Monitoração de eventos relacionados à memória transacional

Mauro Romano Trajber

IPT-SP

26 de maio de 2011

Cronograma

Cronograma

Cronograma

Problema

- Projetar e codificar programas paralelos ainda é muito difícil
- Falhas ocasionadas por concorrência são difíceis de serem reproduzidas

Travas de exclusão mútua

- Deadlocks, condições de corrida, lost wake-ups...
- Granularidade fina/grossa
- Composição para criação de estruturas maiores e mais complexas

```
move(q1, q2) {
    q1.lock.acquire();
    q2.lock.acquire();
    v = q1.remove();
    q2.insert(v);
    q2.lock.release();
    q1.lock.release();
}
deadlock!
```

Transações em bancos de dados

- Utilizadas por décadas com muito sucesso em SGBDs
- O autor de uma consulta não precisa se preocupar com concorrência

```
START TRANSACTION;
SELECT @A:=SUM(salary) FROM table1 WHERE type=1;
UPDATE table2 SET summary=@A WHERE type=1;
COMMIT;
```

Características

- **A**tomicidade
- **C**onsistência
- **I**solamento
- **D**urabilidade

Percebeu-se que o uso de transações pode trazer vantagens para a programação concorrente

Memória Transacional

- Alternativa aos mecanismos clássicos de controle de concorrência (*locks*)
- Nível de abstração mais alto
- Codificação sequencial, basta identificar trechos que devem ser executados de forma atômica
- Sem *locks*, portanto sem *deadlocks*
- Trechos atômicos (tudo-ou-nada)
- Podem ser compostas
- Se a transação falhar ela é executada novamente

```
atomic {
    x = x + 1
}
```

Problema

- O uso de memória transacional é vantajoso em relação a outros mecanismos de controle de concorrência ?
- A memória transacional esta sendo utilizada adequadamente ?

Objetivo

- Monitorar eventos relacionados às transações que estão sendo executadas por uma implementação de STM
- Pode ajudar o desenvolvedor a entender melhor o comportamento de seu código em execução
- Publicar informações no *procfs* por meio de uma chamada de sistema
- Verificar se é viável o uso de um sistema de monitoração

Construções transacionais

Espera condicional (modelo produtor/consumidor)

```
atomic {
    if (buffer.isEmpty()) retry;
    Object x = buffer.getElement();
    ...
}
```

Sentença condicional

```
atomic {
    do {
      x = fila1.getElement();
    } orElse {
      x = fila2.getElement();
    }
```

Composição

Existem basicamente três tipos de composição:

- Flattened
- Closed
- Open

```
int x = 1;
atomic {
   x = 2;
   atomic {
      x = 3;
      abort();
   }
}
```

Garantia de progresso

- Bloqueante: O sistema de TM faz uso de travas para recursos compartilhados
- Não-bloqueante: Sem travas, e o sistema de TM tenta estabelecer justiça entre as *threads* em execução
  - Sem espera: Se uma *thread* está em execução, então ela deve progredir
  - Sem travas: Se várias *threads* estão em execução, então alguma delas deve progredir
  - Sem obstrução: Se apenas uma *thread* está em execução, então ela deve progredir

Controle de concorrência

Fases de um conflito: acontece, é detectado, é resolvido

- Pessimista: As três fases ao mesmo tempo
- Otimista: Detecção e solução somente no final da transação. Mais paralelismo

Gerenciamento de versão

- Prematuro (*eager*): Valor alterado diretamente na região de memória e valor antigo em um *undo-log*. Usado em conjunto com controle pessimista
- Tardio (*lazy*): Escritas da transação são gravadas em um log e só são efetivadas em caso de sucesso

Detecção de conflito

Com controle pessimista de concorrência travas garantem que não haverá acesso concorrente à região de memória
Com controle otimista:

- Granularidade: Tamanho da palavra da arquitetura ou objeto
- Tempo: ao tentar acessar, durante a validação ou ao tentar finalizar a transação

Gerenciamento de conflito

Conflito detectado e agora ?
Várias políticas de resolução de conflito (Gerenciador de contenção)

- Passiva
- Karma
- Educada
- Tempo de vida

Quando é uma boa escolha

- Uso de travas prejudica a escalabilidade
- Modelagem e manutenibilidade são mais importantes que desempenho

Problemas relacionados à TM

- Sincronizações feitas pela TM podem prejudicar o desempenho
- Granularidade da transação ainda depende do programador (muito curtas/muito longas)
- Ainda é mais difícil de escrever/entender/depurar do que um código sequencial
- Programas paralelos são executados de maneira não-deterministica, portanto são mais difíceis de serem testados. Algumas falhas são quase impossíveis de serem reproduzidas

Depuradores

- Depuradores comuns não estão adaptados para memória transacional
- Existem pesquisas nesta área. Mas depuradores inevitavelmente atrasam a execução de um código e isso pode ser fatal para concorrência

Monitoração

- Monitorar durante a execução do código (sem interromper)
- Entender melhor o comportamento do código em execução e saber se a estratégia é a melhor
- Mecanismos clássicos podem ser mais adequados

Como monitorar ?

- Monitorar de maneira pouco intrusiva
- Um monitor não pode adicionar sincronização e alterar o comportamento do código
- Criação de uma chamada de sistema capaz de receber informações sobre transações
- Escrita das informações sobre eventos como *commits*, *rollbacks*, *retries* de cada transação no *procfs*
- Outras ferramentas podem utilizar esta informação

Implementações de memória transacional

- Algumas pesquisas sugerem modificações no hardware (HTM).
- Modificações no compilador C++ (Intel)

- Suporte nativo em algumas linguagens como Haskell e Clojure
- Bibliotecas e extensões para diversas linguagens como C, C++, Java, C#

RSTM, TinySTM, TL2 (Sun), SXM (Microsoft), DSTM2 (Sun)

Rochester Software Transactional Memory (RSTM)

- Biblioteca em C++ composta atualmente por 13 diferentes implementações de memória transacional
- Granularidade de detecção de conflito: objeto

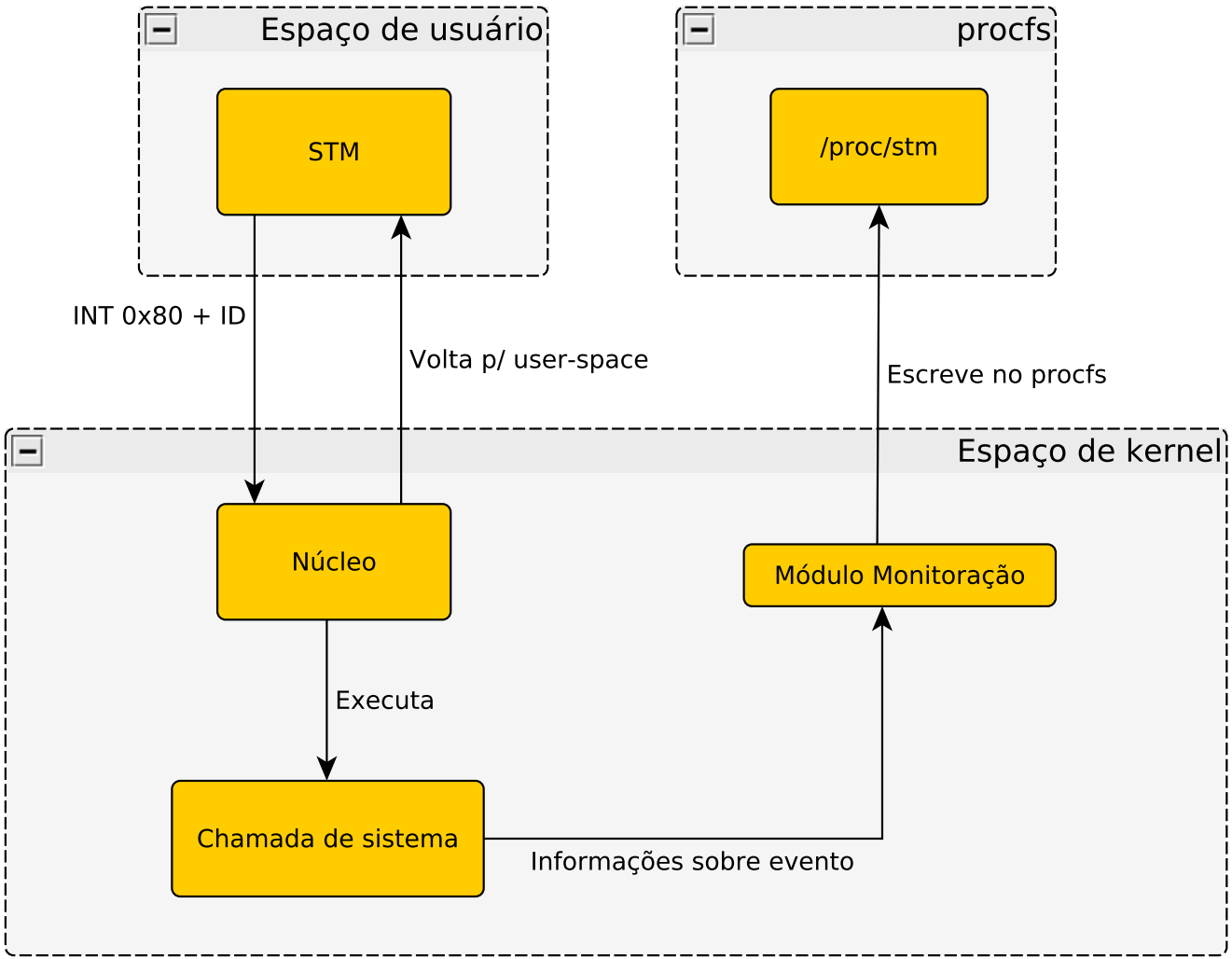
Criação de uma nova chamadas de sistema

- Comunicação entre espaço de usuário e espaço de *kernel*
- Responsável em receber informações sobre o evento relacionado à memória transacional

Nova entrada no procfs

- Sistema de arquivos virtual utilizado para comunicação entre espaço de usuário e espaço de *kernel*
- Neste trabalho apenas o *kernel* envia informações para o espaço de usuário

Arquitetura



Características

- Notificação de novo evento por meio de uma chamada de sistema
- Eventos monitorados: transações iniciadas, finalizadas e abortadas
- Define um padrão para uma chamada de sistema que permite que qualquer implementação de STM seja adaptada com facilidade
- Maior sobrecarga computada como tempo de sistema
- Fácil adaptação para outros eventos

stm_struct

- Representa os eventos monitorados

```
struct stm_struct {
  int tx_id;
  atomic_t commited;
  atomic_t aborted;
  atomic_t started;
}
```

task_struct

- Representa *threads* e processos no SO
- Passa a receber um vetor de stm_struct
- Esta mudança impossibilita usar o TMEM como módulo

Nova chamada de sistema

- Incrementa os contadores de eventos para o processo que fez a chamada
- *thread-safe* e não-bloqueante
- Esta mudança impossibilita usar o TMEM como módulo (exportar syscalltable)

Informações de saída

- Novo arquivo /proc/[pid]/stm_events
- Escrita no procfs não foi feita da maneira convencional (tgid_base_stuff)
- Informações agrupadas por bloco atômico
- Esta mudança impossibilita usar o TMEM como módulo

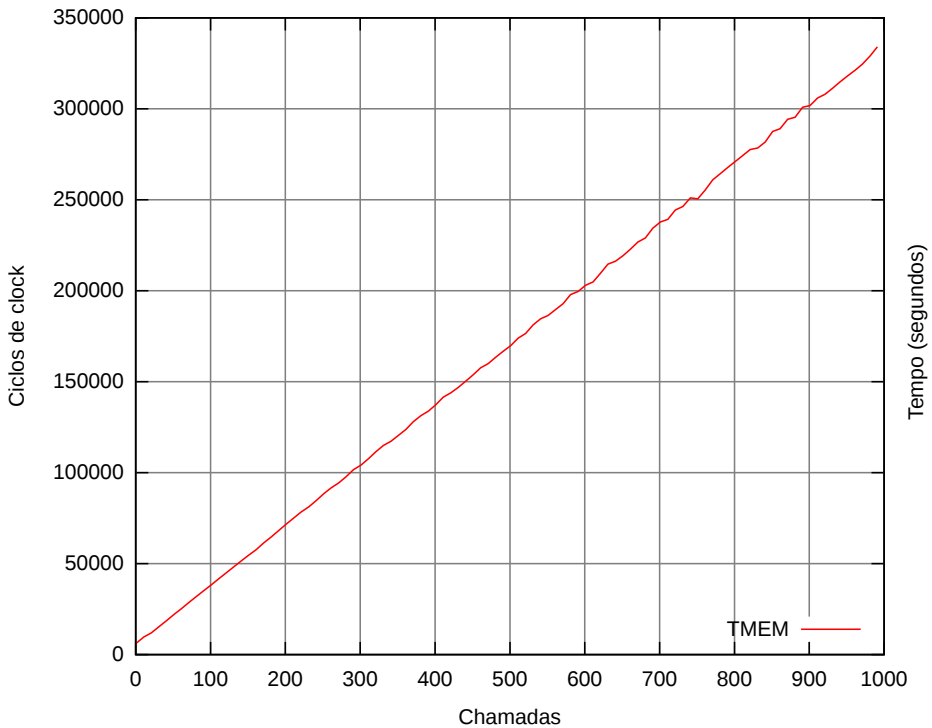
Alterações na RSTM

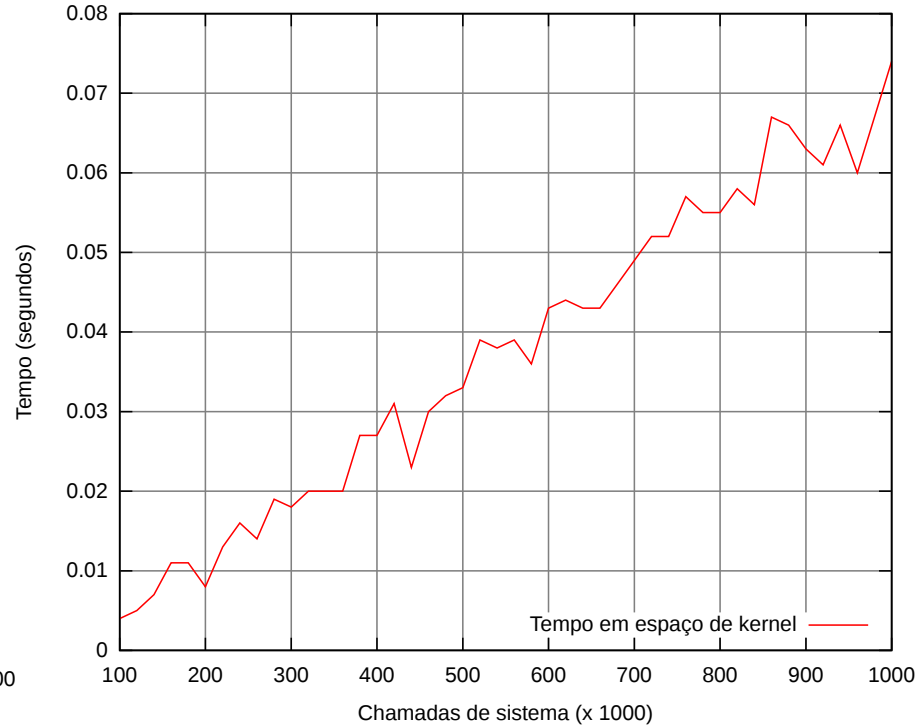
- Blocos atômicos passam a receber um identificador
- Funções em espaço de usuário para notificar o SO (register_transaction_*(id))
- Quando um novo evento ocorre é feita uma chamada de sistema

Avaliação da chamada de sistema (espaço de sistema)

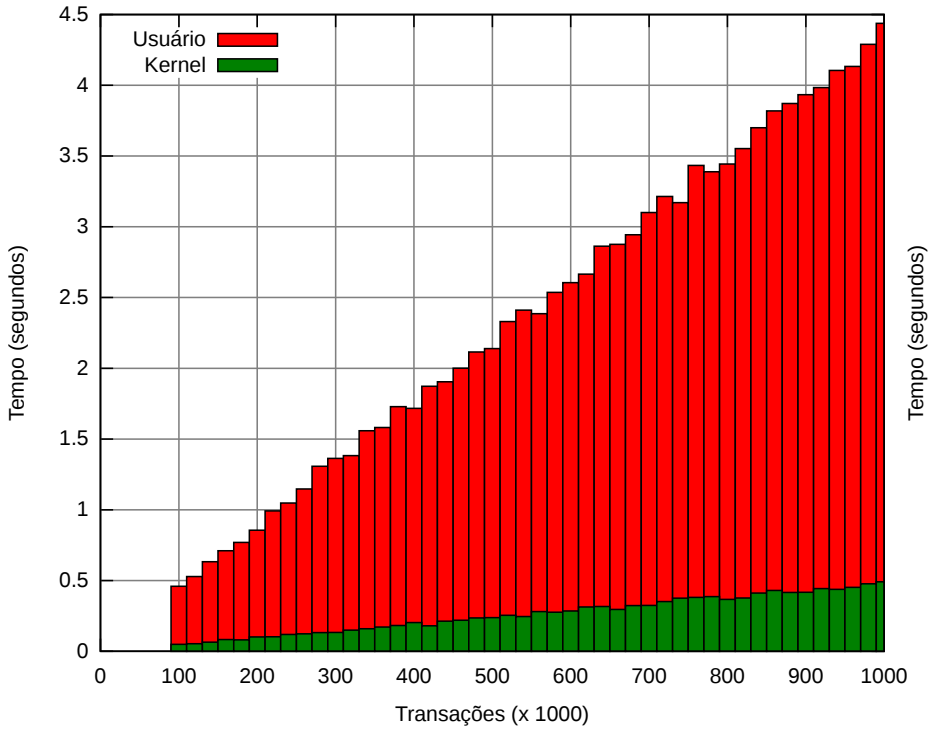
- Ciclos de *clock* (RDTSC)
- Tempo

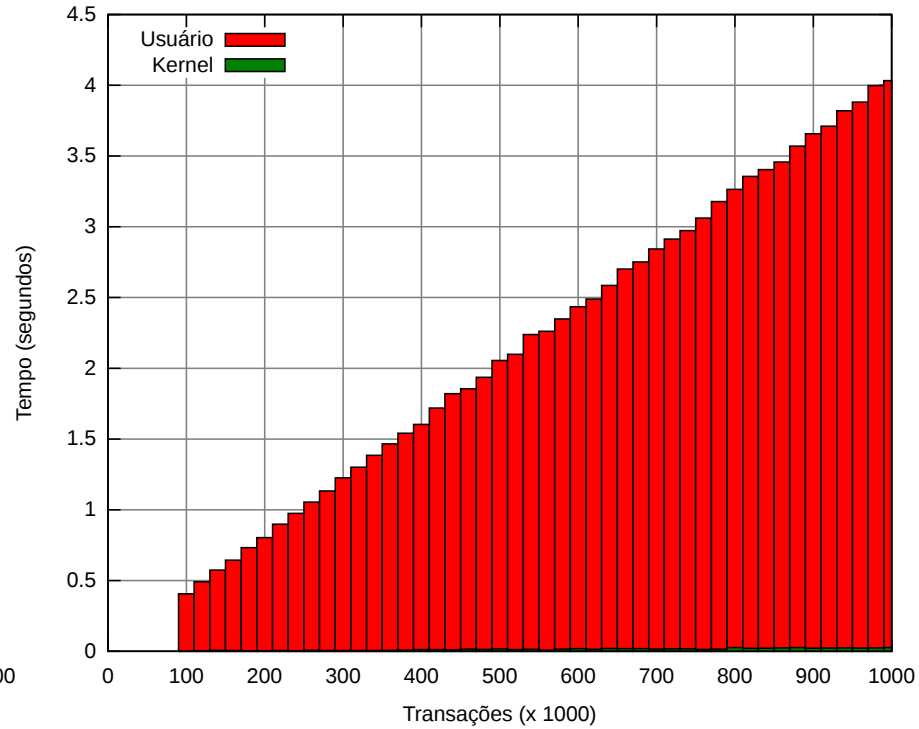
Ciclos de *clock* e duração em espaço de sistema





TMEM - Espaço de usuário x Espaço de sistema





Conclusões

- TMEM pode auxiliar a encontrar transações problemáticas
- Maior intrusão computada como tempo de sistema
- Pode auxiliar a tomar decisões sobre quais estratégias são mais adequadas no uso e na criação de sistemas de memória transacional
- Apesar da intrusão ser baixa, o uso em produção pode prejudicar o sistema monitorado
- Pode ajudar a identificar uso errado da API de memória transacional

Trabalhos futuros

- Admitir novos eventos como falsos conflitos e mais informações como posições de memória conflitantes
- Uso das informações para tomada de decisão em tempo de execução (ex: mudar gerenciador de contenção)
- Adaptar para linguagens executadas pela JVM e monitorar via JMX
- Possíveis melhorias para o TMEM - limitação quantidade de blocos atômicos, atribuição automática de identificadores, níveis de monitoração