Bare Demo of IEEEtran.cls for Computer Society Journals

Yusuke Fujii, *Member, IEEE,* Takuya Azumi, *Member, IEEE,*
Nobuhiko Nishio, *Member, IEEE,* Tuyoshi Hamada, *Membed, IEEE,*
Shinpei Kato, *Member, IEEE,*

Abstract—The abstract goes here.

Keywords—Computer Society, IEEEtran, journal, LaTeX, paper, template.

1 INTRODUCTION

GPU は汎目的アプリケーションを加速させる手法として既に一般的になりつつある。GPU は SIMD 型であり、数千ものコアで大量のスレッドを並列処理することで、データ並列性を持ったアプリケーションにかかる時間を劇的に短縮させる。

GPU はその性能から Cyber-Physical System で利用され始めている。Cyber-Physial systems represent next generation networked and embedded systems. These system was tightly coupled with computation and physical elements to control real-world phenomena. There control algorithms are becoming more and more complex, which distinguishes CPS from traditional safety-critical embedded systems in terms of the computational cost. From their factors, CPS requires more high-performance computing resources to achieve real-fast computing.

既に GPU は HBT-EP Tokamak や Automate driving car などの CPS アプリケーションに利用され始めている。HBT-EP Tokamak は核融合の制御に使われるシステムでセンサから取得したプラズマの状態を GPU で処理し、電磁波によって制御している。Automate driving car に使える、障害物検知、Pedestrian Tracking, ルート推定などに使われている。

しかしながら GPU のようなメニーコアやマルチコアに関する技術は未だ成熟しておらず、性能的な発展を見込めるが、ソフトウェア的な、特に資源管理という面から見ると未熟であり多くの問題を抱えている。GPU は顕著にその傾向が見られる。その理由は GPU のソフトウェアはベンダーによって提供されており、そのほとんどがクローズドソースなためである。開発者はより綿密な資源管理を行うために、ソフトウェアの解析から取り組まなければならない。

我々もこれまでに GPU の資源管理として Gdev や RGEM、TimeGraph に取り組んできたが、それらはリバースエンジニアリングによって支えられてきたシステムである。加えて、それらのシステムが利用するデバイスドライバについても、Linux を支える Nouveau Project が管理する Nouveau を利用しており、こちらもリバースエンジニアリングによって成り立っている。

リバースエンジニアリングには、全ての機能の再現が難しいことや、バージョンアップへの追従が負担などの技術的な問題に加え、その方法次第では法律的な問題も抱えている。

Elliott らの提案する GPUSync では資源管理システムを既存ランタイムから分離し、アクセスの調停を行うことで、既存ランタイムが行う資源管理をスルーした上で自身の資源管理を行っている。しかしながら GPUSync は $LITMUS^{RT}$ 上に実装されており、カーネルへの変更を多分に含んでいる。Gdev についても同様にデバイスドライバ自体への変更を必要としている。この変更は OS へパッチを当てることによるインストレーションを

- *Y. Fujii is with the College of Information Science and Engineering, Ritsumeikan University.*
- *T. Azumi is with the Graduate School of Information Science and Engineering, Osaka University.*
- *N. Nishio is with the Graduate School of Engineering Science, Ritsumeikan University.*
- *T. Hamada is with the , Nagasaki University.*
- *S. Kato is with the School of Information Science, Nagoya University.*

必要としている。このパッチを当てる作業は開発者とユーザへ大きな負担を与える。開発者は、常に最新のカーネルのリリースに追い付くために、パッチを維持していく義務がある。しかし Linux はその更新頻度が早く、開発者が最新のカーネルにむけてポーティング作業を完了させる前に、新しいバージョンのリリースが起きることが多い。そのためカーネルの選択について制限される傾向がある。

本問題に対して我々はこれまでリアルタイム拡張 suit として RESCH を提供してきた。RESCH はローダブルカーネルモジュールを利用しており、カーネルの内部関数を利用することで、カーネル自体にパッチを当てること無くリアルタイム拡張可能としている。しかしながら、RESCH は GPU を保持するシステムのようなヘテロジニアスな環境には対応していない。

Contribution: 本論文では、GPU を保持するシステムにおいてユーザーレベル[1]で資源管理可能なことを Explore し、その成果として、GPU 資源管理システムと RESCH を統合したリアルタイムスケジューリング拡張フレームワークとして Linux-RTX を提供する。

我々はまず、GPU 実行を含んだタスクモデルについて解説し、GPU スケジューリングに必要な要件を示す。その後要件を満たすための手法として、Interrupt Intercept を提案し、そして達成した GPU スケジューラを CPU スケジューラである RESCH に統合、CPU-GPU スケジューリングが可能なフレームワークである Linux-RTX を提供する。本論文は最適なスケジューリング機構を提案するものでなく、最適なスケジューリング機構を探索するためのシステムを提案するものである。

Organization This paper rest of 本論文は◯章で構成される。次章では、対象とするシステムの説明。3 章では、先行研究を解説し、解決すべき点を列挙し、4 章においてそれらの点の解決について説明する。5 章では、Linux-RTXG を利用した際の各オーバヘッドについて計測し評価を行う。6 章で本 Linux-RTXG の問題点と今後の展望について考察していく。

2 System Model

2.1 Scheduling Discussion

これまでの単純なシングルプロセッサ環境におけるスケジューリングでは、

1. OS にパッチを当てずにの意

2.2 GPU Task Model

具体的には、クリティカルセクションをどの範囲で指定するかによって形式が変わる。本研究ではこのクリティカルセクションをユーザが任意で指定することで

例えば GPUSync ではデータ転送開始から、GPU での処理、GPU までの

2.3 Scheduling

2.4 limitation

3 Design and Implementation

In this section, we present linux-rtxg design and implementation.

Figure **??** shows over-view of linux-rtxg. Linux-rtxg is divided into two components that are loadable kernel module and library. linux-rtxg library is interface of communicate between application and linux-rtxg core component(kernel module). it is using system call that is ioctl.

The part of library included speciall method. it is independent Nvidia interrupt raised method (iNVRM). iNVRM is used only on the nvidia driver. if system use nouveau driver, runtime must use part of gdev. Gdev can happen arbitary interrupt of gpu kernel in the user-space mode, and it have no need to be independent interrupt raised method.

linux-rtxg loadable kernel module is positioned kernel-space. Thus, module can use kernel exported function.

3.1 GPU Scheduler

- scheduling mechanism
- interface

Linux-RTXg's scheduler function is provided RTXG API. The basic APIs supported by Linux-RTXg are listed in Table 1. Some APIs have arguments and others do not. Figure **??** shows a sample program that is using CUDA API. Our API is don't modificated existing CUDA API for supporting proprietary CUDA API.

3.2 GPU synchronization

GPU のアプリケーションには共通する特性がある。それは GPU に処理を発行してから、終了するまでの待機時間が発生することである。待機中

TABLE 1
Basic Linux-RTXg APIs

rtx_gpu_open()	To register itself to linux-RTXg, and create scheduling entity. It will must call first.
rtx_gpu_launch()	To control the GPU kernel launch timing, in other words it is scheduling entry point. It will must call before the CUDA launch API.
rtx_gpu_sync()	To wait for finishing GPU kernel execution by sleeping with TASK UNINTERRUPTIBLE status.
rtx_gpu_notify()	To register the notify command to GPU microcontroller.
rtx_gpu_close()	To release scheduling entity.

に同一タスクで処理を継続する例もあるが、その処理結果は同期しなければ受け取ることができないため、必ず同期時間が発生する。この同期時間はself-suspendingに関連する問題を発生させる。加えて、その同期方法によってはレイテンシが大幅に増加するために、適切な手法によって同期が行われなければならない。

GPUの同期は2つの手法がある。一つはfenceを用いる方法。もう一つは割込みを用いる方法である。GPUには多くのエンジン（マイクロコントローラ）が搭載されている。本論文では詳しいアーキテクチャはメインではないので省略するが、詳細は過去の文献 **[?]**, **[?]** に記載しています。このエンジンにはコンテキスト管理用、コマンド受け取り用、データ転送用などが存在している。通常、コマンド受け取り用コントローラが受け取ったコマンドのヘッダから、そのコマンドを使用するコントローラへと送信し、処理が行われる。この順序はすべてFIFOで行われる。

まずFenceを用いた方式では、まず同期用に仮想アドレス空間にマップされたバッファをGPUメモリに用意する。そして、このメモリに値をEngine経由で書き込む用にコマンドを発行する。すると、カーネル実行とメモリ転送終了後にエンジンが値を書き込むため、CPU側でそのマップされたメモリアドレスをポーリングしながらチェックすれば同期が可能である。割込みを用いる方式についてもEngineの機能を利用する。

タスクはTASK_INTERRUPTIBLE/TASK_UNINTERRUPTIBLEにした上でschedule()を呼ぶか、waitqueueなどを用いて上記に相当する処理を行いsuspendする。そしてEngineから割込みを発生させるコマンドを発行し、割り込みコントローラがそれを獲得、GPUドライバが登録したISRを立ち上げる。ISR内では、各割込みに関するステータスをマッピングされたレジスタから読み込み、各割込みごとに処理を行う。処理後は割込み完了フラグを書き込み、初期化する。

一般的な利用の場合、多くはfenceが用いられるが、Gdevなどはスケジューリングにおいてあるタスク終了後、次のタスクを立ち上げる部分に割込みを用いている。一般的にこれらはCPU側の実装の仕方によって異なる。前者は待機するタスクの状態がTASK_RUNNINGの時に、sched_yield()などを用いて他のタスクへの影響を考慮しながらポーリングする場合に適している。後者は待機するタスクの状態がTASK_INTERRUPTIBLEかTASK_UNINTERRUPTIBLEの時に、割込みというeventによってタスクを立ち上げて処理を継続していく。

3.2 Interrupt interception

前述した割込みはデバイスドライバ（カーネルと共にパッケージされている）によって登録されたISRがハンドルする。Linux-RTXではスケジューリング用のワーカースレッドを立ち上げており、次に実行するタスクの選択が終わってからそのタスクの実行が終わるまでは実行停止状態で待機する。上記SCHED_DEADLINE時のwait queueueを用いた場合に置いても、たすくの　実行が終わり同期されるまで実行停止状態で待機する。これらを適切に立ち上げるためには任意の割込みを獲得し、外部ISRがその割込みがどのカーネルに関連しているかを識別できる仕組みが必要である。加えて、割込みの識別はGPUのステータス・レジスタを読み込んで行う必要があり、GPUドライバが割込みレジスタをリセットする前に、実行される必要がある。

そのため我々は、GPUに関する割込みを傍受（我々の割込みハンドラを先に呼び出し、オリジナルの割り込みハンドラへとシーケンシャルに移行するようにする。Linuxの割込みは分割割込みを用いており、前半部分をtop-half、後半部分をbottom-halfと呼ぶ。

gllenらの提供するklmirqdはtaskletがリアルタイム性に及ぼす問題について言及し、bottom-halfに位置するtaskletをオーバライドしている。この手法を我々の目的のために適用しようと考えた際、bottom-halfでは我々の割込み傍受はtop-halfをオーバーライドする。

Linuxでは、割込み番号ごとにirq_descという割込みのパラメータを保持する構造体を持ってい

る。この構造体には割り込みハンドラの関数ポインタを含む irq_action という構造体がリストで接続されている。irq_desc はグローバルな領域に確保されており、カーネル空間からであれば誰でも参照可能である。Linux のローダブルカーネルモジュールはカーネル空間で動作しているため、この irq_desc を取得でき、Interrupt handler の関数ポインタも取得可能である。

我々はこの取得した関数ポインタを保持し、我々の傍受用割込みハンドラを設定、コールバック関数を保持している関数ポインタから設定することで、GPU に関する割込みの発生後、先んじて取得が可能である。ただしこの手法は当然のことながら、割込みを遅延させることにほかならないため、オーバヘッドについて綿密な評価を Sec**??**にて示す。

3.2 Independent interrupt

我々のこれまでの実験から、NVIDIA の Closed-source software ではコンテキスト生成時の設定によってはカーネル実行後に割り込みを発生していることがわかった。実際に interrupt interception によって盗聴した結果が**??**である。NVIDIA のドキュメントによると、CUDA はワーカースレッドを立ち上げて、割込みを受け取り、同期を行っている。

しかしながらこの割込みは、我々のアプローチではどのカーネル実行に関連付けされているかが区別できなかった。したがって、ひとつの GPU への複数のアクセスを許可した場合、割込みを利用したスケジューリングが不可能になる。

そのためここでは、ランタイムから独立した割込み機構として、独自に割込みを発生させる仕組みを実装する。NVIDIA のクローズドソースドライバは Nouveau プロジェクトのリバースエンジニアリングによる解析によって、ioctl を使ったインタフェースになっていることがわかっている。Gdev ではこの解析された情報を用いて、NVIDIA のクローズドソースドライバとオープンソースライブラリという掛け合わせで CUDA を実行できる基盤が構築されている。本論文では、この基盤から割込みを発生させる部位のみ抽出し、スケジューリングに用いる。

本手法は大きく 2 つに分かれ、それぞれ Initialize と Notify と呼ぶ。Initialize は、いわゆるコンテキストの生成に値する。Virtual Address Space やコマンド送信に用いる Indirect Buffer の確保、コンテキストオブジェクトの生成などを行う。Notify は Compute エンジンや Copy エンジンに向けて割込み発生のコマンドを送信する。

本アプローチに用いるインタフェースは公式にサポートされていないために、ベンダーによる急な仕様変更には対応できない。しかしながら、これ以外に割込みを発生させるアプローチがなく、クローズドソースを用いた場合の限界であるといえる。

3.3 Scheduler Integration

Linux scheduler have various real-time scheduling policies that were SCHED_DEADLINE, SCHED_FIFO and SCHED_RR. We support all scheduling policies that was implemented by linux. However, synchronization does not work well in a specific scheduling policy. The problem that is synchronization by fence in the SCHED_DEADLINE. It problem is synchronization by fence under the SCHED_DEADLINE. It because, implementation of sched_yield() cede cpu to other tasks, by to set the next deadline to current deadline and to set 0 to budget.

その理由は SCHED_DEADLINE での sched_yield() の実装がデッドラインを次の周期に設定しバジェットを 0 に設定することで優先度を低下させ、他のタスクに CPU を譲っているためである。つまり、sched_yield() を用いたポーリングではその周期内での実行を諦めるため、必ずデッドラインミスとなる。しかしながら、sched_yeild() を利用しない場合、同期待ちの間 CPU を専有してしまう。CPU の専有は非効率であり好ましくない。

そのため linux-rtxg では SCHED_DEADLINE のポリシーの際は割込みによって GPU 処理との同期を行う。つまり、タスクを一旦サスペンドする。しかしサスペンド後、SCHED_DEADLINE ではスリープからの復帰時に以下式 (1) を用いてスケジューリング可能性のシンプルな検証を行う。

$$\frac{Absolute\_Deadline - Current\_Time}{Remaining\_Runtime} > \frac{Relative\_Deadline}{Period} \quad (1)$$

式 (1) が真の時、バジェットが補充され、デッドラインが次の周期に設定される。従って同期時に必ず自らサスペンドすることによってこの検証に引っかかり、デッドラインが更新されてしまう。これは Constand Bandwidth Server の仕様であり、self-suspension を含んだタスクのスケジューリングを考慮していないためである。この self-suspension はスケジューリング可能性についても影響を与えており、リアルタイムマルチコアスケジューリングの困難な問題の一つである。不幸なことだが我々

の知る限り、この Self-Suspension は global リアルタイムアルゴリズムにおいて解決された例は無く、我々もそれを完全に解決する手段は提供することができない。

我々はこの復帰時のチェックに関しては、スリープ中も順調にタスクが実行していると仮定して、スリープしている時間 (=GPU の実行時間) を Remaining_Runtime から引くことで対応した。システム設計者はこれを想定して、runtime のパラメータには CPU execution time + GPU execution time (included data transmission time) を含めて設定しなければならない制約があるため、最適ではない手法ではあるが、カーネルを操作しない手法としてはこれが最善ということで妥協した。

以上の、割込み方式での同期+復帰時のパラメータ調整によって SCHED_DEADLINE 下での GPU 実行を含むタスクをサポートした。その他のスケジューリングポリシーでは fence による同期でも問題は発生しないため、全スケジューリングポリシーをサポートできたといえる。

4 EVALUATION

4.1 Experimental Environment

本論文では次のマシンを用いて評価する。

CPU は Intel Core i7 2600 3.40GHz、4GB*2 のメモリ、GPU は GeForce GTX680 を用いる。Kernel は Linux kernel 3.16.0 を用い、ディストリビューションは Ubuntu 14.04 である。CUDA ランタイムは cuda-6.0 or Gdev、GPU ドライバは NVIDIA の 331.62 を用いる。

より高精度な測定を行うために、ユーザ空間では clock_gettime を用いて直接 TSC レジスタにアクセスして測定する。カーネル空間では sched_clock() を用いて計測する。

4.2 Interrupt intercept overhead

Interrupt intercept のオーバヘッドの測定を行う。本評価では、GPU ドライバは nouveau を用いる。割り込み処理は、各割り込みの種類によって、処理時間が異なり、その分布は一様ではないため、単に測定して平均をとっても比較ができない。そのため各割り込みの種類の判別のために Nouveau を用いて、割り込みの種類が同一のもので、カーネル内の do_IRQ 関数内でハンドラが呼ばれてから終了までの時間を測定しどの程度のオーバヘッドで割り込みの盗聴及び、盗聴した割り込みの識別ができるかどうかを測定する。

Figure **??**は上記設定で測定した結果である。Raw ISR は通常のルーチンで実行される ISR、

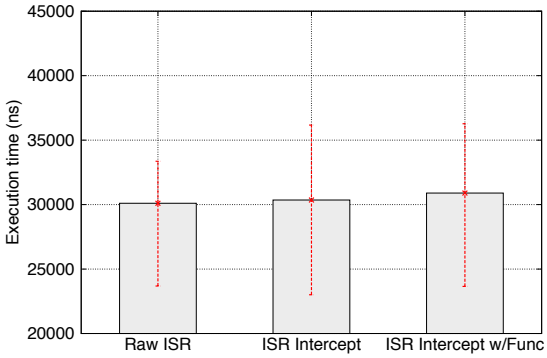


Fig. 1. Interrupt intercept overhead

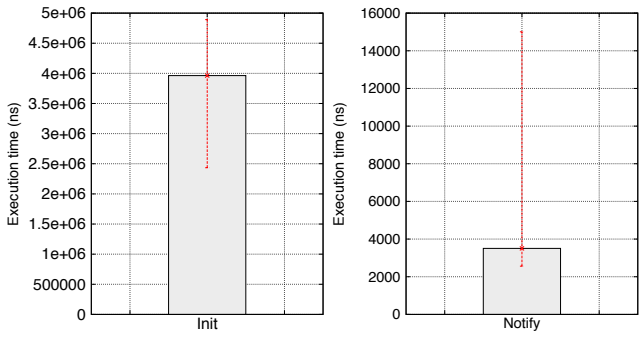
ISR Intercept は割り込みを盗聴するのみ、ISR intercept w/Func は盗聴した上でその割り込みがどの割込みか識別しスケジューラを立ち上げる機能を実行した場合である。それぞれ 1000 回の測定で平均値を取り、最小値と最大値についてエラーバーで示している。この図から見て取れるように、オーバヘッドは確実に存在する。ISR Intercept だと 247ns のオーバヘッドであり、ISR Intercept w/Func でも 790ns のオーバヘッドである。この数値は直感的に考えると小さくシステム自体に影響を及ぼすほどではないと考えられる。しかしその積み重ねによっては影響を与えることは考えなければならない。

4.3 Interrupt raised overhead

本稿では割込み立ち上げのためのオーバヘッドを測定する。割込み立ち上げは 2 つの API の呼び出しを必要とする。一つは cuCtxCreate を呼び出したあとに呼び出す rtx_nvrm_init() である。もう一つは同期したいタイミング (e.g. カーネルラウンチ後) に呼び出す rtx_nvrm_notify() である。スケジューリングを行わない Vanilla な状態ではこれらの API は必要ではないものであるため、これらの API にかかった時間はすべてオーバヘッドとなる。

そのためこれらのオーバヘッドの計測を行う。計測は API の呼び出しから戻るまでを測定する。

結果を Figure 2 に示す。Initialize は Indirect Buffer はプロセスが立ち上がるたびに、コマンド送信用の Indirect Buffer の確保や各エンジンの登録のために呼び出される必要がある。Notify はカーネル実行後や非同期メモリコピー実行後のような実際に割込みを発生させたいタイミングで呼び出される。これらは ioctl システムコールによっ



(a) Part of Initialize (b) Part of notify

Fig. 2. Interrupt raised method overhead

てユーザ空間とカーネル空間をまたいでる影響か、実行時間のバラ付きが大きく出ている。

Initialize は比較的時間がかかっているが、1 プロセスにつき一度しか呼ばれないため、アプリケーション全体への影響は少ないと考えられる。Notify に関してはそれほど時間がかかっておらず、同期待ちの間に実行されるべき処理なため、こちらもアプリケーション全体への影響は少ないと考えられる。

4.4 Overhead

4.5 Compare the prior work

5 Related Work

我々はこれまで、Timegraph[],RGEM[],Gdev[] として GPU の資源管理に関する研究を行ってきた。TimeGraph は GPU に送信されるコマンドをスケジューリングすることで CUDA にかぎらず、OpenGL など、全ての GPU を利用に関する資源管理を行っている。しかしながら GPU のコマンドは処理の実行だけでなく、データ転送、割込み処理登録などの処理時にも送信されており、本当にスケジューリングするべき単位でのスケジューリングには向いていないことがわかっている。そのため、RGEM は GPGPU に特化し、GPU カーネル実行単位でのスケジューリングを目指し、固定優先度でのスケジューリングを実現してる。加えて、データ転送のセグメント分けによってノンプリエンプティブな特性にもたらされるデメリットを最小限にし、レスポンスタイムの向上を目指している。Gdev は RGEM の発展形であり、仮想 GPU と Resource Reservation による QoS 制御や、OS 空間での CUDA 実行などを実現している。加えてデータ転送とカーネル実行をオーバラップさせることで実行時間自体の縮小を実現している。

Elliot et al. present GPUSync[], robust tasklet handling, . GPUSync ではホストから GPU へのデータ転送開始から、GPU での処理、GPU からホストへのデータ転送までをクリティカルセクションと設定し、runtime へのアクセスはクリティカルセクションを区切りとして単一のアクセスとなるように調停を行っている。これによってクローズドソースなランタイムを利用しつつ、自身の GPU 資源管理を実現可能としている。GPUSync はアクセス調停の手法として k-exclusion lock を利用している。加えて各 GPU ごとに Resource Reservation による QoS 担保を行っている。

Han et al. show GPU-SPARC. GPU-Sparc support to automatically split and run the GPU kernel concurrently over multi-GPU,

We show the table2 that is result of comparing the Linux-RTXg and prior work.

6 Conclusion

Acknowledgments

The authors would like to thank...

PLACE PHOTO HERE

Michael Shell Biography text here.

John Doe Biography text here.

Jane Doe Biography text here.

TABLE 2
Linux-RTXg vs prior work

	CPU		GPU		Resource Reservation	Data/Comp.Ovlp.	Multi GPU Aware	Clsd.Src.Compat.	Kernel Free	Configurable
Linux-RTXg										