ソフトウェアは実際に新しい命令セットを使用しているのか?

この記事は、インテル® デベロッパー・ゾーンに公開されている「Question: Does Software Actually Use New Instruction Sets?」の日本語参考訳です。

長年、インテルと他のプロセッサー・ベンダーは、携帯電話、タブレット、ワークステーション、サーバー、その他のコンピューティング・デバイスに搭載されているプロセッサーに命令 (英語) を追加し続けてきました。特定の計算タスク向けに命令を追加することは、プロセッサーのパイプラインを複雑にしたり、実現不可能なクロック周波数でプロセッサーを駆動することなく処理効率を向上できる方法です。新しい命令でいくつかの古い命令を置換できる場合、特定のタスクにおいてパフォーマンスを数倍向上できます。また、新しい命令は、インテル® ソフトウェア・ガード・エクステンション (インテル® SGX) (英語) やインテル® Control-Flow Enforcement (CET) (英語) などの全く新しい機能をもたらします。

1 つの懸念は、命令セットアーキテクチャー (ISA) に追加された新しい命令を、ユーザーがどれぐらい早急に、そして簡単に利用できるようになるかということです。オペレーティング・システムとプログラムは、一般に下位互換性を持ち、あらゆる種類のハードウェアで動作することを考慮すると、実際に新しい命令によって利点がもたらされるかは不明です。以前は、新しいアーキテクチャー向けにソフトウェアを再コンパイルして、互換性のない古いマシンで実行しないようにチェックを追加していました (例えば、「このプログラムは、このハードウェアではサポートされていません」といったメッセージを表示するなど)。

私は、お気に入りの仮想プラットフォーム・ツール Wind River* Simics* (英語) を使って、今日のソフトウェアが古いハードウェアとの互換性を保ちながら、新しい命令をどの程度使用しているかを調べてみました。

テスト環境のセットアップ

ソフトウェアが新しいハードウェアに動的に適合できるか調査するため、Simics* の「汎用 PC」プラットフォームで 2 つの異なるプロセッサー・モデルを実行しました。1 つは、第 1 世代インテル® Core™ i7 プロセッサー (開発コード名 Nehalem) で、もう 1 つは、第 6 世代インテル® Core™ i7 プロセッサー (開発コード名 Skylake) です。

第 1 世代インテル® Core™ i7 プロセッサーは 2008 年後半にリリースされ、私はインテル® Core™ i7-970 プロセッサー搭載の PC を 2009 年前半に購入しました。3 つのメモリーチャネルを備えた素晴らしいマシンで、9GB RAM を搭載することができました。第 6 世代インテル® Core™ i7 プロセッサーは、約 7 年後の 2015 年半ばにリリースされました。

それぞれのハードウェアで 3 つの異なる Linux* オペレーティング・システム (OS) をブートしてみました。

- Ubuntu* 16.04 カーネル 4.4 (2016 年前半リリース)
- Yocto* 1.8 カーネル 3.14 (2014 年前半リリース)
- Busybox* カーネル 2.6.39 (2011 年リリース)

どちらのハードウェアでも同じディスクイメージを使用し、実行されるソフトウェア・スタックに変更はありません。仮想ハードウェアの設定のみ異なります。私は、新しい Linux* オペレーティング・システムは、新しいハードウェアでは新しい命令を使用すると予想しました。実際、その通りになりましたが、驚くべき発見もあります。

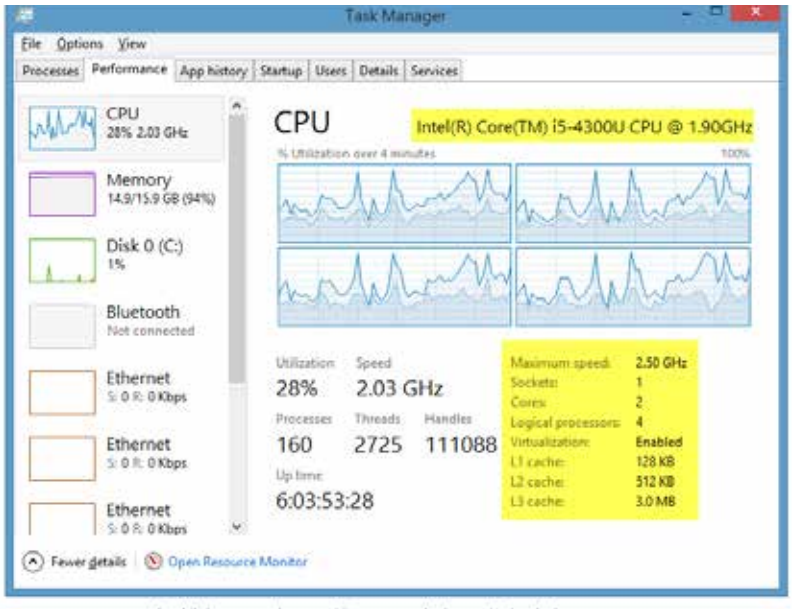
それぞれのセットアップで Simics* インストルメンテーションは、命令の動的頻度 (実行された命令数) をカウントし、オペコード (Simics* プロセッサーで使用されるアセンブリー命令の名前) でグループ化しました。Simics* インストルメンテーションは非干渉で、ターゲットシステムの実行に影響しません。ハードウェア・レベルで動作するため、BIOS と OS カーネルのブート中もインストルメントできます。またターゲットシステムで実行中のソフトウェアは、実行がインストルメントされているかどうか区別できません。それぞれのセットアップを 60 仮想秒実行しました。これは、すべてのケースにおいて、BIOS と OS をブートし、デスクトップまたはユーザープロンプトを表示するのに十分な時間です。それぞれの実行の後で、上位 100 の命令とその回数をリストし、さらなる処理と解析のためデータを Excel* にエクスポートしました。

ハードウェア特性の調査

ここでは、ソフトウェア・スタックは、ハードウェア特性に応じて実行するコードを動的に適合できると仮定しています。つまり、単一のバイナリーのセットアップは、異なるハードウェア・プラットフォームで異なる命令を使用する可能性があります。

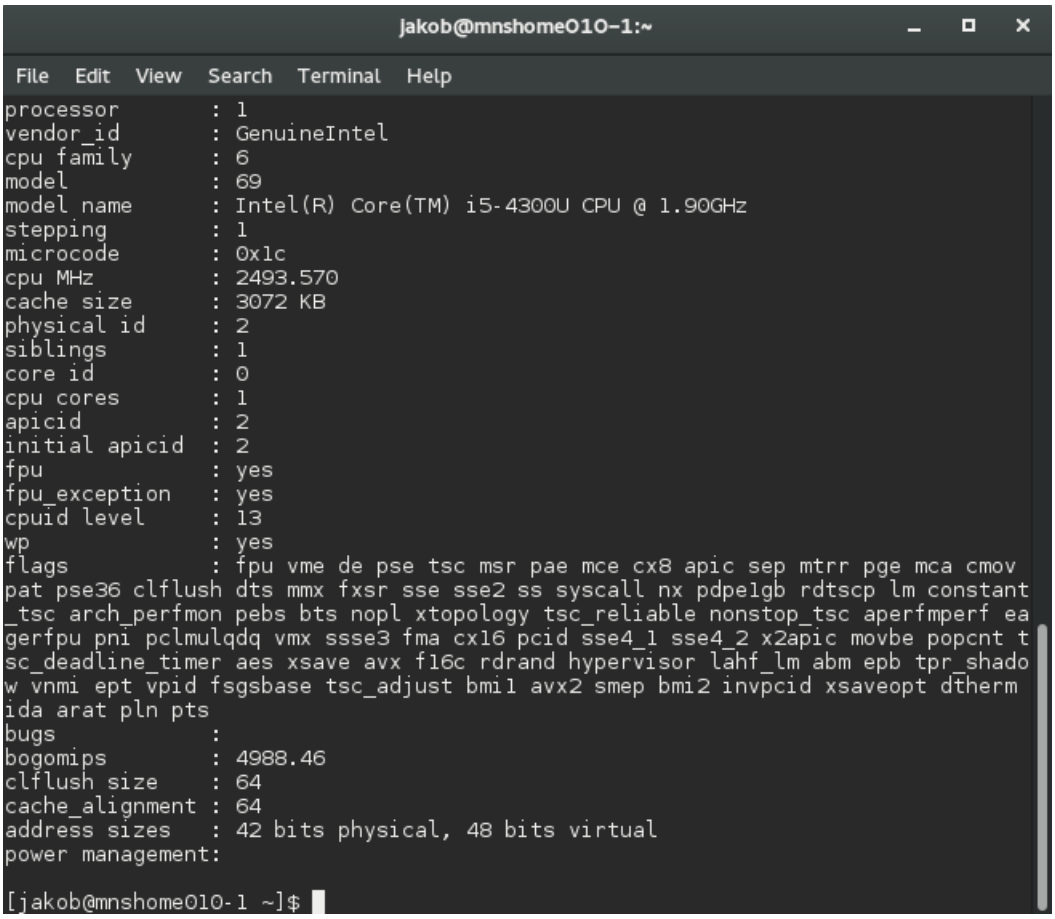
そのような動的な適合では、ソフトウェアを実行しているハードウェアの特性を検出することが重要です。プロセッサーのリリース頻度が少なく、その間隔が長かった頃は、ソフトウェアがプロセッサーをチェックして、インテル® 80386/80486 プロセッサーや Motorola* 68020/68030 などの検出されたプロセッサーに応じて適合させていました。当時は、プロセッサーが数年置きにリリースされ、種類が限られていました。今日では、多種多様なシステムから成る大多数のインストール・ベースが存在するため、プロセッサーの多様性が増しています。これに対応するため、IA プロセッサーには CPUID 命令があります。CPUID は、それ自体がシステムと言え、ハードウェアのさまざまな情報を得ることができます。

皆さんは、すでに CPUID 命令からの情報をその情報源に気付かずに目にしているでしょう。プログラムがプロセッサー・タイプを表示する場合、それは CPUID 出力を基にしています。例えば、Microsoft* Windows* 8.1 タスク・マネージャーは、プロセッサー・タイプとその特性を表示します。これらもすべて CPUID からの情報です。



(and it does seem I am working my poor laptop pretty hard…)

Linux* では、"cat /proc/cpuinfo" を実行すると、利用可能なプロセッサー機能や命令セットを示すフラグを含むプロセッサーの詳細に関する CPUID 情報を確認できます。追加命令セットごとに個別のフラグがあり、ソフトウェアはそれらを使用して利用可能な機能を特定できます。例えば、第 4 世代インテル® Core™ i5 プロセッサーでは、次のような情報が表示されます。



CPUID は、ソフトウェアに利用可能な各種命令セット拡張とハードウェア機能を通知しますが、ホストに応じて異なるコードを選択するため、ソフトウェアは実際にどのようにフラグを使用しているのでしょうか?「命令セット X が利用可能な場合、これを実行する」というような条件をコードに記述することは合理的ではありません。コードが、実行中に変化しない同じ情報を何度もチェックしないようにする必要があります。

Linux* カーネルでこれを行う最も一般的な方法は、利用可能な命令セットに応じて、同じ関数の異なる実装へのポインターを設定します。Linux* カーネル 4.13.5 (http://elixir.free-electrons.com/linux/v4.13.5/source (英語)) の arch/x86/crypto/sha1_ssse3_glue.c に良い例があります。

```
static int register_sha1_ssse3(void)
{
        if (boot_cpu_has(X86_FEATURE_SSSE3))
                return crypto_register_shash(&sha1_ssse3_alg);
        return 0;
}
```

```
static int register_sha1_ni(void)
{
        if (boot_cpu_has(X86_FEATURE_SHA_NI))
                return crypto_register_shash(&sha1_ni_alg);
        return 0;
}
```

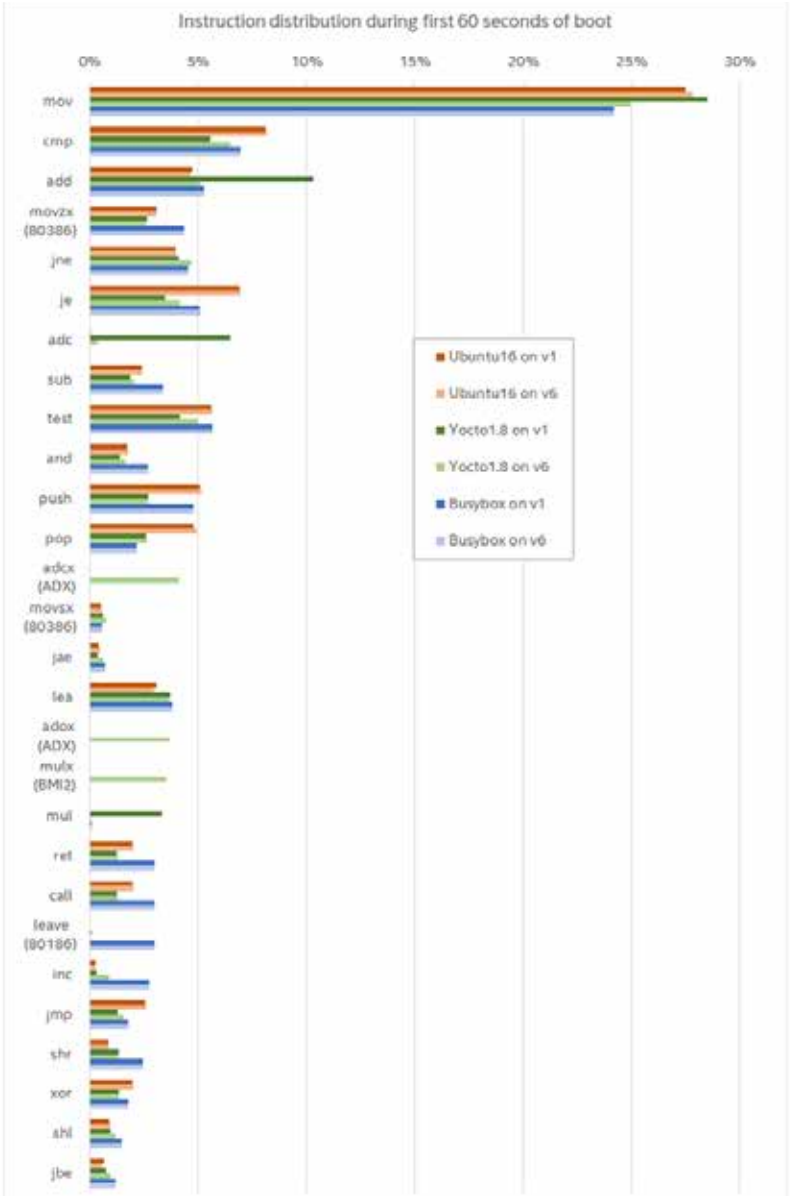
```
static int __init sha1_ssse3_mod_init(void)
{
        if (register_sha1_ssse3())
                goto fail;

        if (register_sha1_avx()) {
                unregister_sha1_ssse3();
                goto fail;
        }

        if (register_sha1_avx2()) {
                unregister_sha1_avx();
                unregister_sha1_ssse3();
                goto fail;
        }

        if (register_sha1_ni()) {
                unregister_sha1_avx2();
                unregister_sha1_avx();
                unregister_sha1_ssse3();
                goto fail;
        }

        return 0;
fail:
        return -ENODEV;
}
```

これらの関数は、ブート・プロセッサーが特定の命令セットをサポートしているかチェックして、適切なハッシュ関数を登録します。最も効率良いソリューションが使用されるように、これらの関数は優先度順に呼び出されます。この特定のケースに最適なソリューションは、特殊な SHA 命令をサポートする (英語) プロセッサーのようですが、それが利用できない場合、カーネルはインテル® AVX またはインテル® SSE 命令を使用します。

このことを考慮して、コードを実行し、どの命令が使用されるか見てみましょう。

結果

以下のグラフは、6 つの異なる実行の結果を示したものです (3 つの異なるオペレーティング・システムと 2 種類のプロセッサー・コアのペア)。いずれかの実行で、1% を超えるすべての命令が表示されます。「v1」は、第 1 世代インテル® Core™ i7 プロセッサーで実行されたソフトウェア・スタックを示し、「v6」は第 6 世代インテル® Core™ i7 プロセッサーで実行されたソフトウェア・スタックを示します。



最初に分かることは、ほとんどの命令は特に新しいものではなく、インテル® 8086 の移動、比較、ジャンプ、加算といった基本命令です。新しい命令は対応する命令セット拡張とともに表示されていますが、グラフでは 28 命令中 6 命令のみです。

プロセッサーの世代に加えて、ソフトウェア・スタックにも多様性が存在することは明らかです。例えば、古い BusyBox セットアップでは LEAVE 命令が使用されますが、ほかのセットアップでは一般にこの命令は使用されず、POP 命令の数もはるかに少ないことが分かります。しかし、このことは、新しい命令が利用可能な場合にソフトウェア・スタックがそれらを利用しているかどうかを示すものではありません。最も興味深い違いは、ソフトウェア・スタックが同じで、プロセッサーの世代が異なる場合に見られます。

異なる実行は、異なる処理を行います。このケースでは、すべての実行は Linux* のブートですが、カーネルの構成やルート・ファイル・システムのコンテンツに違いがあります。異なるディストリビューションとカーネル・バージョンは、異なるコンパイラー・バージョンとコンパイラー設定でビルドされます。そのため、同じソースコードであっても、生成される実行コードは異なる可能性があります。マシンコードの生成方法は多岐にわた

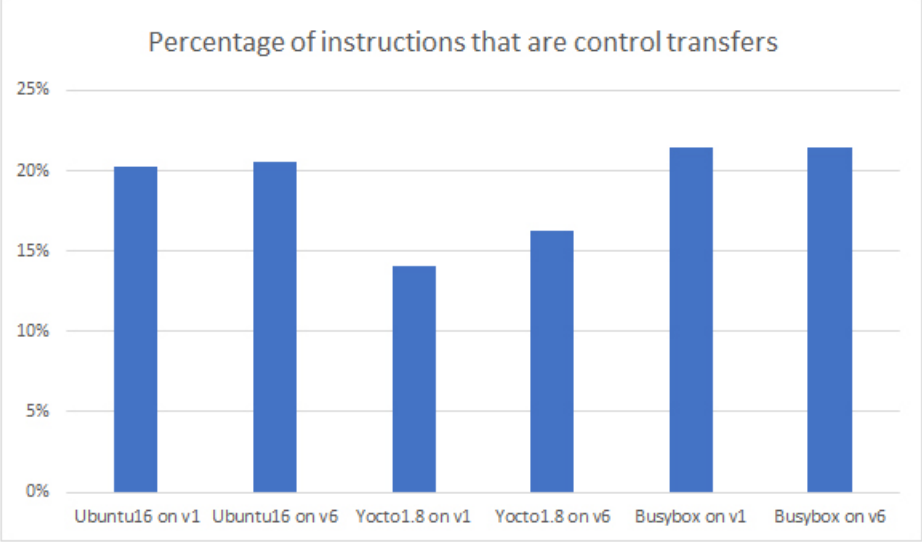
り、それは時間の経過とともに変化し、コンパイラーの各種ターゲットマシン向けの最適化設定によっても異なります。

データでは、そのいくつかを見ることができます。Yocto* セットアップでは、(ADX および BMI2 命令セットの) ADCX、MULX、および ADOX 命令を使用しています。これはまた、ソフトウェアが新しい命令を採用するスピードも示しています。これらの命令は、Yocto* Linux* カーネルと同時期にリリースされた第 5 世代インテル® Core™ i7 プロセッサーで追加されました。プロセッサーがリリースされたときには、ソフトウェアはすでにそのプロセッサーをサポートしていました。通常、新しい命令の仕様は、ハードウェア実装よりもだいぶ前に公開されるため (こちらの論文 (英語) の概要を参照)、ソフトウェア・サポートを早期に追加することができます (ハードウェアがリリースされたときに問題なく動作するように、将来のハードウェアをモデル化した仮想プラットフォームでテストするのが一般的です)。

しかし、新しい Ubuntu* 16.04 セットアップでは、ADX および BMI2 命令が使用されていません。これは、Ubuntu* 16.04 が異なる方法でビルドされていることを示します。可能性として、使用されたコンパイラー・バージョン、コンパイラー・オプション、カーネルフラグ、ディスクイメージに含まれる特定のパッケージなどが考えられます。

制御移行

私は、一般的な制御移行命令についてもチェックしました。ヘネシー & パターソン著『コンピュータアーキテクチャ』の古典的な経験則では、6 命令中 1 つはジャンプです。しかし、ここで使用した Linux* コードベースでは、それよりも頻度が高く、5 命令中 1 つが制御移行命令でした。ただし、Yocto* スタックでは 6 命令中 1 つでした。これも、想定していなかった多様性です。



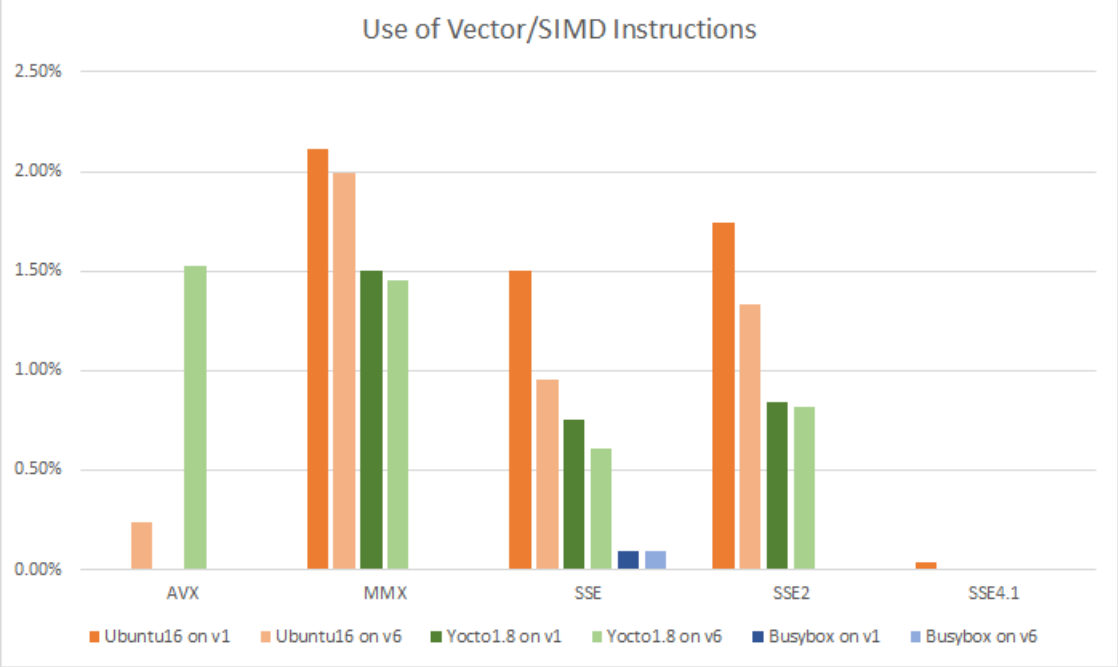
A simple and easy thing to find out using a virtual platform.

ベクトル命令

新しい命令セットの最も良く知られているカテゴリーは、SIMD (Single-Instruction Multiple-Data) またはベクトル命令でしょう。SIMD 命令は、1997 年にインテル® MMX® テクノロジー対応インテル® Pentium® プロセッサーでインテル® MMX® 命令セットがリリースされて以来、導入されています。今日では、当たり前のように使用されています。上記の上位 28 位のグラフには示されていませんが、第 29 位にインテル® MMX® 命令の PXOR がランクインしています。インテル® MMX® 命令の後に登場したのがインテル® ストリーミング

SIMD 拡張命令 (インテル® SSE) の各世代で、さらに最近のものではインテル® アドバンスト・ベクトル・エクステンション (インテル® AVX)、インテル® AVX2、およびインテル® AVX-512 があります。

オペレーティング・システムのブートの調査結果から、私は、ベクトル命令はそれほど使用されていないと考えていました。しかし、実行された命令の 5 ～ 6% がベクトル命令でした。各ベクトル命令がどの命令セット拡張のものかを調べ、命令セット拡張ごとに命令をグループ化したものが以下のグラフです。



最初に、BusyBox ではベクトル命令をほとんど使用していないことが分かります。次に、v1 と v6 プロセッサーを比較すると、v6 では古い命令セットの使用が減り、新しい命令セットの使用が増えています。特に、古いインテル® SSE 命令セットからインテル® AVX への移行が見られます。第 6 世代インテル® Core™ i7 プロセッサーは、インテル® AVX2 命令をサポートしていますが、これらのソフトウェア・スタックではインテル® AVX2 命令がほとんど使用されていないことが分かります。

Simics* テクノロジー

前述のように、この調査は Simics* 仮想プラットフォームで簡単に行うことができます。Simics* は、ターゲットシステムのすべてのプロセッサーで実行されたすべての命令にアクセスします (テストにはデュアルコア・マシンを使用しましたが、ブート中と OS がアイドル状態の間 2 つ目のコアは何も実行しませんでした)。ブートは、ブートデバイスの選択とブートの最後にターゲットシステムへログインすることを含め完全に自動化されており、手動での介入は不要です。

各テストは、複数回実行しても同じ結果になるため、1 回のみ実行しました (ここでは、毎回同じ位置から開始する、繰り返し可能なシナリオについて検証するため)。意図的に変更を加えない限り、ターゲットシステムのリアルタイム・クロックの設定 (シミュレーションのパラメーターで、外部から渡されない) などを含むすべてが繰り返されます。

まとめ

新しいプロセッサーの新しい命令にソフトウェア・スタックがどのように適合しているか知ることは有意義なことでした。今日では、実行しているハードウェアに応じて、バイナリーを変更することなく異なる動作が可能な、適応性が組込まれたソフトウェアがあります。ここでテストしたすべてのケースでは、2 つの異なるタイプのターゲットシステムで同じソフトウェア・スタックを使用しました。各ケースでは、それぞれのターゲットシステムで利用可能な命令に応じて、異なる命令が使用されることが分かりました (ソフトウェア・スタックが古すぎて、新しいハードウェア機能を認識できないケースを除く)。この調査は、シミュレーターでは簡単にできても、ハードウェアでは面倒なデータ収集の例です。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。