2008－OS－109 (7)
2008／8／7

LinuxのI/Oメモリ管理ユニットの性能評価

藤田 智成† 盛合 敏†

Input/Output Memory Management Unit (IOMMU) は，ホストの物理メモリとI/Oデバイスの間に位置し，DMAトランザクションで用いられるアドレスを変換するハードウェア機能である．IOMMUは，I/Oデバイスの物理メモリへの不正なDMAアクセスを防ぐことで，システムの信頼性を向上させる，などの利点を持つが，その一方，アドレス変換のオーバーヘッドがI/O性能を低下させるという欠点も指摘されている．本稿では，Intel社のVirtualization Technology for Directed I/O Architectureと呼ばれるIOMMUが，LinuxのI/O性能へ与える影響を報告する．IOMMUを使用した場合，I/O性能が20%低下したが，Linuxカーネルに実装されているIOMMU制御プログラムを変更することで，性能低下を13%まで抑えることができた．

IOMMU Performance Evaluation

TOMONORI FUJITA † and SATOSHI MORIAI†

Input/Output Memory Management Unit (IOMMU), hardware placed on physical memory between I/O devices, translates address used in DMA transactions. IOMMU can provide valuable features such as protecting physical memory from illegal access by I/O devices, however, it also can impose a performance penalty due to the address translation. This paper presents the evaluation of Linux I/O performance with Intel's IOMMU, called Virtualization Technology for Directed I/O Architecture. With the IOMMU enabled, our benchmarks show a I/O performance decrease of 20 percent on Linux. Our changes to the IOMMU control software in a Linux kernel improved the performance, resulted in a 13 percent decrease.

1. はじめに

Input/Output Memory Management Unit (IOMMU) は，ホストの物理メモリとI/Oデバイスの間に位置し，DMAトランザクション中に，I/Oデバイスが使うアドレス（以後，DMA仮想アドレスと呼ぶ）を変換するハードウェア機能である．

IOMMUが導入された元々の理由は，I/Oデバイスが，物理メモリ全体をアドレス指定することができない問題を解決するためである．IOMMUは，I/OデバイスのDMA仮想アドレスへのアクセスを，任意の物理アドレスへのアクセスに変換する機能を持つ．I/Oデバイスが，アドレス指定できる範囲のDMA仮想アドレスにアクセスすると，IOMMUは，I/Oデバイスがアドレス指定できない高位のアドレスの物理メモリへのアクセスに変換する．IOMMUが利用できない場合，I/Oデバイスは，一時的に確保した，アドレス指定できる範囲の低位のアドレスのメモリに対してDMAを実行し，高位のアドレスとの間でメモリコピー操作を実行しなければならない．

IOMMUは，システムの安定性を向上させるという利点も持っている．オペレーティングシステム（OS）は，IOMMUを使うことで，I/Oデバイスがアクセス可能な物理メモリを限定することができるため，デバイスドライバの誤動作などによる，物理メモリへの不正なDMAアクセスを防ぐことができる．

これまで，IOMMUが搭載される計算機は，主に，Sun Microsystems社のSPARCやIBM社のPOWERプロセッサーなどを使ったハイエンドのワークステーションであった．しかし，近年の仮想化技術[8][3]の需要の高まりにより，I/Oの仮想化を支援する機能として，安価な汎用のx86アーキテクチャにおいても，IOMMUが搭載されるようになってきている[6][2]．

IOMMUは，その利点ゆえに広く使われるようになってきているが，アドレス変換のオーバヘッドのため，I/O性能が低下するという欠点を持っている．近年，メモリベースのディスクドライブであるSolid State Drive （SSD）や10 Gigabit Ethernetなどの出現により，OSに求められるI/O性能は増加する一方であるが，IOMMUに関する設計の妥当性や性能に関する議論はされていない．

† 日本電信電話株式会社 サイバースペース研究所
NTT Cyber Space Laboratories

本稿では，高速なI/Oデバイスをエミュレートする疑似デバイスドライバを用いて，Intel社のVirtualization Technology for Directed I/O Architecture（VT-d）と呼ばれるIOMMU[6]が，LinuxのI/O性能に与える影響を報告する．

本稿の構成は以下の通りである．まず，2章で，IOMMUのハードウェア，Linuxの制御プログラムについて説明する．次に，3章で性能評価結果を報告する．4章で関連研究について述べ，5章でまとめを行う．

2. IOMMU

2.1 IOMMUハードウェア設計

2.1.1 アドレス変換テーブル

IOMMUは，固定サイズのI/Oページ単位で，I/Oデバイスが用いるDMA仮想アドレス空間を物理アドレス空間にマップする．IOMMUの制御プログラムは，DMA仮想アドレスと物理アドレスのアドレス変換テーブルを管理し，IOMMUは，DMAトランザクション中に，そのテーブルを参照する．変換テーブルのエントリは，物理アドレス，リード・ライトのアクセスコントロールなどの情報を含んでいる．なお，IOMMUは，DMA仮想アドレスすべてを物理アドレスにマップするわけではなく，必要に応じてマップする．I/Oデバイスは，物理アドレスにマップされていないDMA仮想アドレスは使用できない．

VT-dのように仮想化環境をサポートするIOMMUは，複数のアドレス変換テーブルをサポートし，システムは独立した複数のI/Oアドレス空間を定義できる．Virtual Machine Monitor（VMM）は，複数の仮想マシンを安全に動作させるために，DMAを分離させることができる．

従来のIOMMUは，図1に示すような，連続したメモリ領域に，変換テーブルを保存する単純なデータ構造を使っている．一方，VT-dのアドレス変換テーブルは，多層のページテーブルから構成されるデータ構造である．VT-dのアドレス変換テーブルは，メモリ管理ユニット（MMU）が用いる，仮想アドレスと物理アドレスのマッピングを管理するデータ構造と類似している．図2は，I/Oページのサイズが4KBで，3層のページテーブルを使う場合のVT-dのアドレス変換テーブルの例である．VT-dの変換テーブルのエントリのサイズは8バイトであり，1つのページテーブルに256個のエントリが保存される（ページテーブルのインデックスとして9ビットが使われる）．

多層のページテーブルを使う変換テーブル構造は，物理アドレスにマップされていないDMA仮想アドレス空間部分の変換テーブルを作成する必要がないため，IOMMUの管理プログラムのメモリの使用量が少ないという利点がある．その一方，このデータ構造は，一つのアドレス変換のために複数回のメモリフェッチを要するため，オーバヘッドが大きいという欠点や，制御プログラムが複雑になるという欠点も持っている．

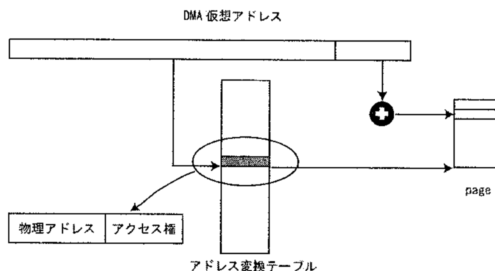

図1 連続するメモリ領域を使ったアドレス変換テーブルの例

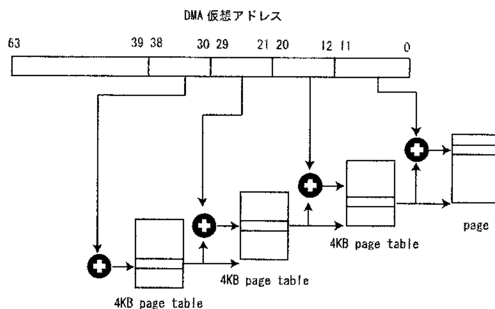

図2 VT-dの多層ページテーブルを使ったアドレス変換テーブルの例

前述のように，VT-dは複数のアドレス変換テーブルをサポートするため，制御プログラムのメモリ使用量を重視し，多層のページテーブルのデータ構造を使っていると考えられる．

2.1.2 I/O TLB

IOMMUのアドレス変換を高速化するために，変換テーブルの内容をキャッシュするのが，IO Translation Lookaside Buffer（IOTLB）と呼ばれるハードウェアである．VT-dは変換テーブルにアクセスするオーバヘッドが大きいため，従来のIOMMUよりも，IOTLBの重要性が増す．

DMA仮想アドレスがマップされる物理アドレスを変更した際には，古い物理アドレスのマッピング情報を，IOTLBから確実に削除するために，フラッシュ（無効化）操作が必要になる．VT-dのように，IOTLBの一部をフラッシュできるIOMMUハードウェアもあれば，IOTLB全体のフラッシュのみをサポートするIOMMUハードウェアもある．IOTLBのフラッシュ操

作はI/O性能を低下させるため，多くのIOMMU制御プログラムは，フラッシュ操作の回数を減らす仕組を持っている．

2.2 DMA要求インターフェイス

DMAを実行するソフトウェアコンポーネントは，IOMMUの制御プログラムに依頼し，DMAによるデータ転送を実行するメモリの物理アドレスを，DMA仮想アドレス空間のどこかにマップしてもらう必要がある．本稿では，DMAを実行するソフトウェアコンポーネントが，IOMMUの制御プログラムに要求を伝えるために使うソフトウェアインターフェイスを，DMA APIと呼ぶ．

IOMMUハードウェアに直接アクセスするIOMMUの制御プログラムは，OSのカーネル内に実装され，仮想化環境では，IOMMUの制御プログラムはVMM内に実装される．一般に，仮想化環境で必要になる，複数の仮想マシンでのI/Oデバイス共有を支援するハードウェア機能[7]をサポートしたI/Oデバイスはまだ販売されておらず，仮想化環境のI/Oの仮想化を支援するハードウェア機能は準備が整っているとは言えない．そのため，本稿では，OS，具体的には，Linuxカーネルが IOMMU を制御する環境について議論する．

Linuxは，IOMMUハードウェアをサポートする数多くのアーキテクチャ（X86，Alpha，POWER，SPARC，IA64，PARISCなど）で動作する．DMA APIは，これら複数のIOMMUハードウェアの違いを隠蔽し，共通のソフトウェアインターフェイスを，DMAを実行するソフトウェアコンポーネントに提供する．

DMA APIの主目的は，IOMMUに，DMAの対象となるメモリをDMA仮想アドレス空間のどこかにマップするよう要求することである．DMA仮想アドレス空間の広さは有限なので，DMAによるデータ転送完了後は，そのマッピングを解除することも必要である．本稿では，この一対の操作を，DMAマップ，DMAアンマップと呼ぶ．

LinuxのDMA APIは，2種類のDMAマップ・アンマップのためのインターフェイスを持っている．1つ目が，dma_map_single関数とdma_unmap_single関数の組で，dma_map_single関数は，DMAの対象となるメモリのカーネル空間の仮想アドレス，データ長，DMAの方向（物理メモリからデバイス，デバイスから物理メモリ，または双方向の3種類）を受取り，DMA仮想アドレスを返す．dma_unmap_single関数は，dma_map_single関数の返り値であるDMA仮想アドレスを受取り，アンマップする．

2つ目のDMAマップ・アンマップのためのインターフェイスは，dma_map_sg関数・dma_unmap_sg関数の組である．dma_map_single関数とdma_map_sg関数の違いは，前者は1つのアドレスを受取るが，後者は，メモリアドレスとその長さの組を扱うことのできるscatterlist構造体（ユーザ空間のiovec構造体に相当する）を受け取る点である．

2種類のDMAマップ・アンマップインターフェイスは，IOMMUハードウェア，IOMMU制御プログラムから見ると違いはなく，どちらも，IOMMUのアドレス変換テーブルの更新操作である．

DMA APIの利用方法の一例として，ストレージデバイスドライバ（SCSIやSATAのホストバスアダプタドライバ）の利用方法を説明する．writeシステムコールなどによるユーザ空間からのI/O要求は，最終的に，ページキャッシュとして使われているページフレームに対するDMA要求として，ストレージデバイスドライバに届く．また，ユーザ空間のI/O要求が，direct I/Oであった場合は，ユーザ空間として使われているページフレームへのDMA要求となる．ストレージデバイスドライバは，このようなページフレームをDMAマップするよう要求し，返り値として得られたDMA仮想アドレスをストレージデバイスのハードウェアに通知し，DMAデータ転送を開始する．DMA完了後，デバイスドライバは，ページフレームをDMAアンマップする．

DMAマップ・アンマップ操作は，オーバーヘッドが大きいため，その回数は少ないことが望ましい．最適化手法の1つとして，I/Oリクエスト毎にDMAマップ・アンマップするかわりに，予めDMAマップしたページフレームを確保し，そのページフレームをアンマップせずに，DMA転送の対象バッファとして繰り返し利用する，という手法がある．しかし，ストレージデバイスドライバは，DMA転送対象のページフレームを選択できず，仮想記憶システムが指定したページフレームにDMA転送するため，この手法はカーネル主要部分への変更が必要となり，容易には実現できない．

2.3 制御プログラム

IOMMUの制御プログラムの主機能は，DMA APIの要求に応じて，アドレス変換テーブルを管理することである．高いI/O性能を実現するためには，この機能の効率が重要である．

アドレス変換テーブルを管理する上で，制御プログラムの最も重要な部分は，DMA仮想アドレス空間のマップされている領域，つまり，使用中・未使用の領域の管理方法である．制御プログラムは，DMA API

からのDMAマップ要求に応じて，できるだけ素早く，使用していないDMA仮想アドレスを見つける必要がある．

DMA仮想アドレス空間の管理は，カーネル・メモリ・アロケーター（KMA）と類似しており，そのアルゴリズムを評価する際の判断基準も同じように，メモリの使用量，速度など同じ基準が適用される．しかし，IOMMUの空間管理は，KMAのそれとは異なる特性も持っている．

まず，2.2節で説明したように，DMA仮想アドレスは，DMAデータ転送中という，ごく短い間だけ使われるという特徴がある．この特徴から言えることは，例えば，KMAと異なり，DMA仮想アドレス空間のフラグメント化は大きな問題にならない可能性が高い．

次に，IOMMUの空間管理には，IOTLBのフラッシュ操作回数を削減するという要求条件がある．IOMMU制御プログラムは，DMAアンマップ操作要求後すぐに，アンマップしたアドレスに関する情報をIOTLBから削除するのが最も安全な管理ポリシーである．変換管理テーブルを更新しても，IOTLBにアドレス変換情報がキャッシュされている場合，I/Oデバイスは，DMAアンマップ後にもかかわらず，引き続き仮想DMAアドレスを使って，物理アドレスにアクセスできてしまう．しかし，多くのIOMMU制御プログラムは，セキュリティよりも性能を重視し，フラッシュ操作を遅延し，まとめて実行するポリシーを適用している．DMA仮想アドレスを異なる物理アドレスにマップした場合は，必ずフラッシュ操作が必要になるため，この最適化を効果的にするためには，一度，使った仮想I/Oアドレス（物理アドレスにマップした）は，できるだけ長い間使わないようにする，という工夫が必要である．

VT-d以外のIOMMU管理ソフトウェアが使うDMA仮想アドレス空間の管理アルゴリズムは，ほぼ同じである．これは，連続したメモリ領域を確保し，I/Oページ毎に1ビットを割り当て，使用中のI/Oページのビットを立てるという方法である☆．

ビットマップを使った空間管理には，一般に検索効率が悪いという欠点があるが，IOMMU制御ソフトウェアに関しては，この欠点は大きな問題にならない．IOMMUの空間管理アルゴリズムは，DMAマップ要求に対して，新しい未使用のDMA仮想アドレスを探す際，最後に使ったDMA仮想アドレスの上位方向に

☆ AlphaのIOMMUの制御ソフトウェアは，連続したメモリ領域を割り当てるのではなく，IOMMUハードウェアが使う変換マップを直接利用しており，I/Oページ毎に1ビットではなく，8バイト（変換マップのエントリのサイズ）消費している

```
struct iova {
        struct rb_node node;
        unsigned long pfn_hi;
        unsigned long pfn_lo;
};
```

図3 使用中のDMA仮想アドレス範囲を表すデータ構造

隣のアドレスから検索を開始し，高位方向に検索する．つまり，DMA仮想アドレス空間は，順番に使われて，最も高位のDMA仮想アドレスを物理アドレスにマップした後は，最も低位のDMA仮想アドレスをマップしようと試みる．前述のように，DMA仮想アドレスは短い時間だけしか使用されない性質があるので，DMA仮想アドレス空間を一周する間に，DMA仮想アドレスはアンマップされて，未使用になっており，大量のビットマップを検査する可能性はほとんどない．

ほとんどのビットマップを使った空間管理アルゴリズムは，DMA仮想アドレス空間を一周するまで，DMA仮想アドレスを新たな物理アドレスにマップしないというポリシーを採用しており，IOTLBのフラッシュ操作回数を容易に削減できる．具体的には，DMA仮想アドレス空間が順番に割り当てられて，1周につき1回だけフラッシュ操作が必要になる．

VT-dのDMA仮想アドレス空間の管理アルゴリズムは，図3の使用中のDMA仮想アドレスの範囲（開始と終了アドレス）を表すデータ構造を持つノードを持つ，Red Blackツリーを使っている．

Red Blackツリーはバランスツリーであり，ビットマップよりも検索効率が良いという利点がある．しかし，前述のように，IOMMU制御プログラムに関しては，ビットマップを使った空間管理アルゴリズムの検索効率は欠点にならないため，Red Blackツリーを使う利点は明らかではない．Red Blackツリーの空間管理アルゴリズムには，DMAマップのたびに新たに構造体を確保するオーバヘッドがあり，DMAマップ・アンマップの毎にツリーのローテーションというオーバヘッドもある．

Red Blackツリーの空間管理アルゴリズムは，ビットマップと比較するとメモリの使用量が少ないという利点もある．ビットマップの空間管理アルゴリズムは，DMA仮想アドレス空間全体を表現するビットマップが必要であり，未使用のDMA仮想アドレス空間に関してもメモリを消費する．例えば，I/Oページサイズが4KB，DMA仮想アドレス空間の広さが4GBであれば，128KBのビットマップが常に必要である．Red Blackツリーの空間管理アルゴリズムは，使用中のDMA仮想アドレスだけを管理すればよいため，使用中のDMA

表 1　評価した IOMMU 制御プログラム

	説明
disabled	IOMMU を無効化.
default	デフォルト設定.
strict	DMA アンマップ毎に IOTLB の相当するアドレスを無効化.
bitmap	ビットマップによる DMA 仮想アドレス管理.
strict bitmap	ビットマップによる DMA 仮想アドレス管理を使い，DMA アンマップ毎に IOTLB の相当するアドレスを無効化.

仮想アドレスがない状態では，ツリーの頂点のデータ構造，たかだか数バイトのメモリしか消費しない.

VT-d の管理ソフトウェアは，IOTLB フラッシュ操作回数を削減するため，DMA アンマップした領域のノードをすぐに開放せずに，アレイに登録し，1 度のフラッシュ操作で，複数の DMA 仮想アドレスをまとめて，IOTLB から削除する．フラッシュ操作のタイミングは，DMA アンマップ操作から 10ms 以上経過する，または，250 個あるアレイが満杯になった場合である．なっており，Intel 社の開発者から，この最適化による UDP の性能向上が報告されており[5]，デフォルトで有効になっている.

3. 性能評価

3.1 評価した IOMMU 制御プログラム

我々は，Linux カーネルのバージョン 2.6.26-rc6 の VT-d 制御プログラムを使って，表 1 に示す 5 種類の設定で性能評価を行った.

disable は，IOMMU を無効にした設定である．カーネルの起動オプションとして，intel_iommu=off を指定している.

default は，標準の設定である．2.3 節で述べたように，IOTLB フラッシュ操作をバッチする最適化が含まれている.

strict は，上記の最適化を無効にして，DMA アンマップ時に，アンマップされた DMA 仮想アドレスが IOTLB から削除されていることを保証する設定である．カーネルの起動オプションとして，intel_iommu=strict を指定している.

bitmap と strict bitmap は，DMA 仮想アドレス空間の管理アルゴリズムとしての Red Black ツリーの有効性を評価するため，空間管理アルゴリズムを，Red Black ツリーからビットマップに置き換えた，我々が実装した VT-d 制御プログラムである．bitmap と strict bitmap の違いは，default と strict の違いと同様で，strict bitmap は，IOTLB フラッシュ操作をバッチで実行しない.

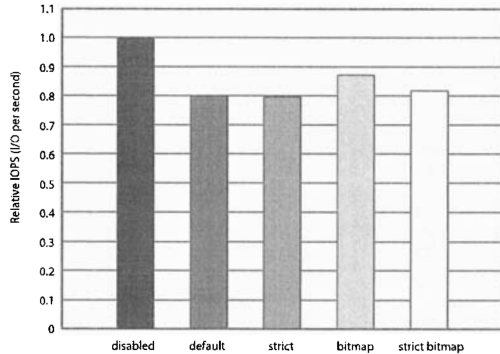

図 4　IOMMU を無効にした設定に対する相対性能

3.2 環　　境

測定では，Intel Core2 Duo 3.0 GHz（E6850）に 4 GB のメモリを搭載した Dell OptiPlex 755 を使った．Linux カーネルのバージョンは 2.6.26-rc6 である.

今回の測定では，10 Gigabit Ethernet のような高速な I/O デバイスが利用できなかったため，PCI デバイスに接続された SCSI ホストアダプタをエミュレートする疑似ドライバを実装し，性能評価に用いた．このドライバは，受け取った I/O リクエストを DMA マップした後，実際の DMA 転送を実行せず，すぐにアンマップし，I/O リクエストを完了させる.

3.3 結　　果

図 4 は，I/O サイズが 1024 バイトの I/O 要求を繰り返し発行するベンチマークの結果である．ベンチマークが同時に発行する I/O 要求の数は 1 に設定した．ベンチマークを 30 秒間動作させ，5 回の測定結果の平均値を示した.

default の IOPS の平均値は，124,337.2 回であり，DMA マップ・アンマップ操作の組を 1 秒間に平均 124,337.2 回実行したことになる．10 Gigabit Ethernet では，この 10 倍以上の回数が要求される可能性がある[1].

default と strict を比較すると，IOTLB フラッシュ操作のバッチは，ほとんど性能向上がなかった．一方，bitmap と strict bitmap では，IOTLB フラッシュ操作のバッチが性能を向上させている.

default の IOPS は，約 125,000 であり，2.3 節で説明した，10ms のタイムアウトより前に，2ms 毎に 250 個のアレイが満杯になり，フラッシュ操作を実行している．一方，bitmap は，4GB の DMA 仮想アドレス空間を使いきるまで，IOTLB フラッシュ操作を実行しない．今回のベンチマークでは，1 つの I/O 要求が，4KB の DMA 仮想アドレス空間を使うため，2^{20} 回の

DMA マップ・アンマップ操作，約 8 秒毎に 1 回のフラッシュ操作を実行している．default と strict の性能差がほとんどないことからも，250 個というアレイのサイズは高速な I/O 負荷には効果がないことがわかった．アレイを大きくすることは可能だが，default が，bitmap と同様に約 8 秒毎に 1 回のフラッシュ操作を実行するためには，2^{20} 個の iova 構造体に 40MB のメモリ，アレイに 8MB のメモリが必要になる．

strict と strict bitmap は，約 2%の差があり，事前に予測したように，性能という評価基準では，IOMMU の空間管理アルゴリズムとしては，bitmap が Red Black ツリーよりも優れていたと言える．

なお，CPU の使用率に関しては，全ての設定で，約 70%と大きな違いは見られなかった．

4. 関連研究

IOMMU が性能に与える影響についての報告は，Muli らの研究[4] のみである．Muli らの研究は，IBM 社のハイエンド x86 サーバや POWER アーキテクチャなどで用いられる，Calgary と呼ばれる IOMMU が性能にあたえる影響を評価している．本稿と異なり，制御プログラムによる性能への影響は評価していない．

仮想化技術における，I/O 仮想化のハードウェアによる支援という観点では，IOMMU のようにシステムに搭載された I/O デバイス全てに適用できる方法ではなく，個々の I/O デバイスに I/O の仮想化のためのハードウェア機能を実装する試み[9] もされている．

5. まとめ

本稿では，IOMMU ハードウェア，Linux の IOMMU 制御プログラムの設計に関して議論し，Intel 社の IOMMU（VT-d）が I/O 性能に与える影響を評価した．性能評価では，IOTLB のフラッシュ操作が性能に大きな影響を与えることを明らかにし，標準の Linux カーネルで，20%の性能低下が確認できた．IOMMU 制御プログラムを変更することで，性能低下を 13%まで改善できた．

今後の課題としては，まず，疑似ドライバの代りに，10GbE ネットワークアダプタなどの実際のハードウェアを用いた性能評価があげられる．また，複雑なワークロードに対する性能評価，および，VMM で IOMMU を使う環境での性能評価があげられる．

性能面の課題としては，並列性の向上も課題である．IOMMU 制御プログラムは，1 つのロックで空間管理のためのデータ構造，ビットマップ，ツリーを排他制御しているため，並列性の高い I/O 負荷に対して，スケーラビリティの問題が生じる可能性がある．

DMA 仮想アドレス空間管理にビットマップを使った VT-d 制御プログラムのソースコードは，`http://lkml.org/lkml/2008/6/4/250` で入手可能である．

参考文献

1) 10 Gigabit Ethernet on UltraSPARC T2, October 2007. `http://blogs.sun.com/puresee/entry/10_gigabit_ethernet_on_ultrasparc`.
2) Advanced Micro Devices. Amd i/o virtualization technology (iommu) specification, February 2007. Publication 34434, Revision 1.20.
3) Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *The 19th ACM symposium on Operating Systems Principles*, pp. 164–177, October 2003.
4) Muli Ben-Yehuda, Jimi Xenidis, Michal Mostrows, Karl Rister, Alexis Bruemmer, , and Leendert Van Doorn. The price of safety: Evaluating iommu performance. In *Ottawa Linux Symposium*, pp. 9–19, July 2007.
5) Mark Gross. iommu-iotlb-flushing, March 2008. `http://lkml.org/lkml/2008/3/5/450`.
6) Intel Corporation. Intel virtualization technology for directed i/o architecture specification, September 2007. Revision 1.1.
7) PCI-SIG. `http://www.pcisig.com/`.
8) VMware. `http://www.vmware.com/`.
9) Paul Willmann, Jeffrey Shafer, David Carr, Aravind Menon, Scott Rixner, AlanL. Cox, , and Willy Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *the International Symposium on High-Performance Computer Architecture*, Phoenix, AZ, February 2007.