Bare Demo of IEEEtran.cls for Computer Society Journals

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

Abstract—The abstract goes here.

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

1 Introduction

REAL-TIME system は社会インフラにおいて重要な役割を担っている。Real-time systems はこれまでも、ロボットやマルチメディアシステム、工場システムなど社会に密接に関わるシステムとして、役割を果たしてきた。さらにこれからは、Cyber-physical system という考え方が発達してきており、より社会を支えていくインフラとして役割を担っていくと考えている。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.

One solution for achieving real-fast computing is to use GPUs. GPUs can be realized fast computing for data-intensive application by parallel computing wiht many-core processor. GPUs application developing is become easier since vendor was official supported GPGPU framework that is include programming language and runtime. So GPUs have been rapidly spread.

さらに驚くことに GPU 技術は発展可能性が高い。2013 年に発表され、その性能から話題になった NVIDIA の Geforce GTX TITAN の単精度浮動小数点演算性能は 4.5TFLOPS であるが、2014 年に発表された Geforce GTX TITAN Z は 8T Flops の性能を持っている。この成長率は異常といっても過言ではない。加えて GPU といえば大量のコアが搭載されていることから、直感的には電力消費が激しいと思われている。しかしながら能力あたりの電力消費は従来の CPU に比べてかなり優れており、CPS のニーズを絶妙に満たしているといえ、実際のアプリケーションでも GPU は利用されている。例えば Object detection, pedestrian tracking, navigation and route plannning.

GPU で汎用計算することを General Purpose GPU (GPGPU) と呼ぶ. GPGPU は如何なる環境でも動作するということではなくいくつかの制限がある。ハードウェアが搭載できるかという点は当然として、ベンダーによってライブラリやデバイスドライバなどのランタイムが提供される環境のみで動作可能である。現状ベンダーによって GPGPU がサポートされているのは Windows、Mac OSX、Linux のみである。Linux のメインラインでは SCHED_DEADLINE や PREEMPT_RT のようなリアルタイムなワークロードをサポートするための取り組みが行われている。加えて、Linux の変異体として RTOS として提供されているものも数多くあり、リアルタイムに最も近いと言える。そのため、本論文では Linux を用いる。

我々の知る限り、現段階で最も優れているであろう GPU 資源管理に関する研究は、GPUSync[] と

- *Y. Fujii is with the College of Information Science and Engineering, Ritsumeikan University, Shiga, Kusatsu Noji-higashi, 1-1-1 Japan*
- *T. Azumi is with the Graduate School of Information Science and Engineering, Osaka University,Japan*
- *N. Nishio is with the Graduate School of Engineering Science, Ritsumeikan University, Shiga, Kusatsu Noji-higashi, 1-1-1 Japan*
- *S. Kato is with the School of Information Science, Nagoya University*

Gdevであり、リアルタイムに関してはGPUSyncが一歩先んじている。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、デバイスドライバのISRに変更を加えており、OSへのパッチをあてることでのインストレーションを必要としている。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。

Contribution: 本論文では、GPUを保持するシステムにおいて、リアルタイムなワークロードをサポートするために、ユーザーレベルで可能なことをExploreする。ここでのユーザーレベルとは、OSをいじらずにユーザができることという意である。

我々はこれまでに、カーネルにパッチを当てずに、ユーザレベルでRTOS拡張が実現可能なRESCH[]を提案してきた。RESCHはLinuxの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してスケジューリングすることで、ユーザレベルでのスケジューリング拡張を実現している。

残念なことに、RESCHは一般的なCPUで構成されたシステムを想定として作られており、GPUのようなコプロセッサを追加したヘテロジニアスな環境には対応できておらず、CPSの基盤として用いることができない。

しかしながらGPGPUランタイムはクローズドソースである。

本アプローチはGPUのアクセスの調停や、GPUカーネル終了時の同期のハンドリングを行い、GPUのスケジューリングを可能とした上で、RESCHのアプローチに載せてユーザレベルでGPUのスケジューリングをサポートする。具体的には、

加えて、CPUのみのタスクとGPUが含まれるタスクが混在するシステムにおいて、考えられるスケジューリングの仕方について考察し、それらを取捨選択できる基盤として提供していくことで、今後の研究発展につなげていく。

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 GPU SCHEDULING DISCUSSION

今回ターゲットとするシステムは、一個以上のCPUがあり、一個以上のGPUが搭載された、単一のノードである。その上でCPUのみを使うタスク、GPUを使うタスクが混在し、それぞれ周期タスク、非周期タスクが存在することを想定する。

Gdevは我々が提案してきたオープンソースGPUフレームワークである。リザベーションベースのBANDスケジューリングを行うことでアイソレーションを確保し、QoSの担保を行っている。BANDスケジューラは仮想GPUという形でリソースを分割しており、タスクはそれぞれの仮想GPUに所属してそのリソースを利用していく。仮想GPU内ではFixed-Priorityによってスケジュールされる階層的スケジューラの形をとっている。Gdevはスケジューリングを GPUへのアクセス開始から、同期するまでとしており、CPUの動作に一切関与していない。Gdev's scheduling target is between the starting of GPU access and the synchronization. It is not involved at all with the operation of CPU.

GPUSyncは、Elliot et al. が提案するリアルタイムGPUフレームワークである。GPUSyncではGPUのスケジューリングではなく、GPUが含まれたタスクとしてスケジューリングを行っており、評価では、Clustered-EDFを用いたことを示している。GPUへのアクセスについてはBudget Enforcement方式

2.1 limitation

3 LINUX-RTXG

3.1 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本題ではないので省略する。詳細は過去の文献 [?], [?] に記載しています。このエンジンには COMPUTE や COPY 用のものがある。これらのエンジンにはコマンドバッファに格納されたコマンドを FIFO で処理していく。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/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。

Linux scheduler have various real-time scheduling policies that were SCHED_DEADLINE, SCHED_FIFO, 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。

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し、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。

3.3 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。

4 EVALUATION

4.1 Experimental Environment

本論文では2台のマシンを用いて評価する。

1台目はIntel Core i7 2600 3.40GHzのCPUで、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。

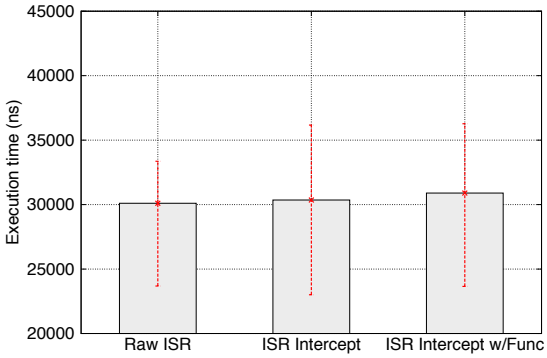
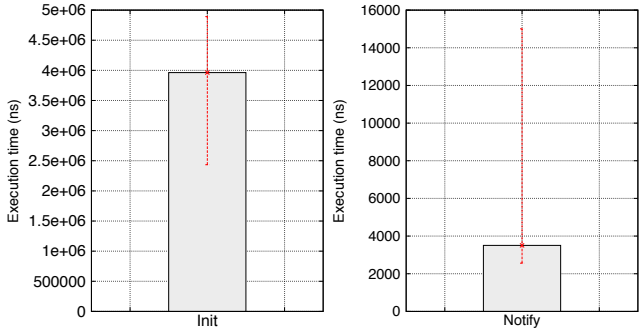


Fig. 1. Interrupt intercept overhead



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

Fig. 2. Interrupt raised method overhead

2 台目は Intel Core i7 3770 3.40GHz の CPU で、4GB*2 のメモリ、GPU は GeForce GTX480 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 Nouveau を用いる。

基本的には NVIDIA ドライバがインストールされる 1 台目を用いるが、Nouveau ドライバを用いる必要性のある評価でのみ 2 台目を用いる。

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、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

APPENDIX A
PROOF OF THE FIRST ZONKLAR EQUATION

Appendix one text goes here.

APPENDIX B

Appendix two text goes here.

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.