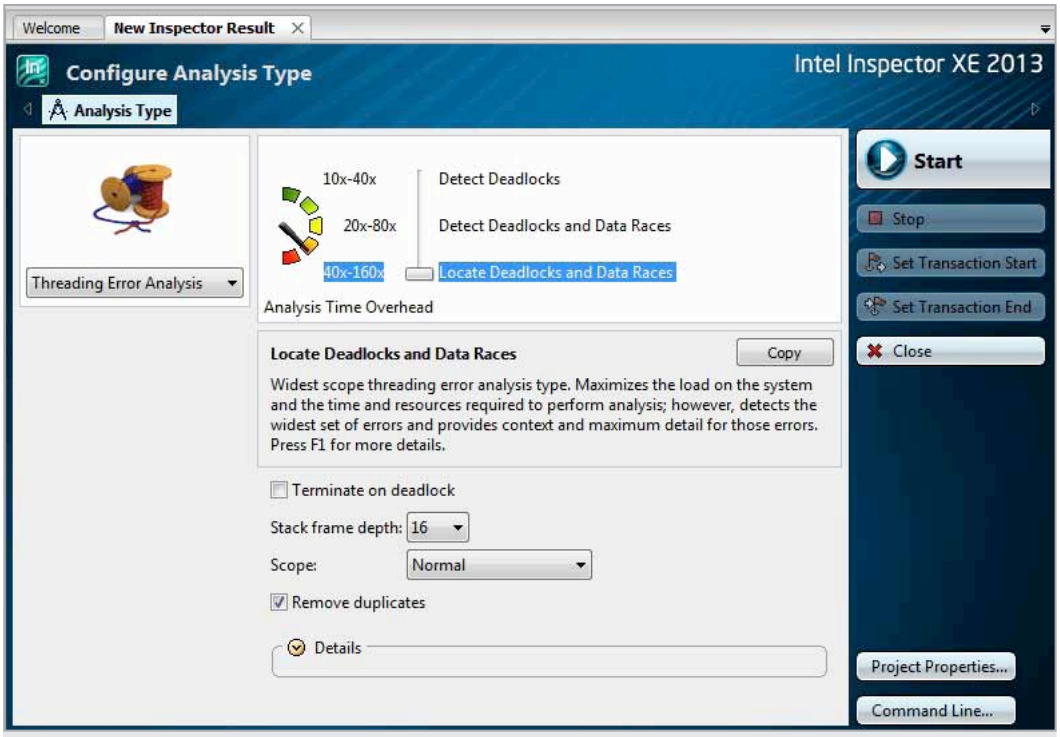


図 5. スレッド解析を開始

C のオリジナルの実装ではグローバル変数を使っていたため、このスレッドエラー解析では、個々のスレッドが共通グローバル変数を変更している問題を調べました。図 6 は、可能性のあるデータ競合条件を解析した結果のサンプルです。上のペインには潜在的な問題領域のサマリーがリストされ、下のペインにはデータ競合に関係する場所とスレッドに関するより詳細な情報が表示されます。このサンプルでは、並列領域内のルーチンでグローバル変数が使用され、2 つの異なるスレッドでこの変数の値を設定していました。複数回の実行のレポートを参照してコードを変更し、データ競合を解決しました。

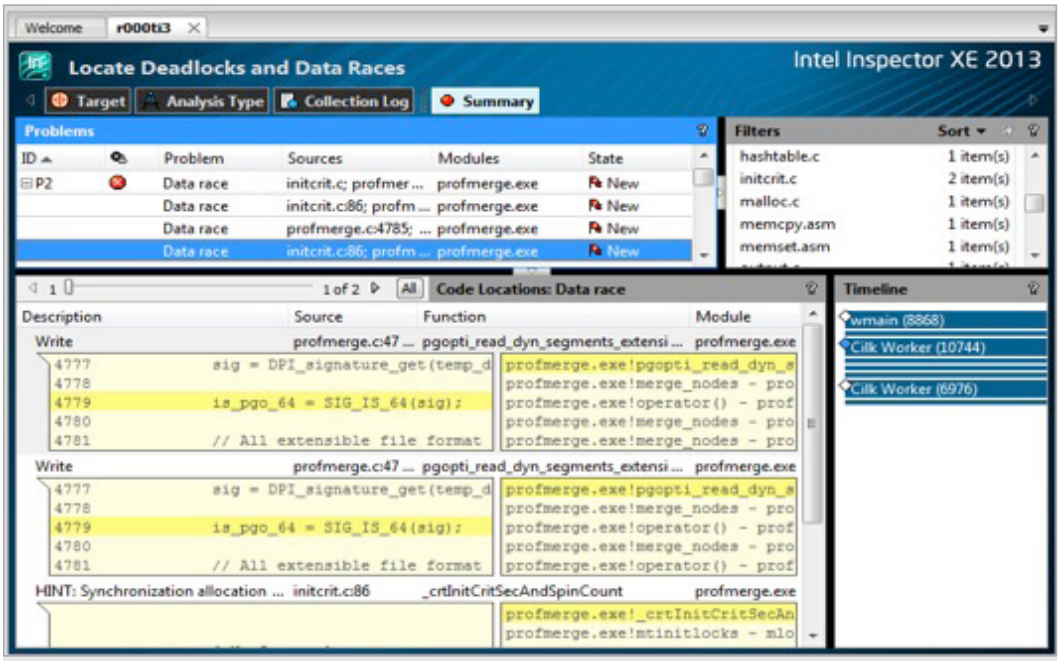


図 6. データ競合レポート

図 7. CPU 使用率別 hotspot

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

パフォーマンスとスケーラビリティーの解析

処理コアの使用効率を確認するため、インテル® VTune™ Amplifier XE 2013 を利用して、オリジナルの 1,000 以上の入力ファイル・データセットで別の hotspot 解析を実行しました。

インテル® VTune™ Amplifier XE のサマリー結果ウィンドウで [Hotspots by CPU Usage (CPU 使用率別 hotspot)] オプションを選択して、CPU コアの使用状況に関する情報を調べました。ヒストグラムは、コアがアプリケーションを実行していた時間を示しています。図 7 を見ると、8 つのコアすべてが、アプリケーションで効率良く利用されていたことが分かります。

アプリケーションのスケーラビリティーは、ワーカースレッドの数を変更して確認しました。図 8 で示されているように、アプリケーションの主な 3 つの処理に費やされた時間をデータとして収集しました。青い部分は cilk_spawn によって分割統治アルゴリズムが適用された処理を示します。赤い部分は cilk_for ループに変換されたループを示します。緑い部分は残りのシリアル処理を示します。図 8 のように、より多くのコアが利用されるとパフォーマンスが向上することが理想的です。しかし、このアプリケーションの場合、8 つのスレッドがアクティブになると、コアを増やしても並列処理のパフォーマンスは向上しません。さらにコードを変更しないと、このコードではコアを増やしてもスケーリングしないことが確認できたため、より多いコアのマシンについての試験を終了しました。このスケーラビリティー制限の調査により、選択したアルゴリズムによる部分的な結果を結合したり、各レコードを処理するごとに動的に割り当てられ解放されたメモリーからの部分結果を結合するため、新たな作業が必要になることが分かりました。シリアル・アプリケーションから並列アプリケーションへの変換では、C ランタイム malloc と free への呼び出しは変更されませんでした。動的なメモリー操作で単一のヒープを共有すると、システムはメモリー操作を適切に行うためにヒープをロックするため、ボトルネックが発生します。さらにパフォーマンスを向上するには、スレッドごとに別のヒープを使うメモリー・アロケーター (インテル® スレッディング・ビルディング・ブロック [インテル® TBB] で提供されるスケラーブル・メモリー・アロケーターなど) を利用するようにコードを変更する必要があります。

BLOG highlights

インテル® Cilk™ Plus の部分配列の新しい規則

ARCH D. ROBISON (インテル コーポレーション)

インテル® Cilk™ Plus のユーザーは、http://software.intel.com にポストされたインテル® Cilk™ Plus の改訂版の仕様にぜひ目を通してください。Clark Nelson がそのほとんどを手がけた今回の改訂により、以前の仕様よりも標準仕様に沿っており、以前の仕様で明確でなかった多くの点が明確になりました。ここでは、言語拡張を向上する新しい仕様の 2 つの重要な変更について説明します。まず、コンパイラーがより効率良いコードを生成できるようになったことです。もう 1 つは、部分配列の矛盾が解決されたことです。

部分配列のより効率良い制御

p と q がポインターであると仮定して、次の部分配列の代入について考えてみます。

```
p[0:n] = q[0:n]+1;
```

この文は `p[i]=q[i]+1 for i in {0...n-1}` を設定します。では、これら 2 つのシーケンスがメモリーでオーバーラップした場合はどうなるでしょうか? オリジナルの仕様では、APL と Fortran 90 の慣例に従い、左辺に代入される前に、まず右辺を評価していました。この慣例に従うと、オーバーラップの有無に関係なくこの例は定義可能です。

しかし、C++ の原則は「最小限のペナルティーによる抽象化」です。APL と Fortran 90 のアプローチは、オーバーラップがないことをコンパイラーが証明できない場合は常に、右辺の結果の一時配列をコンパイラーが生成する必要があるため、この原則に違反します。

この記事の続きはこちら (英語) でご覧になれます。

Go-Parallel.com (英語) でご覧になれます。

Go Parallel では、この他にも次のような関連トピックのブログをご覧いただけます:Translating Multicore Power into Application Performance (マルチコアのパワーでアプリケーション・パフォーマンスを引き出そう)。

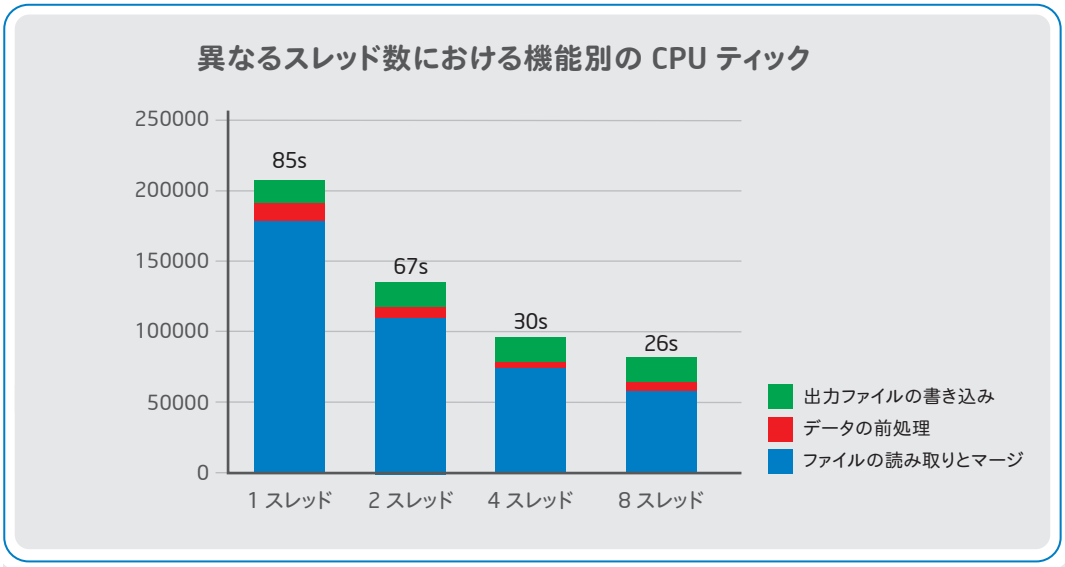


図 8. 並列アプリケーションの実行時間

まとめ

コードを大幅に変更することなく、従来のシリアル・アプリケーションを並列アプリケーションへ変換することができました。オリジナルのアプリケーションでは、テストシステムのワークロードを処理するのに 650 秒かかっていました。インテル® VTune™ Amplifier XE 2013 により hotspot を特定して変更したところ、実行時間は 85 秒に短縮されました。グローバル変数を独立したオブジェクトにカプセル化するコードの再構成には多少の労力が必要でしたが、並列処理を行うためにアプリケーションのコードの大部分を変更することはありませんでした。インテル® Cilk™ Plus を利用して選択したワークロードに並列処理を実装したところ、実行時間は 26 秒に短縮されました。アプリケーションで複数の CPU コアを活用するために必要な変更は単純であり、手動でスレッドを作成してデータ共有と通信を調整するよりもはるかに簡単であるにもかかわらず、従来のアプリケーションのパフォーマンスを大幅に向上させることができたのです。

参考資料

1. インテル® C++ コンパイラー XE 13.0 ユーザー・リファレンス・ガイド、資料番号 323273-130JA。
2. Stephen Blair-Chappell および Andrew Stokes、Parallel Programming with Intel® Parallel Studio XE、Wrox, 2012。
3. Michael McCool、Arch Robison、James Reinders 共著、Structured Parallel Programming: Patterns for Efficient Computation、Morgan Kaufmann、2012。

「インテル® Parallel Studio XE のツールとインテル® Cilk™ Plus を利用することにより、従来のシリアル・アプリケーションのコードを大幅に変更することなく、重要なワークロードの実行速度が 25 倍も向上しました。」

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

リソースおよび WEB サイト

Go Parallel

Go Parallel の目的は、「マルチコアのパワーをアプリケーションのパフォーマンスに変える」ことで開発者を支援することです。豊富な情報を掲載するこのサイトは、マルチコア関連のブログ、ニュース、ビデオ、機能説明、その他の役立つリソースの価値あるクリアリングハウスです。

“What If” Experimental Software

開発中のインテルの先端研究や最新テクノロジーを試せるとしたらどうでしょう?そして、ユーザーのフィードバックが将来の製品に役立つとしたら?それを可能にするのが What If Experimental Software です。最先端のツールをテストしたり、他のユーザーと協力して、“What If?” ブログやサポートフォーラムにより意見を交換できます。

インテル® デベロッパー・ゾーン

並列プログラミングのマネージャビリティーから仮想化やビジュアル・コンピューティングまで、さまざまな開発者向けに広範なソフトウェアのトピックが用意されています。豊富なコンテンツのコレクションには、インテル® ソフトウェア・ネットワーク TV、ブログ、ビデオ、ツール、ダウンロードが含まれます。

最新ソフトウェアの概要

最新のインテル® ソフトウェアの使い方を説明するビデオで、メモリーチェック、スレッドチェック、hotspot 解析、ロックと待機の解析、その他の重要な役割を担っているソフトウェアの機能をご覧になれます。

インテル® Inspector XE ›

インテル® VTune™ Amplifier XE ›

インテル® Software Evaluation Center

インテル® Software Evaluation Center では、インテル® ソフトウェア開発製品の 30 日間評価バージョンをダウンロードできます。評価用ライセンスをリクエストした後、インテル® プレミアサポートのアカウントを作成すると、評価期間中にインテル® プレミアサポートの無料サポートを受けることができます。また、インテル® ソフトウェア・ネットワーク・フォーラム (英語) の無料サポートにアクセスすることもできます。インテル® Parallel Studio を評価する場合は、インテル® ソフトウェア・ネットワーク・フォーラム (英語) の無料サポートにのみアクセスできます。

最適化に関する注意事項

インテル® コンパイラーは、互換マイクロプロセッサー向けには、インテル製マイクロプロセッサー向けと同等レベルの最適化が行われない可能性があります。これには、インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2)、インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3)、ストリーミング SIMD 拡張命令 3 補足命令 (SSSE3) 命令セットに関連する最適化およびその他の最適化が含まれます。インテルでは、インテル製ではないマイクロプロセッサーに対して、最適化の提供、機能、効果を保証していません。本製品のマイクロプロセッサー固有の最適化は、インテル製マイクロプロセッサーでの使用を目的としています。インテル® マイクロアーキテクチャーに非固有の特定の最適化は、インテル製マイクロプロセッサー向けに予約されています。この注意事項で対象としている特定の命令セットに関する詳細は、該当製品のユーザーズガイドまたはリファレンス・ガイドを参照してください。

改訂 #20110804

アプリケーションの
ビルド方法を変える
最新のプログラミング資料

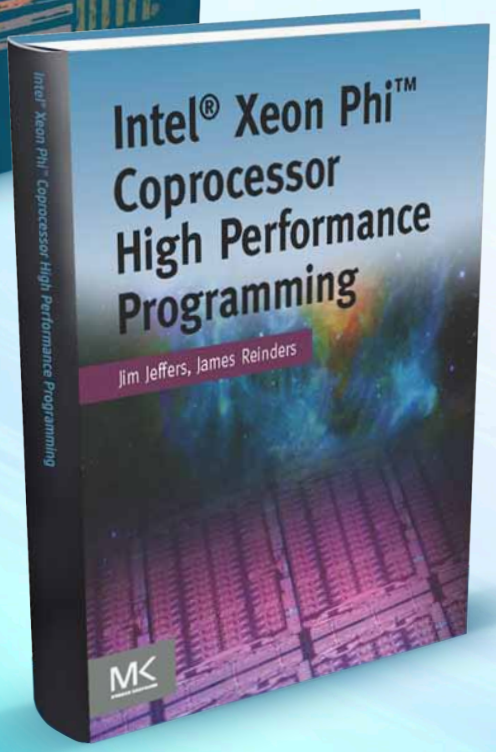
スレッド化から HPC まで最新の並列プログラミングを詳細に解説

構造化並列プログラミング
Structured Parallel Programming

Michael McCool、Arch Robison、James Reinders 共著
詳細 (英語): www.parallelbook.com

「私はこれまで、スレッド化に苦労している同僚やスレッド関連の質問を受けた相手に推薦したり、マルチスレッドの主要な概念を学べて、さらに現在の主要な実装をすべて網羅した新しい書籍が登場することを切望していました。そして、ついにその書籍を手にすることができたのです。」

ドリームワークス アニメーション
主任エンジニア
Martin Watt 氏



インテル® Xeon Phi™ コプロセッサー ハイパフォーマンス・プログラミング (日本語出版予定)
Intel® Xeon Phi™ Coprocessor High Performance Programming

Jim Jeffers、James Reinders 共著
詳細 (英語): www.lotsofcores.com

「本書はすべての HPC プロフェッショナルに必携の一冊です。最近のアーキテクチャーで利用されているさまざまなアルゴリズムのパフォーマンスに関する解説を含む、ハイパフォーマンス・コンピューティングの普遍的な基本事項を我々に思い出させてくれます。本書を読むことで、この先何年も役立つ強力なツールを手に入れることができるでしょう。」

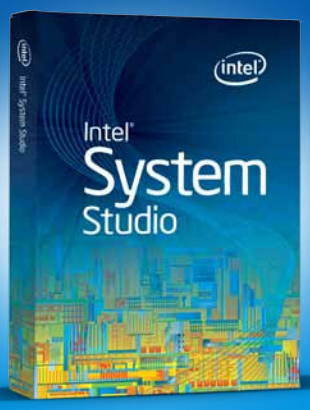
ストーニーブルック大学
Institute for Advanced Computational Scienc
Robert J. Harrison 氏

intel® Software

© 2013 Intel Corporation. 無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、Intel Atom、Intel Core、Intel Inside ロゴ、Cilk、VTune、Xeon、Xeon Inside、Intel Xeon Phi は、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

インテル® System Studio

intel® Software



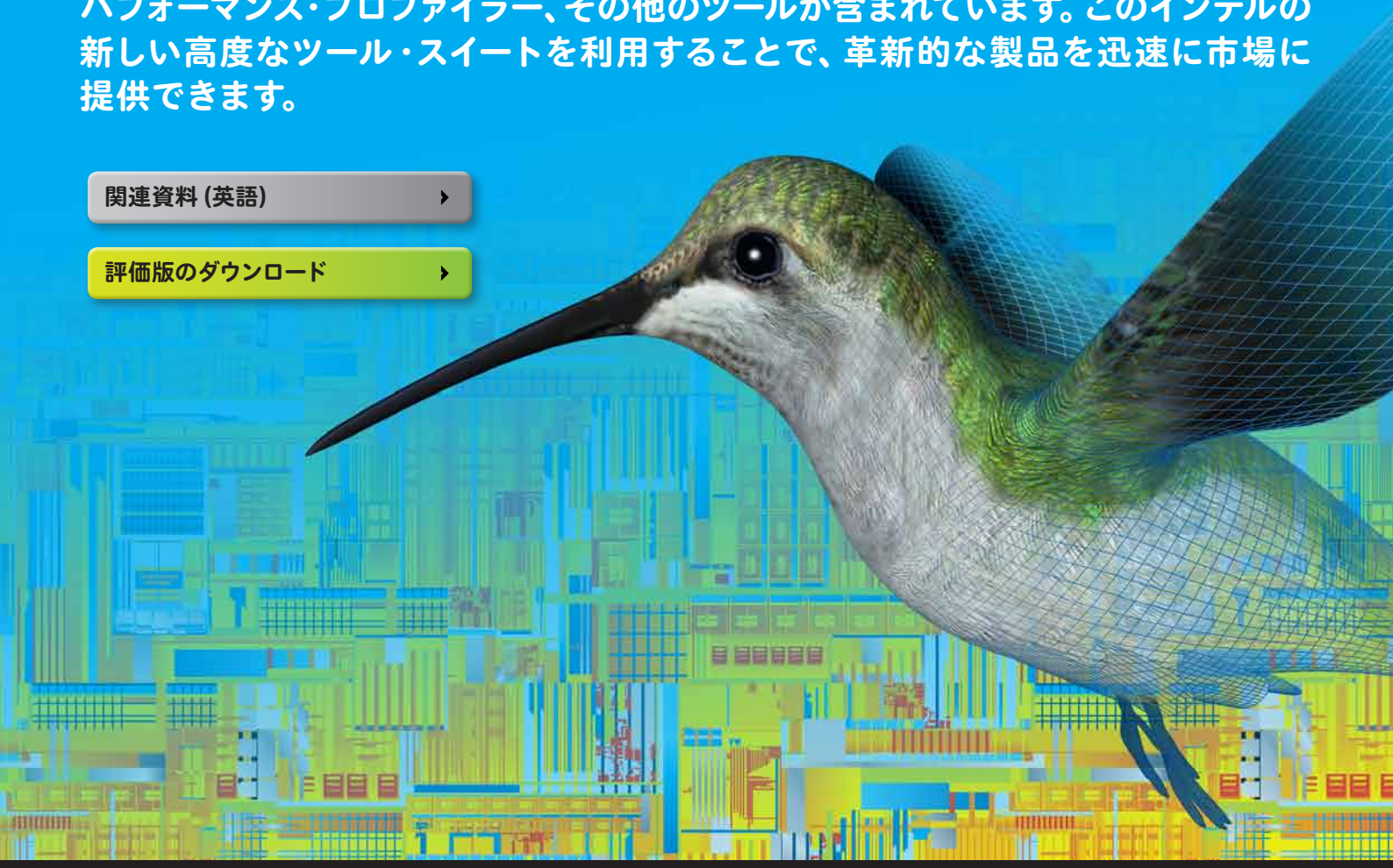
組込みおよびモバイル システム開発を促進

最適な消費電力、パフォーマンス、信頼性について システムレベルの深い洞察を提供

インテル® System Studio には、コンパイラー、JTAG/アプリケーション・デバッガー、メモリー/スレッド・エラー・チェッカー、高度なシステムオンチップ (SoC) 消費電力/パフォーマンス・プロファイラー、その他のツールが含まれています。このインテルの新しい高度なツール・スイートを利用することで、革新的な製品を迅速に市場に提供できます。

関連資料 (英語) ›

評価版のダウンロード ›



© 2013 Intel Corporation. 無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、Intel Atom、Intel Core、Intel Inside ロゴ、Cilk、VTune、Xeon、Xeon Inside、Intel Xeon Phi は、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。