Vol.2022-ARC-248 No.4
Vol.2022-SLDM-198 No.4
Vol.2022-EMB-59 No.4
2022/3/10

プロセッサ・シミュレータ「鬼斬弐」を基にしたマルチスレッドシミュレータの開発

中村 朋生[1,a)]　入江 英嗣[1]　坂井 修一[1]

概要：プロセッサの研究において、性能評価の簡易さや測定速度を重視して、しばしばプロセッサシミュレータが用いられる。プロセッサの内の各モジュールが詳細に実装されているシミュレータを用いれば、より実環境に近い評価を行うことができる。現在、国内コミュニティで盛んに開発されているプロセッサシミュレータ「鬼斬弐」は、各パイプラインステージやマイクロアーキテクチャモジュールが詳細に実装されており、サイクルアキュレートなシミュレーションが可能である。一方で、「鬼斬弐」はシングルスレッドプログラムにのみ対応しており、実行中にスレッド生成、同期をするプログラムを用いた評価が困難である。しかし、メディア処理や大規模な演算を用いるベンチマークプログラムは、マルチスレッド実行を前提として記述されていることが多い。そこで、我々はプロセッサシミュレータ「鬼斬弐」をベースとして、マルチスレッドプログラムを評価できるように拡張した。拡張したシミュレータは、C 言語の標準ライブラリである pthread.h に従って記述されたマルチスレッドプログラムを実行できる。加えて、マルチスレッドの動作を確認するための簡単なサンプルプログラムとベンチマークプログラムを作成した。これらのプログラムの実行を通じて、開発したシミュレータ上でスレッド生成と同期が正しく行われ、マルチスレッド化に伴うパフォーマンス向上の再現を確認した。本稿で述べた拡張は、鬼斬弐の github レポジトリに push する予定である。

1. はじめに

プロセッサの研究において、新たな命令セットアーキテクチャ [1] や分岐予測器 [2], [3]、プリフェッチャ [4] といったマイクロアーキテクチャを評価する際に、しばしばプロセッサシミュレータが用いられる。プロセッサシミュレータには、プログラムのトレースを入力とする簡単なもの [5], [6] から、詳細なパイプラインステージをもシミュレートできるもの [7], [8], [9] まで開発されている。その中でも、プロセッサシミュレータ「鬼斬弐」[7], [10] は、各パイプラインステージやマイクロアーキテクチャモジュールが詳細に実装されており、投機的な命令発行からリプレイに至るまでサイクルアキュレイトにシミュレートすることができる。加えて、国内コミュニティによる開発とメンテナンスが現在でも盛んである。

メディア処理や大規模な演算を用いるプログラムを題材としたプロセッサの研究において、評価に用いるプロセッサシミュレータはマルチスレッド環境をサポートしていることが望ましい。これらのプログラムでは大規模な並列化やマルチスレッドによるキューイング処理などの効果が高く、マルチスレッド実行を前提として記述されていることが多いからである。そのため、マルチスレッド環境がサポートされたプロセッサシミュレータであれば、これらの処理をシングルスレッド化したシミュレーションに対して、より実環境に近い評価を行うことができる。しかし、詳細なマイクロアーキテクチャが実装されている「鬼斬弐」には、実行中にスレッド生成、同期をする機能が実装されておらず、それらのプログラムの詳細な評価を行うことが困難である。

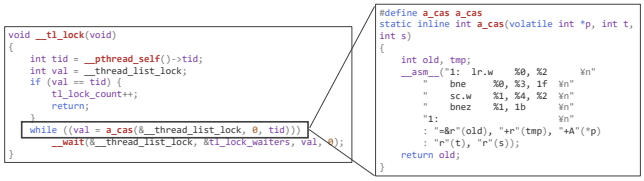
そこで、我々はプロセッサシミュレータ「鬼斬弐」をベースとして、マルチスレッドプログラムを評価できるシミュレータを開発した。開発したシミュレータは、C 言語の標準ライブラリである pthread.h で定義されたスレッドの生成と同期に関する基本的な関数の実行をサポートする。我々は、pthread.h の関数を用いた 3 つのサンプルプログラムを記述し実行することで、シミュレータが正しく動作することを確認した。加えて、マルチスレッド環境を前提としたベンチマークプログラムの例として、マージソートと行列ベクトル積を行うプログラムを記述し、マルチスレッド化に伴うパフォーマンス向上の再現を確認した。

以降、マルチスレッド実行をサポートするため、鬼斬弐に加えた実装と、開発したシミュレータによるマルチス

[1] 東京大学 大学院情報理工学系研究科
[a)] tomokin@mtl.t.u-tokyo.ac.jp

ⓒ 2022 Information Processing Society of Japan

Vol.2022-ARC-248 No.4
Vol.2022-SLDM-198 No.4
Vol.2022-EMB-59 No.4
2022/3/10



```
void __tl_lock(void)
{
    int tid = __pthread_self()->tid;
    int val = __thread_list_lock;
    if (val == tid) {
        tl_lock_count++;
        return;
    }
    while ((val = a_cas(&__thread_list_lock, 0, tid)))
        __wait(&__thread_list_lock, &tl_lock_waiters, val, 0);
}
```

```
#define a_cas a_cas
static inline int a_cas(volatile int *p, int t,
int s)
{
    int old, tmp;
    __asm__("1:  lr.w    %0, %2      ¥n"
        "      bne     %0, %3, 1f  ¥n"
        "      sc.w    %1, %4, %2  ¥n"
        "      bnez    %1, 1b      ¥n"
        "1:                        ¥n"
        : "=&r"(old), "+r"(tmp), "+A"(*p)
        : "r"(t), "r"(s));
    return old;
}
```

図 1　riscv-musl [11] 上 pthread.h における不可分操作

レッドプログラムの実行・解析結果を述べる。

2. 実装

C 言語の pthread.h を用いたマルチスレッドプログラミングをする際に、最低限サポートすべき関数は、スレッドの生成を実行する pthread_create() 、 スレッド間の同期を行う pthread_join()、メモリのロックと解放を行う pthread_mutex_trylock()、pthread_mutex_unlock() である。プロセッサシミュレータ上で、上記の関数を動作させるためには、命令セットアーキテクチャの命令ニーモニックに加え、いくつかのシステムコールを処理する必要がある。鬼斬弐では、プログラムの実行に必要な多くのシステムコールがすでに実装されいる。そのため、新たに実装する必要のあったシステムコールは、clone と futex の 2 つであった。

システムコールに加えて、上記の pthread.h の関数の実行には、不可分操作をサポートする必要がある。pthread.h に頻出する不可分操作の実装例を図 1 に示す。図 1 は、RISC-V の標準ライブラリ実装の 1 つである riscv-musl [11] から抜粋した。riscv-musl 上では、load reserve 命令と store conditional 命令によって実装された Compare and Swap (cas) 関数によって不可分操作が実現されている。鬼斬弐では、load reserve 命令と store conditional 命令のデコードがサポートされているが、通常の load ・store 命令と等価に扱われている。これは、シングルスレッドプログラムを実行する場合、両者の命令は同じ振る舞いをするためである。そこで、これら load reserve 命令、store conditional 命令を処理するよう追加の実装を行った。

なお、マルチスレッド環境をサポートする実装では、広く研究されている命令セットアーキテクチャである RISC-V [12] 環境を前提とした。また、簡単のため、スレッドの割り当ては、Simultaneous Multi-Threading (SMT) 幅までとした。以降、2 つのシステムコールの実装方式と、load reserve 命令、store conditional 命令の実装方式について説明する。

2.1 システムコール

2.1.1 clone

risc-musl [11] を含む多くの標準ライブラリにおいて、pthread_create() によるスレッド生成は、システムコールの clone によって実現される (図 2)。clone を処理すべく

```
int __pthread_create(pthread_t *restrict res, const pthread_attr_t *restrict attrp,
void *(*entry)(void *), void *restrict arg)
{

    /* *** 中略 *** */

    __tl_lock();
    libc.threads_minus_1++;
    ret = __clone((c11 ? start_c11 : start), stack, flags, args, &new->tid, TP_ADJ(new),
&__thread_list_lock);

   /* If clone succeeded, new thread must be linked on the thread
    * list before unlocking it, even if scheduling may still fail. */
    if (ret >= 0) {
        new->next = self->next;
        new->prev = self;
        new->next->prev = new;
        new->prev->next = new;
    }
    __tl_unlock();
    __restore_sigs(&set);
    __release_ptc();

    /* *** 中略 *** */
```

図 2　riscv-musl [11] 上、pthread_create() 関数の一部

鬼斬弐内の システムコールを処理する部分に、以下の処理を追加実装した。

(1) システムコール番号が clone だと判定される。
(2) 新しく生成するスレッドのために、命令セットアーキテクチャ の定める論理レジスタ数分、フリーリストから物理レジスタを確保する。
(3) 親スレッドと各論理レジスタの値が等しくなるように、物理レジスタ間でコピーを行う。
(4) clone の引数で指定されているアドレスを、子スレッドの stack pointer を示す物理レジスタに記録する。ただし、clone の引数によっては、親スレッドと stack pointer を共有する場合もある。
(5) 親スレッドの clone の返り値処理を行う。具体的には、返り値を格納すると規約で定めれた論理レジスタにプロセス id を格納する
(6) 子スレッドの clone の返り値処理を行う。子スレッドの生成が正しく成功した際の返り値である 0 を規約で定められた論理レジスタに格納する。

2.1.2 futex

前述した pthread.h の関数では、スレッドの同期処理はシステムコール futex を利用して実現される。pthread_join() のメインループを図 3 に示す。この関数では、futex を利用してスレッドの wait を実現している。futex の処理は引数に応じて複数あるが、前述した関数内では wait と wake フラグのみを用いてコールされていたので、これら 2 つの処理を実装した。

wait：wait フラグをつけて futex をコールしたスレッドを停止する。停止したスレッドは、wake されるまで新たな命令がフェッチされなくなる。wake されるときのため、引数で指定されたアドレスとコールしたスレッドの id のペアを記録しておく。無効なアドレスを指定している場合や、すべてのスレッドが wait してしまうようなコールがある場合は、エラーを返す。

wake：引数で指定されたアドレスで wait しているスレッドを wake する。wake されたスレッドは、次のサイクルから新たな命令がフェッチされる。指定されたアドレスで

ⓒ 2022 Information Processing Society of Japan

Vol.2022-ARC-248 No.4
Vol.2022-SLDM-198 No.4
Vol.2022-EMB-59 No.4
2022/3/10

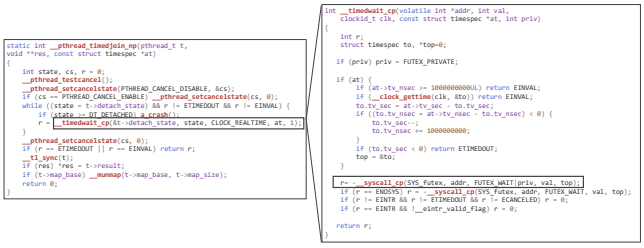


図 3 riscv-musl [11] 上、pthread_join() 関数の一部

wait しているスレッドが記録されていない場合は、何も処理しない。返り値として、wake したスレッド数を返す。

2.2 load reserve 命令と store conditional 命令

多くの命令セットアーキテクチャと同じく RISC-V [12] での load reserve 命令と store conditonal 命令の仕様は以下である。まず load reserve 命令は指定されたアドレスの値を返す。その後、store conditional 命令によって同じアドレスにメモリ書き込みを行う。store conditional 命令による書き込みの前に、そのアドレスになんらかの書き込みが発生していた場合、store conditional 命令は失敗する。

これを実現するため、鬼斬弐内でメモリを管理する VirtualMemory クラスに以下の関数を追加実装した。

- SetReservedAddress(uint64_t addr) : addr で指定したアドレスを監視対象のアドレス群に追加する。
- CheckReservedAddress(uint64_t addr) : addr で指定したアドレスに対して、store conditional 命令以外のメモリアクセスが生じているかを確認する。
- ReleaseReservedAddress(uint64_t addr) : addr で指定したアドレスを監視対象のアドレス群から解放する。

load reserve 命令によるメモリアクセスが発生した際、SetReservedAddress() がコールされ、そのアドレスは監視対象に追加される。store conditonal 命令によるメモリアクセスが発生した際、CheckReservedAddress() によって命令の返り値が決定される。その後、ReleaseReservedAddress() がコールされ、そのアドレスは監視対象から解放される。

上記の関数に加えて、監視対象のアドレスと、そのアドレスにアクセスがあったか否かを記録するフラグの可変長配列変数を追加した。任意のメモリアクセスが発生した際には、監視対象のアドレス群にアクセスするかを判定し、アクセスする場合はフラグをセットする。

3. 評価

3.1 評価環境

上述したシステムコールおよびに、load reserve 命令、store conditional 命令の処理を鬼斬弐 に実装し、マルチスレッドコードを含むプログラムを正しく実行できるかの評価を行った。実行するプログラムとして、C 言語の

表 1 評価で用いたプロセッサシミュレータのコンフィグ

Modules	Parameter
Fetch Width	6 way
Fetch Policy	RoundRobin / icount
SMT Width	3 way
Scheduler Size	256 entries
Branch Predictor	gshare

Memory	Capacity	Cache line Size	associativity	Latency	Prefetcher
L1I Cache	32 KiB	64B	8-way	5 cycles	NextLine
L1D Cache	32 KiB	64B	8-way	4 cycles	NextLine
L2 Cache	256 KiB	64B	8-way	8 cycles	N/A
L3 Cache	2 MiB	64B	16-way	30 cycles	Stream
Main Memory	Infinite	-	-	200 cycles	N/A

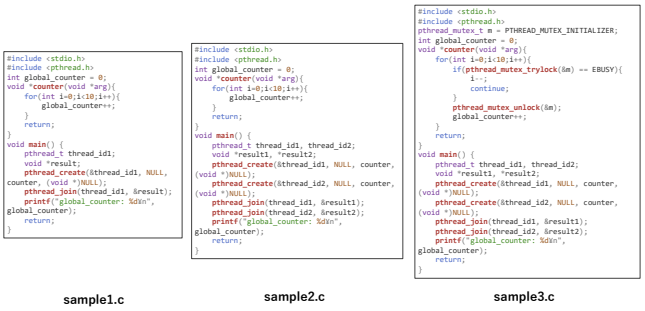


図 4 pthread.h を用いて記述したサンプルプログラム

pthread.h に従った 3 つのサンプルプログラムと 2 つのベンチマークプログラムを記述し、用いた。

サンプルプログラムは、スレッドの生成と実行、同期が正しく動作するか確認するものである。詳細を以下に示す (図 4)。

- sample1.c : グローバル変数をインクリメントするスレッドを 1 つ生成して実行
- sample2.c : グローバル変数をインクリメントするスレッドを 2 つ生成して実行
- sample3.c : グローバル変数をインクリメントするスレッドを 2 つ生成し、メモリロックをしつつ実行

ベンチマークプログラムは、マルチスレッド化による性能向上を確認するものである。詳細を以下に示す。

- merge_sort.c : 与えられた配列をマージソートする。マルチスレッド時は、配列をスレッド数で分割してソートし、最後に単一のスレッドがマージを行う。
- matrix_vector_mul.c : 与えられた行列とベクトルの積を求める。マルチスレッド時は、行列をスレッド数で分割する。

鬼斬弐で実行する際には、RISC-V クロスコンパイラ [13] を用いて上記のプログラムをコンパイルした。コンパイル時の最適化オプションとして O2 を用いた。また、標準ライブラリとして riscv-musl [11] を利用した。シミュレーションでは Intel Sunny Cove [14] をベースとして決定したパラメータを用いた。それらの詳細を表 1 に示す．ベンチマークプログラムの実行に際して、マルチスレッド

ⓒ 2022 Information Processing Society of Japan

Vol.2022-ARC-248 No.4
Vol.2022-SLDM-198 No.4
Vol.2022-EMB-59 No.4
2022/3/10

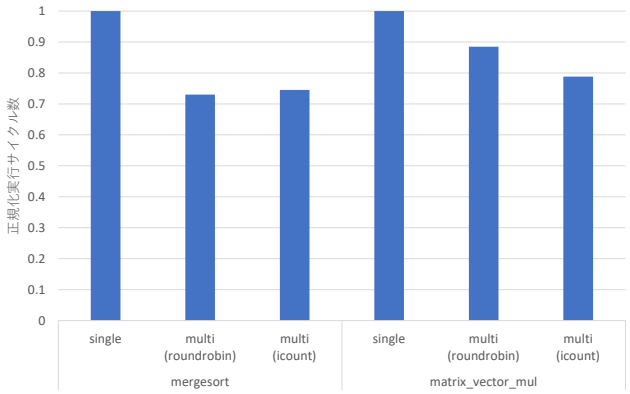


図 5 ベンチマークプログラムの実行サイクル数。シングルスレッド実行時の値で正規化している。

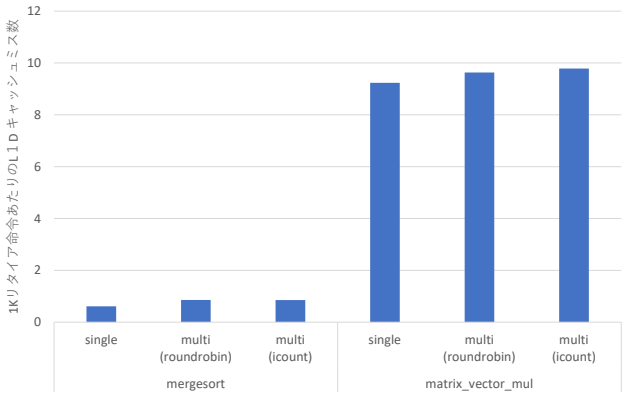


図 6 ベンチマークプログラムにおける 1K リタイア命令あたりの L1D キャッシュミス数。

化の効果をより詳細に確認するため、フェッチポリシーには RoundRobin と icount [15] を用いた。RoundRobin は、フェッチを行うスレッドを順番に切り替えるナイーブなポリシーである。icount は、フェッチした命令のが少ないスレッドが優先的にフェッチを行うポリシーである。

3.2 評価結果

サンプルプログラム sample1.c、sample2.c、sample3.c に関して、鬼斬弐 がエラーを出力せずに正常終了することを確認した。また、sample2.c と sample3.c を比較することで、正しく同期処理が行えていることを確認した。同様に、ベンチマークプログラムに関して、鬼斬弐がエラーを出力せずに正常終了することを確認した。なお、実行結果の相違がないことも確認した。

ベンチマークプログラムにおけるマルチスレッド化によるパフォーマンスの変化を図 5 に示す。merge_sort.c の場合、シングルスレッド実行に対して マルチスレッド (RoundRobin) 実行によって 37%、マルチスレッド (icount) 実行によって 34% 実行サイクル数の削減が確認できた。matrix_vector_mul.c の場合、シングルスレッド実行に対して マルチスレッド (RoundRobin) 実行によって 12%、マルチスレッド (icount) 実行によって 25% 実行サイクル数の削減が確認できた。

マルチスレッド化による高速化の要因として、予測ミスによるリカバリ回数、およびにリカバリされた命令数を計測した。その結果を、図 7、図 8 に示す。図 7 より、どちらのベンチマークプログラムにおいても、リカバリの発生する回数自体は、マルチスレッド化してもあまり変化しないことがわかる。ここで、merge_sort.c でのリカバリは分岐予測ミスに由来するもの、matrix_vector_mul.c でのリカバリはレイテンシ予測ミスに由来するものが大半である。一方、図 8 より、リカバリされた命令数はマルチスレッドによって大きく削減されていることがわかる。

上記の計測に加え、マルチスレッド化が及ぼしうる悪影響であるキャッシュ競合についての計測を行った。その結果を図 6 に示す。どちらのベンチマークプログラムにおいても、L1D キャッシュミス数は、マルチスレッド化によって多少増加している。

これまでの数値評価に加えて、パイプライン可視化ツール Konata [16] を利用し、マルチスレッド化による効果の可視化を行った。その結果を 図 9 に示す。どちらのベンチマークプログラムにおいても、2 つのスレッドが並行して命令をフェッチし、実行していることがわかる。マルチスレッド化した merge_sort.c の場合、パイプライン上にある分岐命令に後続する命令数が減り、予測ミスによってフラッシュされる命令数が減少していることがわかる。マルチスレッド化した matrix_vector_mul.c の場合、長いレイテンシがかかるメモリアクセスがオーバーラップされ、単位サイクルあたりに実行される命令数が増加していることがわかる。

4. おわりに

本稿では、プロセッサシミュレータ「鬼斬弐」を拡張してマルチスレッドシミュレータを開発した。開発したシミュレータは、C 言語の標準ライブラリである pthread.h に従って記述されたマルチスレッドプログラムを実行できる。簡単なサンプルプログラムを記述し、実行することで正常に動作することを確認した。加えて、マルチスレッドを利用したベンチマークプログラムを記述し実行することで、マルチスレッド化によるパフォーマンスの向上が確認できた。本稿で述べた拡張は、鬼斬弐の github レポジトリに push する予定である。今後は、マルチスレッド環境が前提となるベンチマークプログラムを題材とする近似計算プロセッサの研究において、本シミュレータをベースとした評価を行う。

謝辞 本研究の一部は、JST さきがけ JPMJPR20M1、JSPS 科研費 JP21J11687 による。

ⓒ 2022 Information Processing Society of Japan

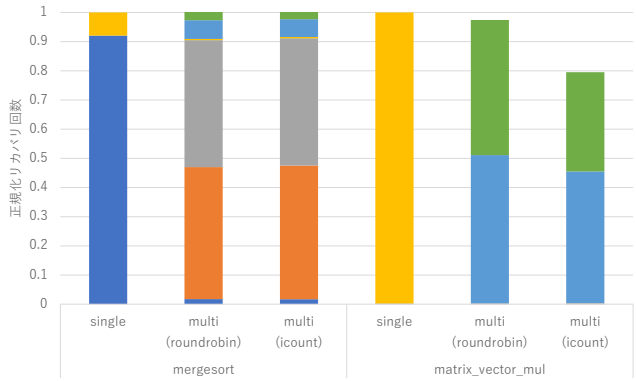


図 7 ベンチマークプログラムのリカバリ回数。
シングルスレッド実行時の値で正規化している。
凡例は図 8 と同様。

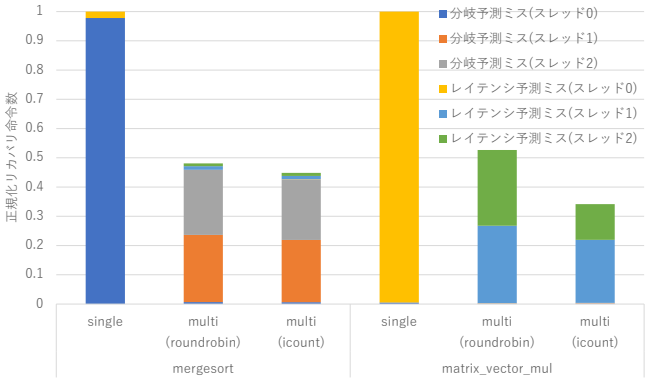


図 8 ベンチマークプログラムのリカバリされた命令数。
シングルスレッド実行時の値で正規化している。

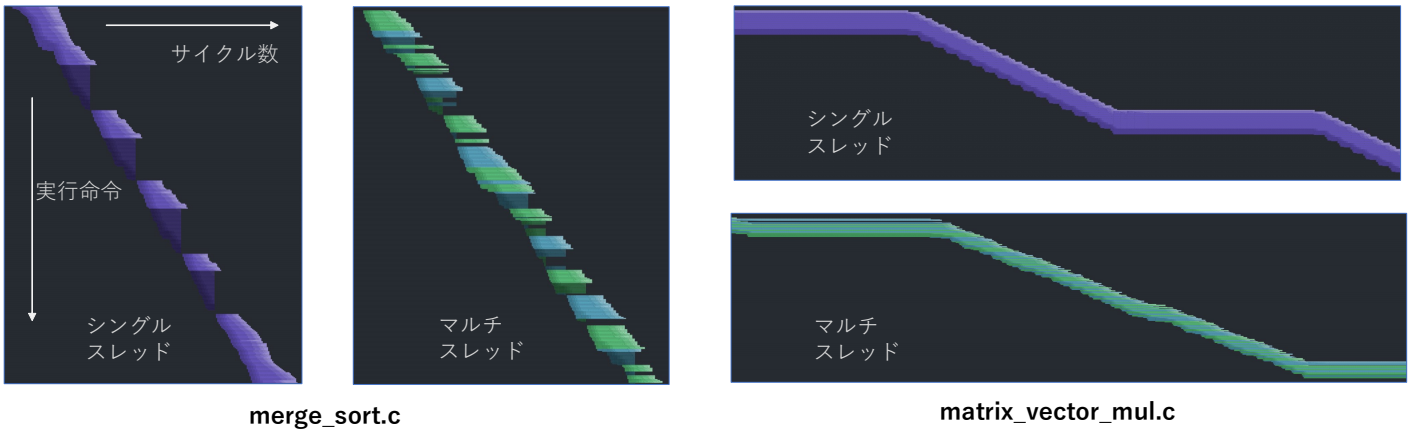


図 9 Konata [16] によるパイプライン可視化図。縦軸が実行される命令を表し、横軸が命令の実行にかかるサイクル数を表す。各行の先頭位置はその命令がフェッチされた時点の実行サイクル数を表し、末尾はリタイアした時点での実行サイクル数を表す。各バーはその命令をフェッチしたスレッド id に応じた色に着色している。半透明のバーはフラッシュされた命令であることを示す。

参考文献

[1] Irie, H., Koizumi, T., Fukuda, A., Akaki, S., Nakae, S., Bessho, Y., Shioya, R., Notsu, T., Yoda, K., Ishihara, T. et al.: Straight: Hazardless processor architecture without register renaming, *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 121–133 (2018).

[2] Seznec, A.: A new case for the tage branch predictor, *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 117–127 (2011).

[3] Seznec, A. and Michaud, P.: A case for (partially) TAgged GEometric history length branch prediction, *The Journal of Instruction-Level Parallelism*, Vol. 8, p. 23 (2006).

[4] Nakamura, T., Koizumi, T., Degawa, Y., Irie, H., Sakai, S. and Shioya, R.: D-JOLT: Distant Jolt Prefetcher, *The 1st Instruction Prefetching Championship (IPC1)* (2020).

[5] Champsim: https://github.com/ChampSim/ChampSim/.

[6] Championship Value Prediction (CVP) simulator. : https://github.com/eric-rotenberg/CVP/tree/cvp2v2.2.

[7] 塩谷亮太，五島正裕，坂井修一：プロセッサ・シミュレータ「鬼斬弐」の設計と実装，先進的計算基盤システムシンポジウム SACSIS，Vol. 2009, No. 4, pp. 120–121 (2009).

[8] gem5 : The gem5 simulator system: https://www.gem5.org/.

[9] Lowe-Power, J., Ahmad, A. M., Akram, A., Alian, M., Amslinger, R., Andreozzi, M., Armejach, A., Asmussen, N., Beckmann, B., Bharadwaj, S. et al.: The gem5 simulator: Version 20.0+, *arXiv preprint arXiv:2007.03152* (2020).

[10] Onikiri 2: https://github.com/onikiri/onikiri2.

[11] riscv-musl: https://github.com/richfelker/musl-cross-make.

[12] RISC-V: The Free and Open RISC Instruction Set Architecture: https://riscv.org/.

[13] riscv-gnu-toolchain: https://github.com/riscv/riscv-gnu-toolchain.

[14] Doweck, J., Kao, W.-F., Lu, A. K.-y., Mandelblat, J., Rahatekar, A., Rappoport, L., Rotem, E., Yasin, A. and Yoaz, A.: Inside 6th-generation intel core: New microarchitecture code-named skylake, *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Vol. 37, No. 2, pp. 52–62 (2017).

[15] Tullsen, D. M., Eggers, S. J., Emer, J. S., Levy, H. M., Lo, J. L. and Stamm, R. L.: Exploiting choice: Instruction fetch and issue on an implementable simultaneous

ⓒ 2022 Information Processing Society of Japan

multithreading processor, *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 191–202 (1996).
[16] Konata: https://github.com/shioyadan/Konata.

ⓒ 2022 Information Processing Society of Japan