論文紹介: Hive: Fault Containment for Shared-Memory MPs

紹介者: 久野*

1996.4.26

1 Introduction

- SMP はサーバとして多く使われるようになった←動的なマルチプログラミング負荷に強い
- しかし SMP OS は (登場し始めた) 大規模 SMP にはいまいち。

Hive: 大規模 SMP 用 OS。根本的な違い→複数の分散 OS(セルと呼ぶ) の集合体だということ。利点:

- 障害に強い: クラッシュは 1 つのセルに限定される。
- スケーラブル: 単一カーネルのボトルネックがない。

課題:

- 障害隔離 (fault containment): いかに障害を 1 セルに限定するか。とくに野蛮書 (wiled write) 対策。
- 資源共有: セルの境界を越えて CPU、メモリ等を融通したい。
- 単一システムイメージ: ユーザには 1 つの OS に見える。

本論文は、障害隔離に焦点。プラットフォームは Stanford FLASH(障害隔離のためのハードウェア機構を持つ)。具体的な方策は:

- 防火壁 (firewall): 他のセルからの書き込みを捕捉するハードの利用。
- 障害を起こしたセルが書き込めるページを破棄。
- 迅速な障害検出。

セル間のメモリ共有は 2 レベル:

- 論理レベル: ファイルやメモリ空間。
- 物理レベル: あるセルのメモリが不足→他のセルのメモリを借りる

どちらも、障害隔離のための考慮必要。たとえば他のセルのメモリを読もうとして読めない→タイムアウト→障害処理。

Hive は SGI IRIX 5.2 に基づき、これと互換。テスト/評価は SimOS と呼ばれる FLASH シミュレータによる。現在得られている結果:

- 各種のメモリ障害が隔離可能。
- カーネルの障害も隔離可能。
- 4 セル版で、IRIX5.2 と比べて 0%〜11%の性能低下。

ゆえに、大規模 SMP の OS として有望 (だがまだ研究途上)。

2 SMP における障害隔離

- 障害隔離: 多くの分散 OS で実装。耐障害 (fault tolerance) とは違う←部分的な障害は起きてよい。だから二重系は不要。
- SMP をサーバに使う場合、障害隔離は魅力的←複数の独立したプロセスが動いているから
- ただし、OS が賢くないとだめ。プロセスのメモリがそこら中のセルに入ってしまえば、1 セルがダウンしたらそのプロセスも死ぬ。

要求:

> アプリケーションが死ぬ確率はアプリケーションが占有するシステム資源に比例すること。cf. システム全体の資源に比例…通常の SMP

*Proc. 15th ACM Symp. on Operating Systems Prinsiples, ACM SIGOPS vol. 29, No. 5, 1995

大規模なアプリケーションはどのみちシステムを全部使うので助からない (あたりまえ)。チェックポイント等を使うべし。

障害隔離は分散 OS でもやっているが、SMP だとより大変。ハード的には:

- CC-NUMA (fig 2.1) →障害の単位はノード→ノードのメモリがアクセスできなくなる/そのノード上のキャッシュが失われる。
- OS としては: 障害のあと何を保証すべきか: 影響されなかったメモリはこれまで通り。影響のあったメモリをアクセスしようとした CPU にはエラーが返される (無限遅延はこまる)。エラーのあったメモリの範囲が切り分けられる。
- つまりメモリ障害モデルが必要。FLASH では、(1) プロセサ間通信は途絶しない、(2) 影響を受けるメモリは障害のあったセルが書き込み可能であった部分のみ (防火壁の機能)。

ソフト的には:

- 共有メモリの弱点: 野蛮書。文献 7 →ハード障害よりソフト障害の方が多い。文献 20 → 5 年間にわたる 3000 の障害調査で、15〜25%は野蛮書を伴った。
- 既存の SMP: 野蛮書には無防備。メモリ保護は仮想記憶機構のみで、それは野蛮書を行う CPU が管理している。
- 対策: ハード的またはソフト的。ハード→キャッシュコントローラに入れるのが自然。ソフト→他のセルの信頼できるソフトウェア機構によって野蛮書を防止。ここでは信頼性、簡潔さからハードを選択。
- 各メモリページごとに、書き込み許可ビットベクタを持たせる。

3 Hive Architecture

- セルの集合体 (図 3.1)。各セルの範囲はブート時に決定。各セルは自分の CPU、メモリ、I/O を管理。セル群は協調して 1 つのシステムであるかのようにふるまう。
- Hive の機能→障害隔離のためのものと、資源共有のためのもの。

3.1 Fault Containment Architecture

- ハード的には前述の通り。OS レベルでは他のセルに悪い影響を及ぼすのは次のもの。
- 悪いメッセージを送る。
- 間違ったデータを作成し、それを他のセルが読む。
- リモート書き込み。
- なお、重要なデータを読めなくするというのもあるが、これは別の問題として取り上げない。

悪いメッセージ:

- RPC での消毒 (sanity check) を行う。普通の分散 OS と同じ。

間違ったデータを読ませる:

- 読む側のセルで注意深くする。そのため用心参照 (careful reference) プロトコルを用意。読んだ後はもちろん消毒。

リモート書き込み:

- 各セルは他のセルのカーネル内部データを直接書くことはしない。もちろん、ユーザレベルページは共有され書くことができる。なので:
  - 保護するページの選択: セルローカルなプロセスのページは他のセルには書かせないし、セルにまたがるものでも最小限にする。ただし許可を落とす方向に設定するのは相手セルとの通信を伴う (向うのキャッシュをフラッシュしてもらう) ため高価。
  - 野蛮書対策: ユーザページへの野蛮書→メモリ故障と同等程度にまで減らしたい (でないと使ってもらえない) →障害を検出して、壊された可能性のあるものはすべて破棄…悲観的アプローチ

    - ただし、検出される前に使われたり、書き込み権を放棄する可能性→完全に防止するのは書き込み共有の放棄→それは受け入れ難い (SMP の利点の放棄だから) →放棄せずにどこまで行けるか、普通の SMP OS 程度にまでなるかが課題。
- 検出: 分散 OS で研究されて来た課題。停止したセルはすぐわかるが、狂ったまま動いているセルはわかりにくい。他のセルから「あいつは狂ってるから止める」と言えることにすると、狂ったセルが他のセルを止めまくれることになってしまう。
- そこで、各セルは常に他人を観察し、あやしいと「怪しい!」とだけ言う→ユーザプロセスはすべて停止して、合意アルゴリズムの実行→合意された場合にそのセルをリブートする→ユーザプロセスの再開。
- 危険期間 (window of vulnerability): 障害発生から最初のチェックで誰かが「怪しい!」というまで。チェック間隔と性能のトレードオフ。

3.2 Resource Sharing Architecture

- 課題: SMP の密な通信の利点と障害隔離の両立。メカニズム→カーネル、ポリシー→ Wax と呼ばれるユーザレベルプロセス。
- セルはリソースの配分について判断しない←グローバルな情報が必要だから。Wax はこの情報を持つがセルは持たない。セルは自分の持ち分の整合性と性能にのみ責任を持つ。

資源共有のメカニズム:

- 資源: メモリと CPU
- メモリ共有は 2 レベル (図 3.2)。論理レベル→ファイル (共有バッファキャッシュ) や仮想空間。物理レベル→メモリ資源の有効利用と、アクセスの起きる箇所にデータを置ける (CC-NUMA の特性活用)。
- プロセスはセル境界にまたがれる (またがりタスク、spanning task)。各セルはそのうちローカルな持ち分を管理するとともに、セルにまたがった共有情報も管理。プロセスのセル間移送も可能。

資源共有のポリシー:

- ポリシーは全セルにまたがったプロセス Wax が決定 (図 3.3)。何についてのポリシーか→図 3.4。
- 過去の分散システムでは次の 2 通りの方法
    - 各カーネルが通信しながらポリシー決定→全体像は見えないまま
    - 1 箇所で集中管理→ボトルネック
- Wax ではまたがりタスクの共有メモリにすべての情報を保持できる。その各スレッドは個々のセルで動作しそのセルの情報を調べる。
- Wax はふつうのユーザレベルプロセス。セルが死んだら Wax は一旦終了し、再度起動しなおし、情報を収集する (情報のリカバリが簡単)。
- Wax から OS に渡すデータは各セルで消毒され、また各セルの動作の整合性自体は Wax ではなく各セルのカーネルが保証→ Wax によってシステムの信頼性が損なわれることはない。

3.3 Implementaiton Status

- とりあえず、障害隔離機構に焦点をあてて実装中。セル、用心参照プロトコル、野蛮書対策、障害検出、リカバリ、論理共有、物理共有、セル間高速 RPC など稼働。
- 単一システムイメージは途中。セル間 fork、分散プロセスグループ/シグナル、共有ファイルなど稼働。またがりタスク、Wax、合意プロトコル、耐故障ファイルシステムなど実装中。
- 現在のプロトタイプでも、障害隔離が実装可能なこと、それでいてオーバヘッドは小さいことは示せている。しかしフル実装による実証はこれから。
- 性能評価は SimOS による。SGI Challenge クラスのマシン (R4000 200MHz 4CPU、700nsec メモリアクセス時間) をモデル化。実行例としては pmake(並列 make)、ocean(シミュレーション)、raytrace。詳細は 7 節。

4 Fault Containment Implementation

- あるセルが別のセルに危害を与えるケース: (1) 悪いメッセージ、(2) 悪いデータを他のセルに読ませる、(3) 他のセルへの書き込み。
- (1) については、分散システム (e.g. NFS) で実績がある。
- (2) と (3) が Hive の新しい点。

4.1 Careful Refrence Protocol

- 他のセルのデータを直接読む←RPC では遅すぎる、最新情報がほしい、多数のセルから参照など。
- 読み出したデータはメッセージの場合同様消毒すればよいが、読み出しそのものに問題。
- ノードが fail すると、メモリ参照は bus error に。
- 普通の OS →カーネル実行中のメモリ参照 fail → panic(自分から shut down)。しかし他のセルの場合はこれでは困る。
- Hive では、注意参照 (careful reference) プロトコル。
  1. careful_on() を呼ぶ (スタックフレームの保管、bus error が起きたらハンドラにより復帰)
  2. リモート参照の前に必ず境界揃えとメモリ範囲の検査
  3. 消毒の前にすべてのデータをローカルメモリにコピー
  4. データ構造ごとに構造識別子 (malloc が設定) を検査
  5. すべて済んだら careful_off() で元に戻る
- 例: クロックモニタアルゴリズム: 各セルのクロックハンドラが他のセルのクロック値をチェック。
- 200MHz CPU で careful_on() から careful_off() まで 1.16us(232cycles)、うち 0.7us(140cycles) はリモートセル参照のキャッシュミス。
- RPC だと最低でも 7.2us。

4.2 Wild Write Defense

野蛮書対策→2 部構成。(1)FLASH ハードウェアの firewall 機構で、他のセルに書けるページを最小限に。(2) セルが fail したらそのセルに書ける状態のページを全部捨てる。

4.2.1 Firewall

- 4KB page 毎の 64bit vector で CPU ごとに書き込み可否を設定 (64CPU より多い場合はグループ化して map)
- bit が off なら書き込みは bus error に。bit の変更は local CPU のみ可能。
- 各ノードのキャッシュコントローラ→そのノード用のビット状態を追跡。キャッシュラインの所有者が誰か/書き戻し時点も追跡。
- 他のセルの I/O エリアのアクセスは bus error。DMA による read/write は DMA を持っているセルの CPU による read/write と同じ。
- 4KB というのは OS のページサイズに合わせた。これより大きくてもメモリ割り当てがやりにくい。小さくするといいかどうかは?
- bit vector の代案: 1bit の書き込み保護ビット (エラー隔離にならない)、1 バイトで書き込み可能 CPU を指定 (セルの内部での付加バランスができない) [1]
- firewall のコストは小さい←remote write cache miss の平均処理時間は firewall check が入ると pmake で 6.3%、ocean で 4.4%増加。write cache miss 自体の比率はかなり小さい。

4.2.2 Firewall management policy

- エラー隔離と性能のトレードオフ。
- remote write が必要なのはある CPU の TLB に書き込み可能なリモートページがある場合。しかしこの状態は TLB miss ごとに変わるので、そのたびに RPC で元のセルに許可を求めていられない。[2]

[1] 書き込み可能セルを指定したら?
[2] MIPS R は TLB のみでマッピングを行い、TLB miss は OS で扱う

- では firewall をどうやって管理? 当面は単純で比較的 writable の少なくてすむ方法。
- ページに write fault が起きたら、そのセルの CPU すべてにまとめて書き込み許可ビットを立てる→セル内での仕事の配分は自由。
- そのセルのどれかの CPU がページを write map している間は許可ビットは立ったまま。
- ページが writable で map されるのは、ファイルを書き込み可能モードでマップした場合→そのセルのプロセスが要求した場合のみ。
- pmake(共有 write 少) → 5.0 秒の実行中を 20msec 間隔でサンプル→ 15 remote writable pages/cell。max 42 — /tmp のファイルサーバとなったセル。独立プロセスの集合に対してはうまく働く。
- ocean(共有 write 多) →〃→ 550 remote writable pages/cell。どのみち全セルが共同で仕事をしているから、隔離は無理。

4.2.3 Preemptive discard

- cell failure の後どのページを破棄するか調べるのは大変→全 TLB をフラッシュし、全 mapping を remove →後に再度 page fault が起きたときチェックする。
- ページを破棄→ファイルシステムのセマンティクスが変わってしまう可能性。
- dirty なページを破棄した場合、後にそのページをアクセスしようとしたプロセスはエラー通知を受けるべき…だが、ずーっと後でアクセスするかも知れないのでむずかしい。
- ほとんどの UNIX ではクラッシュ時の dirty なページを調べたりせず、reboot 時に単にディスクにあるものを読み出す←クラッシュしたらローカルなプロセスも死ぬので、dirty なページをアクセスしたプロセスが不整合を観測するということがない。[3]
- そこで Hive でもセルが fail した後そのセルのディスクから読み出すプロセスは単にディスクにあるものが読める、ということに。だからエラーを通知されるのは fail の前にファイルを open していたプロセスのみ。
- 実装は世代番号。dirty なページを破棄したら世代が進み、古い世代番号の記録された file descriptor やアドレスマッピングから参照しようとするとエラー。

[3] でもその情報を別のページに書いてそれはディスクに入ったら…

4.3 Failure Detection and Recovery

- 野蛮書き込みによる影響を少なくするため、できるだけ迅速に fail を検出←定期的な整合チェックによる←チェックが失敗したら分散合意アルゴリズムで合意
- RPC timeout ← fail の可能性 (分散 OS と同じ)。それ以外に (1) メモリアクセスが bus error、(2) クロックモニタチェックが失敗、(3) 読み出したデータの整合性チェックが失敗。
- 一定時間内に同じ警告を出して 2 回とも合意アルゴリズムでウソだと言われたノードも fail したとみなす。
- 合意アルゴリズムは標準的なもの (未実装)。

4.3.1 Recovery algorithm

- 生きているセルがどれとどれか決まったら、回復アルゴリズムを実行。その最初には、ページフォルト処理や他のセルからのリモートメモリアクセスがまだ残っているかも。
- というのは、回復中はユーザプロセスは凍結されるがカーネルレベルのプロセスは凍結されない (回復アルゴリズムのためにも必要)。
- そこで、まず TLB をフラッシュしてマッピングを削除したら最初のバリアに入る。ここを通過した後のページフォルトはクライアント側で保留しておく。
- そこで、以後は正当なページフォルトもリモートアクセスも残っていないので、firewall の書き込み許可ビットを落として VM cleanup(fail したセルが書けるページを破棄したりそれをファイルシステムに通知したり世代番号を進めたりといった処理をする)。

- VM cleanup が終わったら 2 番目のバリアに入り、それを抜けたら通常動作に復帰 (保留していたページフォルトも処理)。
- ページフォルト RPC のサーバ側は回復のためにブロックする必要がなく、割り込みレベルのみで動作でき効率的。
- 回復の最後にマスタを決め、マスタは fail したセルのハードウェアチェックを行い、正常なら reboot させる。

5 共有メモリの実装

- エラー隔離を実装しながらいかにうまく共有も許すか?
- 論理レベル共有と物理レベル共有がある。
- セルの役割り→ 3 種類
  - Client: データを参照するプロセスが動いているセル
  - Memory Home: 物理ページを持っているセル
  - Data Home: そのページの中身を持っているセル
- データホームは各種の管理をする。現在の実装ではデータホームは常にそのページのバッキングストアを持つ。

以下ではまず Hive の基にした IRIX の説明から。

5.1 IRIX Page Cache Design

- 各ブロック (物理ページ) は pfdats データ構造が対応。pfdats には論理ページ id を格納。論理ページ id は tag と offset から成る。tag はその論理ページが所属する object を示し、offset はその中の位置を示す。ハッシュ表で論理 id から pfdats が引ける。
- ページフォルト→ pfdats を検索し、なければ vnode 経由でファイルシステムに頼んで物理ページを割り当て内容を読み込んでもらう。
- read/write システムコールもほぼ同様。

5.2 Logical-Level Sharing of File Pages

- Hive では、あるセルが別のセルのページをアクセスしたい場合、新しい pfdat(extended pfdat) を割り当てる→残りのシステムはリモートメモリアクセスだと知らないまま動作
- export/import — あるセルのページを別のセルの extended pfdat と対応づける操作。
- ページフォルトハンドラ→ないページ→ vonde に読み込み依頼→リモートノードを現す vnode →データホームに RPC →サーバ側はまだページがメモリになければ読み込む→データホームは export を実行→そのことはデータホームの pfdat に記録 (解放されないように/回復時に参照) →クライアントにそのページのアドレスを変えす→クライアントは import を実行→ extended pfdat が割り当てられハッシュ表に入る。
- 以上が一度起これば、あとはハッシュ表を見ればあるのでページフォルトは迅速に処理できる。
- クライアントがページを解放→ release を実行→ extended pfdat を解放してデータホームに RPC →データホームは (他に使われていなければ) ページをフリーリストに→再度必要になったら取り返せる
- ローカルページフォルト: 6.9us、リモートページフォルト: 50.7us (table 5.2)。リモートのうち 17.3us は PRC コスト、14.2us は IRIX の遅いデータ構造アクセス。
- 実際には完全に割り込みレベルで処理できない場合も。全体の効果を見るため pmake で計測。6 秒の実行→ 8935 faults → 4946 が remote →平均 455msec/fault (cf. 117msec on 1-cell system)。pmake の 1cell から 4cell になって遅くなった部分のうち 13%を占める。

5.3 Logical-level sharing of anonymous pages

- 無名ページ (swap エリア) もほぼ同様。
- 無名ページは copy-on-write tree で管理 e.g. Mach
- ただし parent と child が別のセルにあるかも。これを分散カーネル構造で実装してみた。

- copy-on-write tree の構造は同様→ポインタがリモートポインタ。ローカルプロセスに対応する leaf は必ずローカルメモリにあるが、他のノードはどこにあってもよい。
- 参照のみで変更は不要なので野蛮書の問題はない。
- 共有ページの fault は注意プロトコルしながらポインタをたどっていって該当ノードをみつけ、RPC で export/import をたのむ。
- この方法で安定して動いている (fault injection もパス) ので、安全な分散データ構造ができることは示せた。が、速くもない。普通に RPC でいきなり頼むのでもいいかも。

5.4 Physical-level sharing

- ここまでのところ、すべてのページはデータホームのキャッシュにあることになっていたが…データホームのメモリに、ということなら CC-NUMA の利点がない→物理レベル共有が回答
- メモリホームは extended pfdat を利用して物理ページを他のセルに貸し出せる (fig 5.3b)。貸し出したページは覚えておき自分では使わない。
- データホームは extended pfdat を割り当ててこのページを普通に利用 (ただし firewall の設定を変えるときは RPC でメモリホームに頼む)。
- とりあえず demand driven で。将来は Wax でポリシーを。
- 借りたページは一部の目的には使えない。メモリホームの野蛮書に無防備だから、カーネルには使えない。
- 借りたフレームはいらなくなったらすぐ返しているが、もっとよい方法を探求。

5.5 Logical/Physical intereactions

- 物理レベル共有と論理レベル共有は独立に動作。
- たとえば、クライアントがページを貸してデータホームがそこにデータを入れれば効率が良い。CC-NUMA を生かしている。
- これをサポートするため、VM は貸したページが import される時には元の extended pfdat を再利用する。

5.6 Memory Sharing and Fault Containment

- 共有により、他のセルの fail が自分のセルのユーザプロセスにダメージを与え得る。そこで:
- ページ割り当て/移動時に、なるべく同じところのを使う。
- 世代番号方式なので、cell fail で失われるデータの単位はファイル。だからファイルのメモリホームも絞る。
- むずかしいところ。

5.7 Summary of Memory Sharing Implementation

- 論理レベルと物理レベルの分離が重要。
- extended pfdat による位置透明性。
- 貸したページを返せとは今はいえない (むずかしい)。
- 将来は Wax で。

6 RPC Performance Optimization

- RPC の高速化は性能上重要→専用ハード
- 専用ハードなしでは、プロセサ間割り込み (IPI)+スキャン→遅いし壊されやすい。
- Short InterProcessor Send 機能 (SIPS) →キャッシュライン (128bytes) と同じ大きさのメッセージを特定プロセッサのキューに投げられる。速度はキャッシュミスと同じ。
- 信頼できるメッセージ→分散システムにくらべて実装が楽。

- null RPC → 7.2us (1440cycles)、うち 2us が SIPS。なので、返事待ちは spin wait (50usec したら timeout)。
- null でない実際には平均 9.6us (1920cycles)。おもにスタブ実行のオーバヘッド。
- この上にキュー機能を持つ RPC。null RPC で 34us。
- この落差→できるだけ割り込みレベル RPC でできるように設計。また、割り込みレベルで処理できない (ブロックする必要のある) 場合だけキュー機能に移行するように設計。

7 Experimental Results

7.1 SimOS Environment

7.2 Simulated Machine

7.3 Performance Tests

7.4 Fault Injection Tests

8 Discussion

9 Related Work

10 Concluding Remarks