UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

APERFEIÇOAMENTO DA BIBLIOTECA LIBRASTRO DE GERAÇÃO DE RASTROS DE EXECUÇÃO DE PROGRAMAS

TRABALHO DE GRADUAÇÃO

Rafael Keller Tesser

Santa Maria, RS, Brasil

2007

APERFEIÇOAMENTO DA BIBLIOTECA LIBRASTRO DE GERAÇÃO DE RASTROS DE EXECUÇÃO DE PROGRAMAS

por

Rafael Keller Tesser

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de
Bacharel em Ciência da Computação

Orientador: Prof. Dr. Benhur de Oliveira Stein (UFSM)

Trabalho de Graduação Nº 232
Santa Maria, RS, Brasil

2007

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

APERFEIÇOAMENTO DA BIBLIOTECA LIBRASTRO DE GERAÇÃO DE RASTROS DE EXECUÇÃO DE PROGRAMAS

elaborado por
Rafael Keller Tesser

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Prof. Dr. Benhur de Oliveira Stein (UFSM)
(Presidente/Orientador)

Profª Drª Andrea Schwertner Charão

Prof. Antonio Marcos de Oliveira Candia (UFSM)

Santa Maria, 02 de Março de 2007.

AGRADECIMENTOS

Gostaria de agradecer principalmente ao meu pai que sempre teve o desejo de que eu cursasse o ensino superior. E, além disso, sempre se esforçou em dar-me o apoio necessário para que isso fosse possível. Gostaria de agradecer também a minha irmã que teve que agüentar morar só comigo durante mais de quatro anos.

Também gostaria de agradecer a quem me ajudou na realização desse trabalho principalmente ao professor Benhur Stein, meu orientador. Obrigado por ter paciência comigo quando entregava as coisas em cima do prazo! Agradeço também aos professores que no decorrer da graduação me ajudaram a adquirir parte do conhecimento utilizado na realização deste trabalho.

Também não posso deixar de agradecer àqueles que tiveram paciência comigo enquanto atrapalhava a realização de seus trabalhos.

É isso então. Não quis citar muitos especificamente para evitar a reclamação de outros que poderiam ser esquecidos.

“O que sabemos é uma gota, o que ignoramos é um oceano.” — ISAAC NEWTON

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

APERFEIÇOAMENTO DA BIBLIOTECA LIBRASTRO DE GERAÇÃO DE RASTROS DE EXECUÇÃO DE PROGRAMAS
Autor: Rafael Keller Tesser
Orientador: Prof. Dr. Benhur de Oliveira Stein (UFSM)
Local e data da defesa: Santa Maria, 02 de Março de 2007.

Nos dias atuais, há uma grande demanda por poder de processamento. Para suprí-la, é comum a utilização de máquinas com múltiplos processadores. É normal também a formação de sistemas distribuídos com múltiplas máquinas interligadas através de uma rede. Para aproveitar completamente a capacidade de máquinas multiprocessadas utilizam-se programas com vários fluxos de execução ou *threads*. O acompanhamento da execução de programas em que várias partes executam ao mesmo tempo, em *threads* ou máquinas diferentes, não é uma tarefa trivial. A geração de rastros de execução é uma técnica de monitoramento altamente aplicável para a depuração e análise de desempenho de aplicações *multithread* e distribuídas. A libRastro é uma biblioteca e um conjunto de ferramentas que servem para instrumentar programas para que gerem rastros de execução, possibilitando a visualização *post-mortem* de seu comportamento. Nesse tipo de ferramenta é desejável um alto nível de precisão na datação de eventos e pouca modificação no comportamento original do programa instrumentado. Esse trabalho descreve o aperfeiçoamento dessa biblioteca, no que diz respeito à datação de eventos e à gerência de *buffers* de rastros. Quanto à datação, é descrita a implementação de um mecanismo mais otimizado e preciso que o original. Já no que diz respeito aos *buffers*, foi implementado o uso de um *buffer* compartilhado entre as *threads*. Para isso utilizou-se uma instrução em nível de montagem, que garante a exclusão mútua de maneira eficiente.

Palavras-chave: Monitoramento, geração de rastros, mutithread, análise de execução.

ABSTRACT

Graduation Work
Graduate Program in Computer Science
Federal University of Santa Maria

ENHANCEMENT OF THE LIBRASTO EXCUTION TRACE GENERATION LIBRARY

Author: Rafael Keller Tesser
Advisor: Prof. Dr. Benhur de Oliveira Stein (UFSM)

Nowadays there's a great need of processing power. It's common the use of multiprocessor machines to satisfy this need. It's used also for this purpose to build distributed systems composed of various machines connected through a network. Multithreaded programs are used to take total advantage of the multiprocessor machines capacity. The following of the execution of programs which execute various parts at the same time isn't an easy task. The execution trace generation is a well suited technique for multithreaded and distributed software debugging and performance analysis. The libRastro is a library and a toolkit which can be used to instrument programs to generate execution traces. These traces are useful to perform a *post-mortem* visualization of its behavior. In this kind of tool it's desirable a high precision level in the events time stamp and few modification of the instrumented software's original behavior. This paper describes the enhancement of this library's time stamping and buffer management. Concerning to the time stamps, it's described the implementation of a method of obtaining it which is better optimized and accurate than the original. Concerning to the buffers, it has been implemented the use of a shared buffer among the threads. To do it, an assembly instruction wich guarantees mutual exclusion in an efficient way was used.

Keywords: monitoring, trace generation, multithread, execution analysis.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

FUT	Fast User Trace
FKT	Fast Kernel Trace
GNU	GNU Is Not Unix
GCC	GNU Compiler Collection
PThreads	POSIX Threads

SUMÁRIO

1 INTRODUÇÃO

A demanda por capacidade de processamento, principalmente por grandes empresas, centros de pesquisa e órgãos governamentais, gerou a necessidade da criação de arquiteturas computacionais de alto desempenho. Duas soluções comumente utilizadas para suprir essa necessidade são o uso de máquinas multiprocessadas e a interligação de vários computadores através de uma rede de alta velocidade, podendo formar o que é chamado "sistema distribuído".

Para tirar proveito total da capacidade de máquinas com vários núcleos de processamento, utiliza-se programas com vários fluxos de execução chamados *threads*. Isso possibilita que partes diferentes do programa executem simultaneamente em núcleos diferentes.

Analisar o que acontece em mais de uma *thread* em tempo de execução é uma tarefa complicada. Isso se torna ainda mais inadequado quando se quer analisar o desempenho do programa, caso em que não se deve interferir demais no comportamento normal do programa, ou seja, a ferramenta utilizada não pode ser muito intrusiva. Por isso, é comum quando se trata da análise e depuração de programas *multithread*, a utilização de técnicas que permitam uma visualização *post-mortem* da execução, ou seja, após o término da mesma.

Uma técnica muito difundida que permite a análise do comportamento de programas *multithread* é a geração de rastros de execução. Esses rastros são registros de eventos que ocorrem durante a execução do programa e que podem ser utilizados para a sua depuração ou análise de desempenho possibilitando, por exemplo, a detecção de gargalos de desempenho.

A libRastro (SILVA; STEIN, 2002) é uma biblioteca de geração de rastros que possibilita a instrumentação de programas, através da inserção de funções de registro de eventos.

Durante a execução esses eventos gravam rastros em um arquivo. Esses rastros podem ser utilizados para visualizar aspectos relevantes da execução do programa e podem ser convertidos para um formato utilizado por alguma ferramenta de visualização de execução.

A datação de eventos em uma ferramenta de rastreamento deve ser o mais precisa possível e deve perturbar minimamente o comportamento do programa. Apesar disso, a técnica utilizada pela libRastro em sua versão original é pouco otimizada e imprecisa.

Além disso, para evitar o uso de mecanismos de exclusão mútua de alto nível, a cada vez que se registra um rastro no *buffer* de memória, a libRastro original utiliza um *buffer* individual para cada *thread*. Em contrapartida, tem-se que arcar com a gerência desses vários *buffers*.

Esse trabalho apresenta soluções, utilizando mecanismos de baixo nível, que possibilitam uma melhor datação de eventos e o uso de um único *buffer* de registro de eventos sem a necessidade de mecanismos de exclusão mútua de alto nível, que são altamente intrusivos.

No capítulo seguinte, é apresentada um contextualização sobre o assunto de geração de rastros de execução, bem como trabalhos em que se baseiam as soluções adotadas, e algumas ferramentas utilizadas na implementação dessas. No capítulo 3 são descritas as melhorias implementadas. O capítulo 4 descreve a implementação desse trabalho. Uma avaliação do trabalho realizado, comparado à versão original da libRastro é mostrada no capítulo 5. Por fim, o capítulo 6 encerra com a conclusão do trabalho e sugestões de trabalhos futuros, relacionados ao mesmo.

2 CONTEXTUALIZAÇÃO

Nesse capítulo serão abordados alguns tópicos relacionados ao trabalho a ser realizado. Primeiramente é apresentada uma pequena introdução sobre a geração de rastros de execução. Após é apresentada a libRastro em seu estado original. A seguir são apresentadas as ferramentas *Fast User Trace* e *Fast Kernel Trace*, nas quais se baseia boa parte desse trabalho. Por último, são apresentadas as ferramentas utilizadas na implementação desse trabalho.

2.1 Geração de Rastros de Execução

A geração de rastros de execução é uma técnica que possibilita a análise de execução de programas após seu término (*post-mortem*). Essa técnica é utilizada para a depuração e análise de desempenho de programas *multithread*, distribuídos e paralelos. A motivação para isso é que a visualização da execução das várias *threads*, ou dos vários processos que inclusive podem estar em nodos diferentes, é muito difícil de ser feita de maneira adequada enquanto o programa ainda está executando. Além disso, principalmente quando se trata de análise de desempenho, deseja-se que o comportamento normal do programa seja modificado ao mínimo pelo uso da ferramenta de análise. Esse fato faz com que uma análise *post-mortem* seja mais adequada.

Essa técnica consiste na geração de eventos os quais registram rastros de execução em pontos de interesse para a análise desejada. A inserção desses eventos é chamada de instrumentação.

Quando um programa instrumentado é executado ele gera rastros, que posteriormente são utilizados para a análise *post-mortem* da execução do programa, podendo-se para isso usar ferramentas de visualização de execução.

2.2 A Ferramenta libRastro

A libRastro (SILVA; STEIN, 2002) é uma biblioteca genérica de geração de rastros para visualização de programas. Sua principal utilidade é a depuração e análise de desempenho de programas *multithread*, distribuídos e paralelos. Essa ferramenta pode ser utilizada para analisar programas em linguagem C e que utilizem a biblioteca de *threads* pthreads, no sistema operacional Linux.

Para utilizar a libRastro deve-se instrumentar o programa a ser analisado incluindo eventos que gerarão os rastros, nos pontos de interesse para a análise desejada. Por exemplo, pode-se gerar eventos para identificar a criação e o término de uma *thread*, bem como pode-se querer identificar eventos de comunicação entre *threads* ou processos.

A inserção desses eventos é feita através de chamadas de funções. Essas funções podem ter os seguintes formatos, de acordo com os rastros que se deseja armazenar:

- `rst_event(u_int_16_t type)`, que recebe como argumento apenas um valor que identifica o tipo de evento;
- `rst_event_?...(u_int_16_t type,...)`, que recebe como argumento, além do tipo do evento, outros valores a registrar, cujo tipo acorda com os caracteres do nome da função (representados por "?"), que podem ser:
  - `c`: 8 bits;
  - `w`: 16 bits;
  - `i`: 32 bits;
  - `l`: 64 bits;
  - `f`: float;
  - `d`: double;
  - `s`: string;

  "..." indica que pode haver mais elementos "?" diferentes, por exemplo:

  - `rst_event_iicliff(...);`
  - `rst_event_siicicffl(...);`

As funções no segundo formato são geradas dinamicamente, através da utilização de uma ferramenta fornecida juntamente com a libRastro. Essa ferramenta recebe os arquivos fonte, preprocessa-os, utilizando a ferramenta GCC, depois identifica as chamadas às funções de registro de eventos, e passa o resultado para um programa que gera um arquivo fonte contendo as funções geradas, e um de cabeçalho a ser incluído no programa original.

Além disso é necessário chamar as funções `rst_init()` e `rst_finalize()` no início e no fim de cada *thread* ou processo. Dessa forma será criado um *buffer* para cada *thread* e cada uma dessas gravará seus rastros em um arquivo diferente. A função de inicialização, recebe como argumento dois números que identificarão o arquivo de rastros gerado. A figura 2.1 mostra um exemplo de programa instrumentado utilizando libRastro. Quando é chamada uma função de evento, ela grava em um *buffer* em memória, um *ti-*

```
int main(int argc, char *argv[])
{
   rst_init(10, 20);
   rst_event(1);
   rst_event_lws(2, 1, 2, "3");
   rst_event_wlsfcd(3, 1, 2, "3", 4, '5', 6);
   rst_event_iwlsifcd(4, 1, 2, 3, "4", 5, 6, '7', 8);
   rst_finalize();
   return 0;
}
```

Figura 2.1: Exemplo de programa instrumentado usando a libRastro

mestamp, o número que identifica o tipo do rastro, e os dados recebidos como argumento. Caso uma *thread* descubra que o *buffer* está cheio, essa grava seu conteúdo no arquivo de rastros e o esvazia.

Caso se esteja monitorando uma aplicação distribuída, é necessário executar um programa de sincronização, antes e depois da execução do programa monitorado. Com isso será gerado um arquivo de sincronização, utilizados mais tarde para corrigir os tempos de registro dos rastros contidos nos arquivos de rastro.

Para a leitura dos arquivos de rastros, deve-se criar um programa utilizando funções fornecidas pela libRastro para a leitura dos arquivo de rastros, e sua tradução para um formato legível. Esse programa receberá como argumento os arquivos de rastros e o de sincronização, se houver. Um exemplo de programa de leitura de rastros é mostrado na figura 2.2.

Ao utilizar esse programa para a leitura do arquivo gerado pelo exemplo da figura 2.1, têm-se como saída os rastros decodificados, mostrados na figura 2.3. Para cada evento que registrou rastros é mostrado seu tipo, datação e os dados registrados agrupados por tipo de dado.

A figura 2.4 mostra o processo de utilização da libRastro, exceto o processo de sincronização necessário para rastrear programas distribuídos. Primeiramente têm-se os arquivos fonte do programa instrumentados. Então utiliza-se uma ferramenta que gera dinamicamente as funções de registro de evento chamadas pelo usuário e cria um arquivo fonte contendo-as e um de cabeçalho. Compilando-se os arquivos instrumentados originais, juntamente com o de funções de eventos e com a libRastro, obtém-se um arquivo executável. Ao executar esse arquivo são gerados arquivos de rastro em um formato ilegível.

Para a leitura desse arquivo de rastro cria-se um programa que utiliza funções fornecidas pela libRastro para ler o arquivo, e transformar seus dados para um formato legível.

A utilização de mecanismos de sincronização como *mutex*, por exemplo, para exclusão mútua no acesso a um *buffer* de rastros compartilhado faria com que as *threads* ficassem freqüentemente bloqueadas. Isso representaria uma modificação significativa no comportamento normal do programa. Devido a esse fator, a biblioteca original utiliza um *buffer* para cada *thread* mas, em contrapartida tem que arcar com os custos para a gerência desses *buffers*.

A datação dos rastros é feita utilizando a função `gettimeofday` na versão original da biblioteca. O uso dessa função acarreta uma chamada ao sistema, o que pode perturbar o comportamento normal da aplicação. Além disso, os valores obtidos através desse método são imprecisos.

Se o programa a ser rastreado for distribuído é utilizado um método estatístico para corrigir os tempos registrados no arquivo de rastros (MAILLET; TRON, 1995).

2.3 Fast Kernel Trace e Fast User Trace

Fast Kernel Trace (FKT) (RUSSEL; CHAVAN, 2001) é uma ferramenta de análise de desempenho para Linux. Ela funciona através da instrumentação do núcleo (*kernel*) do sistema operacional pela inserção de sondas (macros de pontos de teste) nos componentes do *kernel*, que geram rastros de execução em nível de sistema operacional. Embora a primeira versão de FKT limite o tamanho dos rastros à quantidade de memória disponível,

```
int main(int argc, char *argv[])
{
    rst_file_t arq_rst;
    rst_event_t ev;
    if(rst_open_file(argv[1], &arq_rst, "arquivo_sync",
                                          10000) == -1){
        perror("Impossivel abrir arquivo de rastros!\n");
        exit(1);
    }
    while(rst_decode_event(&arq_rst, &ev)){
        rst_print_event(&ev);
    }
    rst_close_file(&arq_rst);
    return 0;
}
```

Figura 2.2: Exemplo de programa para leitura de um arquivo de rastos

```
type: 1 ts: 1172458670124528
type: 2 ts: 1172458670124530
u_int64_ts-> (1)
strings-> (3)
u_int16_ts-> (2)
type: 3 ts: 1172458670124533
u_int64_ts-> (2)
strings-> (3)
floats-> (4.000000)
u_int16_ts-> (1)
u_int8_ts-> (5)
doubles-> (6.000000)
type: 4 ts: 1172458670124535
u_int64_ts-> (3)
strings-> (4)
floats-> (6.000000)
u_int32_ts-> (1) (5)
u_int16_ts-> (2)
u_int8_ts-> (7)
doubles-> (8.000000)
```

Figura 2.3: Rastros decodificados pelo programa de exemplo de leitura de dados

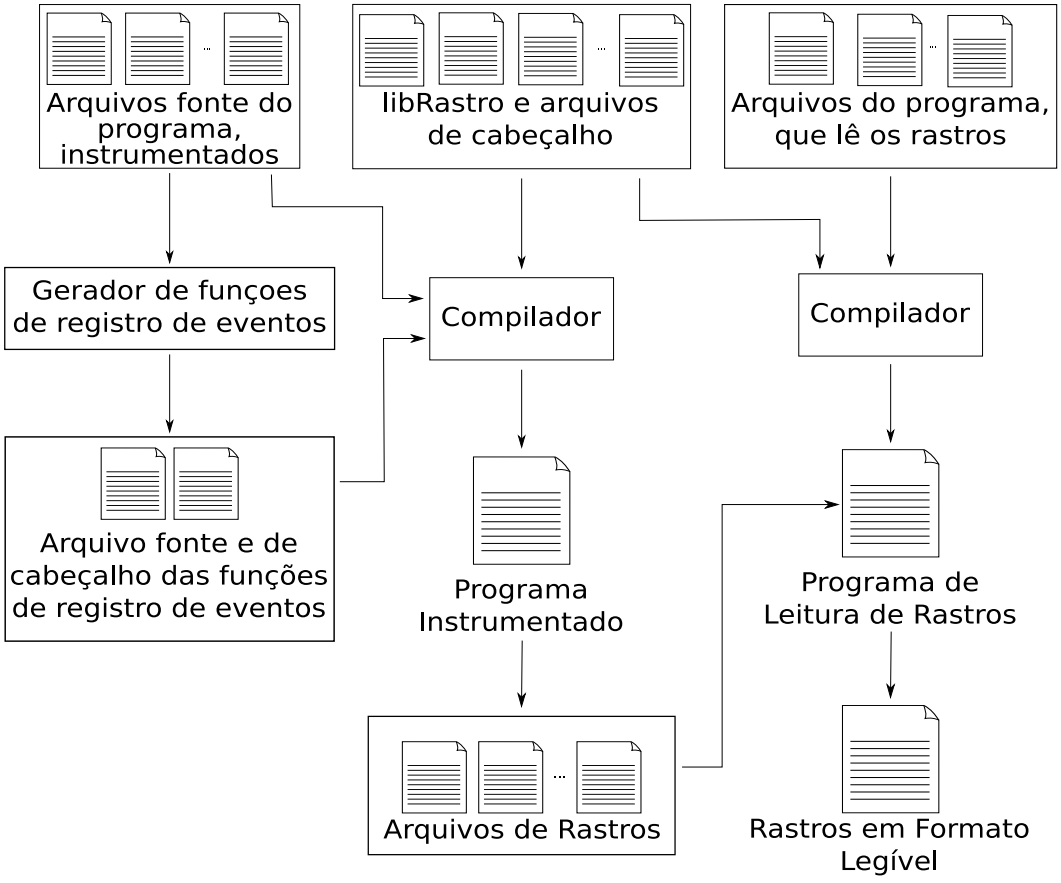

Figura 2.4: Processo de utilização da libRastro

foi desenvolvida uma nova versão que elimina essa limitação (SAMUEL THIBAULT, 2005).

Fast User Trace (FUT) (SAMUEL THIBAULT, 2005) é uma ferramenta baseada na FKT, que serve para gerar rastros de execução em programas em nível de usuário.

Essas ferramentas fazem uso de um programa chamado `fkt_record`, responsável por gravar os rastros gerados. Também há o programa `fkt_print` que lê o arquivo de rastros que é humanamente ilegível e o converte para um formato adequado para a visualização.

Existe também um conjunto de ferramentas de rastreamento multi-nível, capaz de rastrear tanto *threads* do usuário quanto de núcleo, bem como as híbridas, utilizando FKT e FUT (DANJEAN; NAMYST; WACRENIER, 2005). Tal trabalho constou da instrumentação da biblioteca de *threads Marcel* que faz parte do ambiente de programação paralela PM^2.

Duas técnicas utilizadas por FKT e FUT são interessantes para este trabalho. A primeira diz respeito à datação de eventos e a segunda à gerência dos *buffers*. A implementação desses é feita em nível de montagem, sendo que essas ferramentas foram implementadas para serem utilizadas em arquiteturas do tipo Intel Pentium.

A datação é feita utilizando o *Timestamp Counter*, que é um registrador presente nos processadores da família Pentium e compatíveis. Esse registrador armazena o número de ciclos de relógio decorridos desde que o processador foi ligado, utilizando 64 bits para isso. Dessa maneira, é possível datar os eventos, com alta precisão e sem a necessidade de chamadas ao sistema. Em sistemas com freqüência de relógio a partir de um Gigahertz é possível ter-se uma resolução de nanossegundos.

Quanto à gerência de *buffers*, é usada a instrução `cmpxchg` (*compare and exchange*). Com ela há como permitir a exclusão mútua na alocação de espaço no *buffer* sem a utilização de dispendiosos mecanismos de alto nível.

2.4 Ferramentas Utilizadas

Essa seção apresenta as ferramentas utilizadas para a implementação desse trabalho.

A libRastro é implementada na linguagem de programação C. A biblioteca de *threads Pthreads* também foi utilizada. Além disso, foi utilizado a funcionalidade *inline assembly*, suportada pela ferramenta de compilação GCC (*GNU Compiler Collection*), para integrar

trechos de código em linguagem de montagem, aos programas em linguagem C.

A biblioteca *pthreads* e *GCC Inline Assembly* são apresentados respectivamente nas subseções seguintes.

2.4.1 Pthreads

A utilização de *threads* ou fluxos de execução, é uma técnica de programação muito utilizada atualmente. Elas permitem dividir um processo em vários fluxos que executam concorrentemente, compartilhando o mesmo espaço de endereçamento.

Processos podem ficar bloqueados freqüentemente, esperando por uma operação de entrada e saída por exemplo. Caso se utilize *threads*, enquanto uma está bloqueada as outras podem continuar executando. Outra vantagem do uso de *threads* é tirar proveito de máquinas com mais de um processador, nesse caso, as *threads* podem executar em paralelo, em processadores diferentes. Ambos esses aspectos geram ganhos de desempenho na execução. Para maiores informações sobre *threads* pode-se consultar um livro de sistemas operacionais, como o de A. S. Tanenbaum (TANENBAUM, 2001).

A libRastro utiliza a biblioteca de *threads POSIX threads*, que é um padrão para os sistemas Unix. Implementações que seguem esse padrão são referidas como POSIX *threads* ou Pthreads.

2.4.2 GCC Inline Assembly

Para a elaboração desse trabalho necessita-se o uso de alguns trechos de código em linguagem de montagem. A ferramenta de compilação *GNU Compiler Collection* possui uma funcionalidade chamada *Inline Assembly* que possibilita o uso de dessa linguagem em meio ao código em linguagem C.

A inserção do código de montagem é feita utilizando o comando `asm()` ou `__asm__()` (STALLMAN; GCC DEVELOPER COMMUNITY, 2005; BOLDYSHEV; RIDEAU, 2006), sendo que dentro dos parênteses vai o código *assembly*. A figura 2.5 mostra um exemplo de seu uso.

O código que está após o primeiro ":" indica as variáveis do código em C usadas como saídas e o que está após o segundo, indica as entradas. Para referenciar essas variáveis, utiliza-se o formato "%#", onde "#" é um número, que começa em zero representa a posição da declaração da variável nas entradas e saídas. No exemplo dado "%0" representa a primeira variável de saída que é "b" e "%1" representa a variável de entrada "a". O código

```
int a = 1, b = 0;

asm("addl %1, %0"
    : "=r" (b)
    : "r" (a), "0" (b)
);
```

Figura 2.5: Exemplo do comando `asm`

que vem antes do nome da variável entre parênteses, contem uma letra, a qual é precedida de um sinal de igualdade para as saídas. Essa letra indica onde deve estar armazenada a variável, por exemplo em um registrador (r), em memória (m), ou qualquer um dos dois (rm).

É interessante ressaltar que é utilizada a sintaxe AT&T, e não a da Intel. Uma das principais diferenças é que na sintaxe da Intel o operando de destino vem antes e o de origem depois, enquanto na AT&T a ordem é inversa.

3 DESCRIÇÃO DAS MELHORIAS IMPLEMENTADAS

O mecanismo utilizado pela libRastro para a datação de evento é pouco otimizado e preciso, conforme foi relatado. Além disso, a utilização de *buffers* separados para cada *thread* faz com que se tenha que arcar com os custos de seu gerenciamento.

Esses dois problemas precisam ser atacados de maneira a melhorar a precisão da datação e também diminuir a perturbação introduzida pela instrumentação, ao comportamento normal do programa. Esses dois aspectos são muito importantes, principalmente quando se pretende realizar análises de desempenho.

Esse trabalho implementa modificações na biblioteca de rastreamento, de maneira a otimizar a datação e a gerência dos *buffers* e aumentar a precisão dos mesmos.

As ferramentas Fast Kernel Trace e Fast User Trace foram apresentadas no capítulo 2, seção 2.3. Essas utilizam mecanismos de baixo nível tanto para a datação como para garantir a exclusão mútua no acesso a um *buffer* compartilhado pelas *threads*.

Inicialmente pretendia-se a integração dessas ferramentas à libRastro. No entanto isso mostrou-se inviável, conforme é discutido no capítulo 4 que trata da implementação realizada. Devido a esse fato, resolveu-se estudar as técnicas de datação e alocação de espaço no *buffer* usada por FKT e FUT, e posteriormente implementá-las de forma similar na libRastro.

O objetivo do uso dessas técnicas é otimizar e melhorar a precisão da datação de eventos. Também foi implementado um *buffer* compartilhado entre as *threads* para gravar os rastros, a fim de diminuir o custo da gerência de *buffers*.

Para implementar o compartilhamento do *buffer*, foi necessário mudar vários aspectos do programa. Um exemplo disso é a necessidade de um cuidado maior quando se grava os dados para o disco. As modificações necessárias, são descritas na seção 3.2.

3.1 Datação

Para a implementação da datação utilizou-se o valor lido do registrador *Timestamp Counter* (TSC), que pode ser lido utilizando a instrução `rdtsc`. O TSC armazena a contagem de ciclos de processador que ocorreram desde que todos os núcleos de processamento da máquina foram inicializados. Para isso é utilizado um valor de 64 bits.

A instrução coloca os 32 bits mais significativos do TSC no registrador EDX, e os 32 menos significativos no registrador EAX. Após obtidos esses valores, é feito o cálculo do tempo em microssegundos. Para isso divide-se o número de ciclos pela freqüência de operação do processador dividida por 1000. Em um processador com freqüência de relógio de 1 gigahertz ou mais, seria possível calcular o tempo com resolução de nanossegundos.

3.2 Gerência de Buffers

A implementação dos *buffers* de rastro pode ser feita de duas maneiras: cada *thread* possui um *buffer* individual onde registra seus eventos ou existe apenas um *buffer* compartilhado entre as *threads*. Cada uma dessas alternativas possui suas vantagens e desvantagens.

A libRastro original utiliza *buffers* individuais para cada *thread*. Algumas das vantagens desse esquema são:

- a criação e destruição de um *buffer* por cada *thread*, incluindo alocação e liberação de memória pode ser custosa;
- é necessário utilizar mecanismos como as funções `pthread_setspecific` e `pthread_getspecific` da bilioteca *Pthreads* para possibilitar o acesso a variáveis específicas de cada *thread*.

Uma importante vantagem dessa técnica é:

- não há a necessidade do uso de mecanismos de exclusão mútua no acesso ao *buffer*, os quais podem modificar consideravelmente o comportamento do programa.

Quanto à utilização um *buffer* compartilhado entre as *threads* algumas das vantagens obtidas são:

- simplificação da gerência devido a haver apenas um *buffer*;

- criação e destruição de apenas um *buffer*, ou seja, apenas uma alocação é feita;
- não necessita de mecanismos especiais para obter a localização do *buffer*.

No entanto, há uma grande desvantagem:

- necessidade de mecanismos de exclusão mútua para evitar problemas relacionados à concorrência no acesso ao *buffer*.

A solução adotada pra melhorar a gerência de *buffers* de rastro, foi a utilização da instrução *cmpxchg* (*compare and exchange*), para garantir a exclusão mútua na gravação de dados no *buffer* de forma eficiente. Com a utilização dessa instrução deve ser implementado o uso de um único *buffer* compartilhado por todas as *threads* de um processo a ser rastreado.

A instrução *cmpxchg* é usada para sincronizar operações em sistemas que usam múltiplos processadores (Intel Corporation, 2005a). Essa instrução requer três operandos: um operando de origem em um registrador, outro no registrador EAX, e um operando de destino. Se os valores contidos no destino e no registrador EAX forem iguais, o operando de destino é substituído com o valor do outro operando de origem (o que não está em EAX. Caso contrário, o valor original do operando de destino é carregado no registrador EAX. A figura 3.1 ilustra o funcionamento do comando *cmpxchg* `destino, RegOrig`.

Em sistemas com múltiplos processadores *cmpxchg* pode ser combinado com o prefixo `lock` para realizar a operação de comparação e troca atomicamente.

A alocação de espaço no *buffer* pode ser feita então conforme os passos mostrados na figura 3.2. No passo 1, copia-se o valor do ponteiro que aponta para a posição atual no *buffer*, carregando-o no registrador EAX. No passo 2, calcula-se a nova posição, levando em conta o tamanho dos dados a serem gravados. Em 3, utiliza-se a instrução *cmpxchg* para comparar o valor da posição obtido anteriormente com o valor atual do ponteiro e, caso sejam iguais, trocar o valor atual pelo calculado para a nova posição. Caso contrário, a troca não ocorrerá e o processo deverá ser repetido.

Se ocorrer o fato de mais de uma *thread* ler o mesmo valor de ponteiro, a que executar *cmpxchg* primeiro conseguirá alocar espaço no *buffer* naquela posição. Nos outros fluxos, a instrução não fará a troca, pois o ponteiro foi modificado pelo que conseguiu realizar a alocação. Isso garante a exclusão mútua na gravação dos rastros no *buffer*.

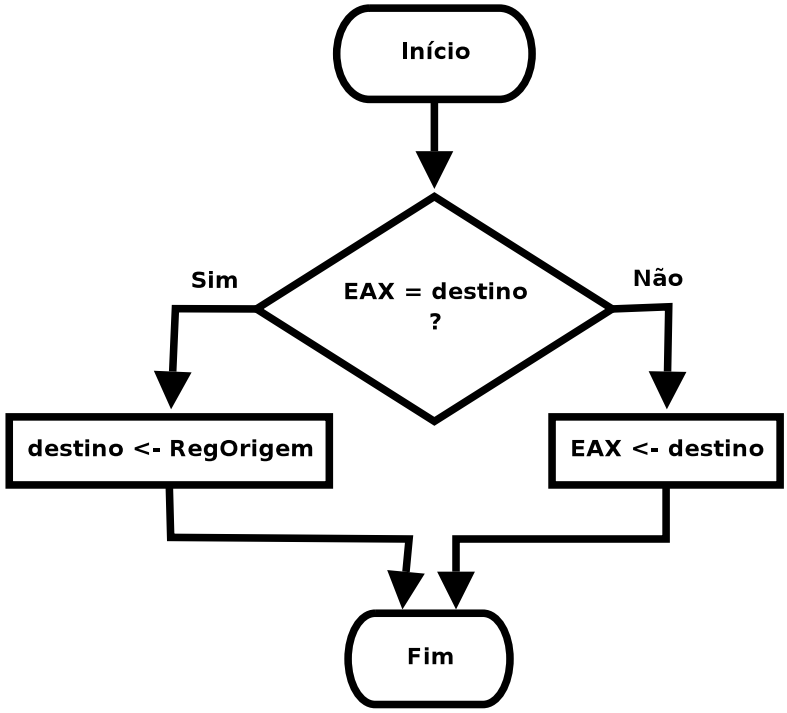

Figura 3.1: Funcionamento da instrução *cmpxchg*

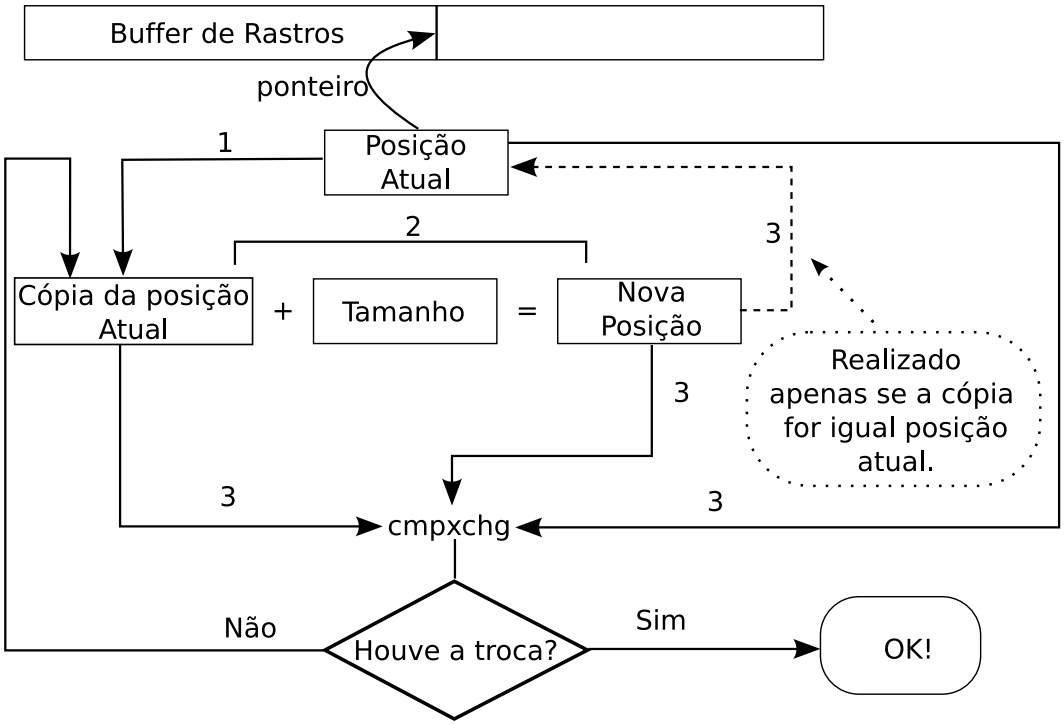

Figura 3.2: Alocação de espaço no *buffer* usando `cmpxchg`

Algumas modificações necessitaram ser feitas para fosse possível utilizar um *buffer* compartilhado para registrar os rastros. Uma dessas é que devido à necessidade de alocação prévia de espaço no *buffer*, é necessário calcular tamanho dos dados a serem gravados.

Como há apenas um *buffer*, também há apenas um arquivo de rastros, portanto, é necessário gravar uma identificação de qual *thread* está registrando cada rastro. Essa mudança gerou a necessidade de modificar as funções utilizadas para a leitura de arquivos de rastros, também.

Outro aspecto é que a gravação dos dados em disco quando o *buffer* estiver cheio pode gerar problemas de concorrência. Para lidar com isso foi necessário implementar um mecanismo que possibilita essa operação ser realizada corretamente, sem corromper os dados e sem que seja preciso que todas as *threads* fiquem bloqueadas quando tentarem registrar rastros, até que a gravação termine.

É interessante ressaltar que, no que diz respeito aos *buffers*, apenas a alocação de espaço nos mesmos utilizando *cmpxchg* e parte do mecanismo de gravação dos dados em arquivo baseou-se em FKT e FUT. As outras modificações, necessárias para suportar o uso dessa técnica de alocação e de um *buffer* compartilhado de rastros, não se baseiam nessas ferramentas. A gravação dos dados em arquivo é feita de maneira parecida com a utilizada nessas ferramentas, no entanto elas utilizam recursos de baixo nível que dificilmente poderiam ser utilizados em um programa escrito em linguagem C.

4 IMPLEMENTAÇÃO REALIZADA

Nesse capítulo será apresentado o estado atual da implementação das melhorias propostas à libRastro. Essas dizem respeito à datação dos rastros e à gerência de *buffers*.

Inicialmente pretendia-se tentar a integração à libRastro das ferramentas *Fast Kernel Trace* (FKT) e *Fast User Trace* (FUT) , apresentadas no capítulo 2, seção 2.3. Infelizmente não foi possível realizar essa tarefa, devido a problemas na execução das ferramentas cujo motivo não pode ser identificado.

Devido a não haver sido encontrada uma maneira de corrigir os erros, optou-se por estudar a implementação de FKT e FUT, de maneira a utilizar técnicas similares às utilizadas por essas para a datação dos rastros e a alocação de espaço em um *buffer* compartilhado entre *threads*. As soluções implementadas apenas baseiam-se em FKT e FUT, tendo sido implementadas de maneira diferente. O principal motivo para isso é que essas ferramentas são implementadas completamente em linguagem de montagem enquanto a libRastro é implementada em linguagem C. Portanto, a implementação das funcionalidades tal qual são feitas em FKT e FUT tornaria muito difícil, se não impossível, a integração com a libRastro original.

Após o estudo do FKT e FUT, foi necessário o estudo da libRastro com o objetivo de entender seu funcionamento e identificar as modificações necessárias para utilizar as novas técnicas.

Nas duas seções seguintes é descrita a implementação dos novos mecanismos de datação e gerência de *buffers*, respectivamente. Ambos utilizam código em linguagem de montagem para o Intel Pentium (Intel Corporation, 2005b,c).

4.1 Datação

Inicialmente foi implementada uma macro que utiliza *inline assembly* para ler o TSC e calcula um tempo em microssegundos. Feito isso, substituiu-se a chamada à função `gettimeofday` do código original, pelo uso dessa macro.

Após, no entanto, decidiu-se integrar a leitura do tempo e a alocação de espaço no *buffer* em uma mesma macro. Fazendo dessa maneira garante-se a ordenação temporal crescente dos rastros no *buffer*. Para diminuir a quantidade de cálculos durante a execução, o tempo passou a ser registrado no formato fornecido pelo TSC, com 64 bits. O cálculo em unidades de tempo (microssegundos ou nanossegundos) passou a ser feito durante leitura do arquivo de rastros. Para que isso fosse possível, a freqüência do processador foi adicionada ao rastro registrado por um evento que é gerado na inicialização da biblioteca.

Um efeito colateral dessa mudança é o aumento de 32 bits no tamanho dos dados registrados a cada evento. Isso ocorre devido a na versão original da libRastro serem utilizados apenas 32 bits para datação.

4.2 Gerência de Buffers

Quanto à gerência de *buffers*, primeiro foi implementada uma macro que executa a operação de alocação de espaço no *buffer*. Essa recebe o ponteiro para a posição atual, um segundo ponteiro onde ela armazena a posição onde inicia o espaço alocado, e o tamanho a ser alocado em *bytes*.

A seguir, foi implementada a datação do evento nessa mesma macro, logo após a alocação do *buffer* conforme o explicado na sessão anterior. Feito isso, passou-se às modificações nas funções de escrita e leitura de rastros e geração dinâmica das funções de registro de eventos.

Quanto à escrita, foi implementada a gravação da identificação da *thread* em cada rastro, necessária devido a haver apenas um *buffer* e conseqüentemente, um arquivo de rastros por processo. Também foram implementadas a integração da alocação e datação utilizando a macro implementada anteriormente. Para que isso fosse possível, foi necessário implementar o cálculo do espaço necessário para armazenar o rastro antes do uso da macro. Isso implicou modificação do programa de geração dinâmica de funções de eventos. O único tipo de dado registrado pela libRastro, do qual não é possível saber

o tamanho antes da execução são as seqüências de caracteres (*strings*). As funções de registro de eventos ficaram responsáveis calcular seu tamanho. Esse então é adicionado ao valor correspondente ao espaço a ser ocupado pelos outros tipos de dados, a datação e cabeçalhos que identificam o tipo de rastro, e os tipos dos dados registrados, calculado na geração das funções.

Na etapa seguinte, foram executadas modificações na leitura dos dados devido às modificações implantadas no formato dos rastros. Essas corresponderam à leitura da identificação das *threads* e a conversão da datação para unidades de tempo. Essa identificação é um valor de 32 bits, retornado pela função `pthread_self`. É interessante notar que esses 32 bits representam um aumento no tamanho do registro de cada evento, em relação à libRastro original.

Também foi modificado a leitura do rastro de inicialização da biblioteca, já que o mesmo passou a conter a freqüência do processador, a qual é utilizada na conversão citada.

O próximo passo foi implementação de um novo esquema de gravação do *buffer* em arquivo. Isso foi necessário por poder haver problemas de concorrência entre as *threads* nessa gravação. O *buffer* foi dividido em páginas, e foi criada uma *thread* responsável por gravar essas páginas no arquivo e depois liberá-las para a escrita novamente. A figura 4.1 apresenta de maneira geral o novo esquema alocação de *buffer*. Essa implementação baseia-se parcialmente nas técnicas utilizadas por FUT e FKT para eliminar a limitação no tamanho *buffer* existente na primeira versão de FKT (SAMUEL THIBAULT, 2005) No entanto, essas ferramentas utilizam funcionalidades em nível de sistema operacional, e em nível de montagem impossíveis de serem utilizadas na libRastro sem que seja necessária sua total reimplementação ou quase.

A detecção de quando uma página está cheia foi implementada na macro de alocação. Caso o final do espaço a ser alocado ultrapasse o limite da página atual, é incrementada uma variável que indica que a página está cheia e não é executado *cmpxchg*.

Para evitar que páginas sejam gravadas em arquivo sem que os rastros tenham sido registrados nas mesmas, elas possuem um contador de escritas. Cada *thread*, antes de tentar alocar espaço no *buffer*, incrementa atomicamente o contador de escritas da página atual. Esse contador é decrementado atomicamente quando é detectado que a página está cheia ou no final do registro dos rastros de um evento. O *buffer* implementado é circular, ou seja, a página seguinte à última é a primeira. Devido a esse fato durante os testes

da implementação descobriu-se a necessidade de testar se o endereço atual está antes do início da página e, caso esteja, decrementar o contador de escritas. Isso ocorre quando a página já foi trocada por outra *thread* por uma situada em um endereço anterior ao da atual.

Caso no final da execução da macro houver sido detectado que a página está cheia, ocorre o processo ilustrado na figura 4.2. Primeiro, a *thread* tenta travar um *mutex* de troca de página. A *thread* que conseguir travar o *mutex* primeiro, testa se a próxima página não possui dados ainda não escritos no arquivo. Caso possua, espera que uma variável de condição seja sinalizada pela *thread* de gravação e então re-executa o teste até que a página tenha sido gravada. Quando a próxima página estiver liberada, é efetuada a troca da página atual pela seguinte e é sinalizada uma condição para alertar a *thread* de gravação, caso essa esteja esperando por páginas a serem gravadas.

A *thread* de gravação (ver figura 4.3) por sua vez, inicialmente testa se há páginas a serem gravadas, caso não existam espera por um sinal em uma variável de condição. Quando recebe o sinal, re-executa o teste. Quando há páginas a gravar, ela espera até que o contador de gravação da página tenha o valor zero e, então grava essa página no arquivo de rastros. Feito isso ela envia um sinal para desbloquear alguma *thread* que possa estar esperando que a página atual seja liberada para reutilização. Por último, é trocado o índice da página atual para o da seguinte e o processo é reiniciado.

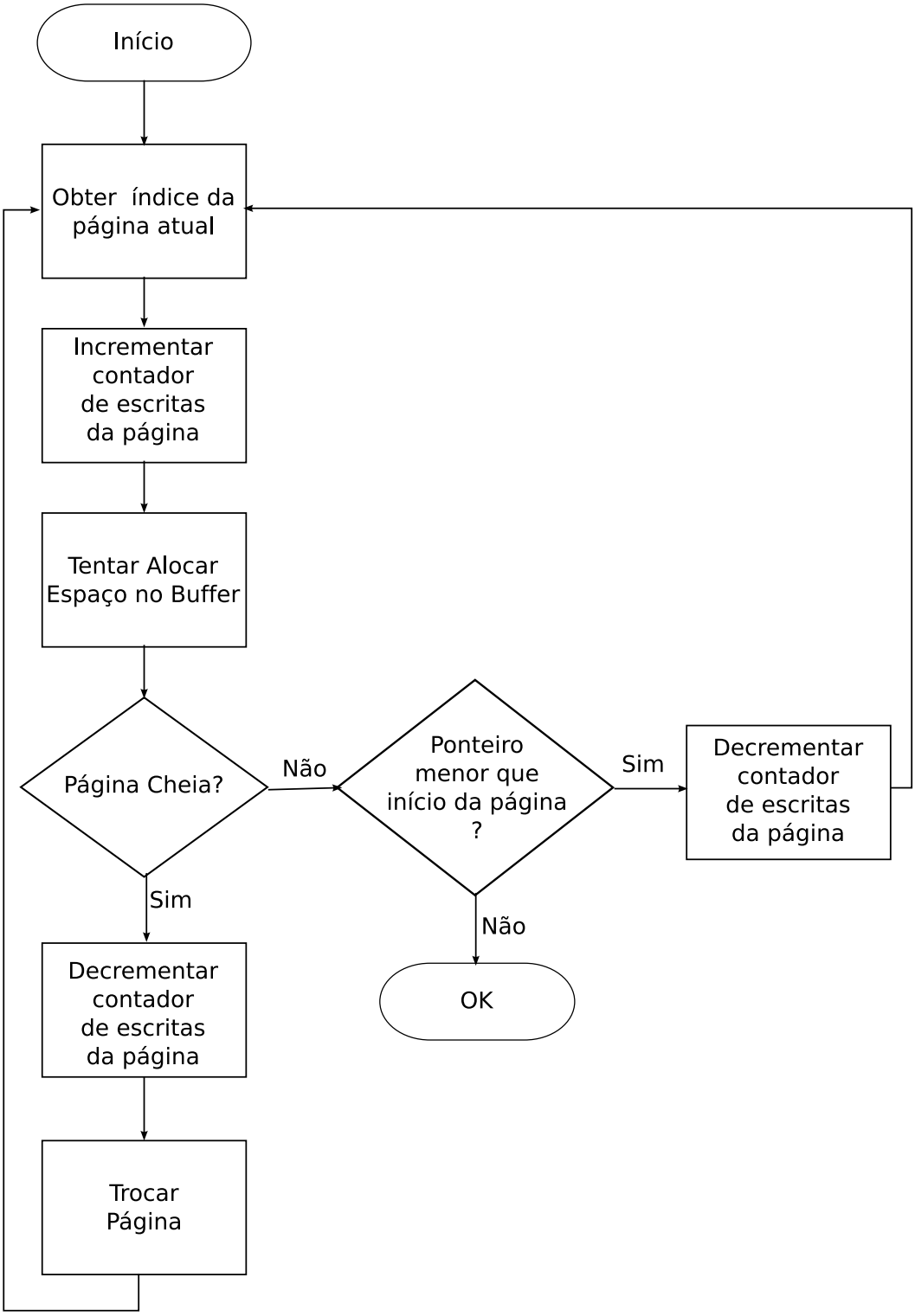

Figura 4.1: Fluxograma simplificado do esquema de alocação de *buffers*

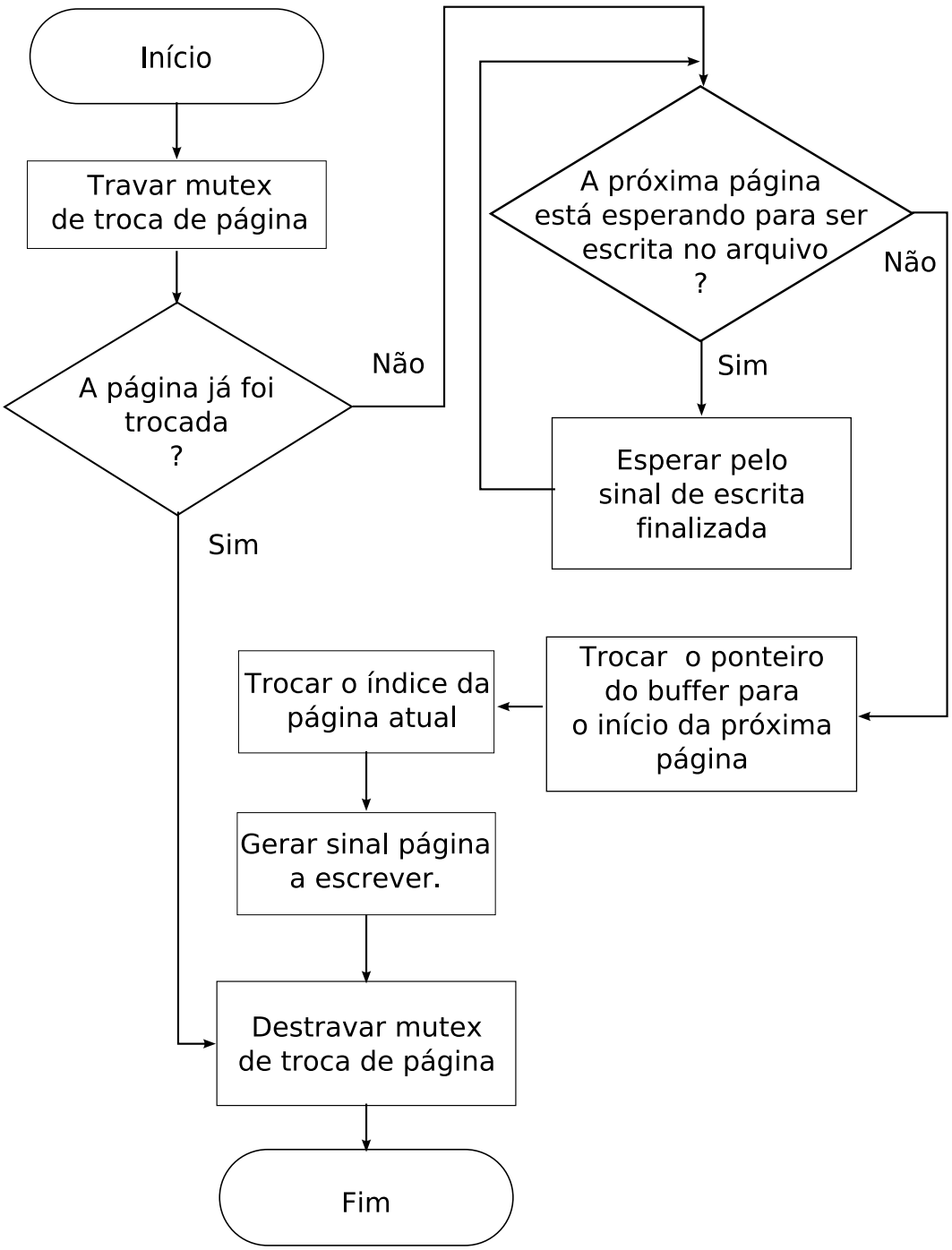

Figura 4.2: Funcionamento da troca de páginas do *buffer*

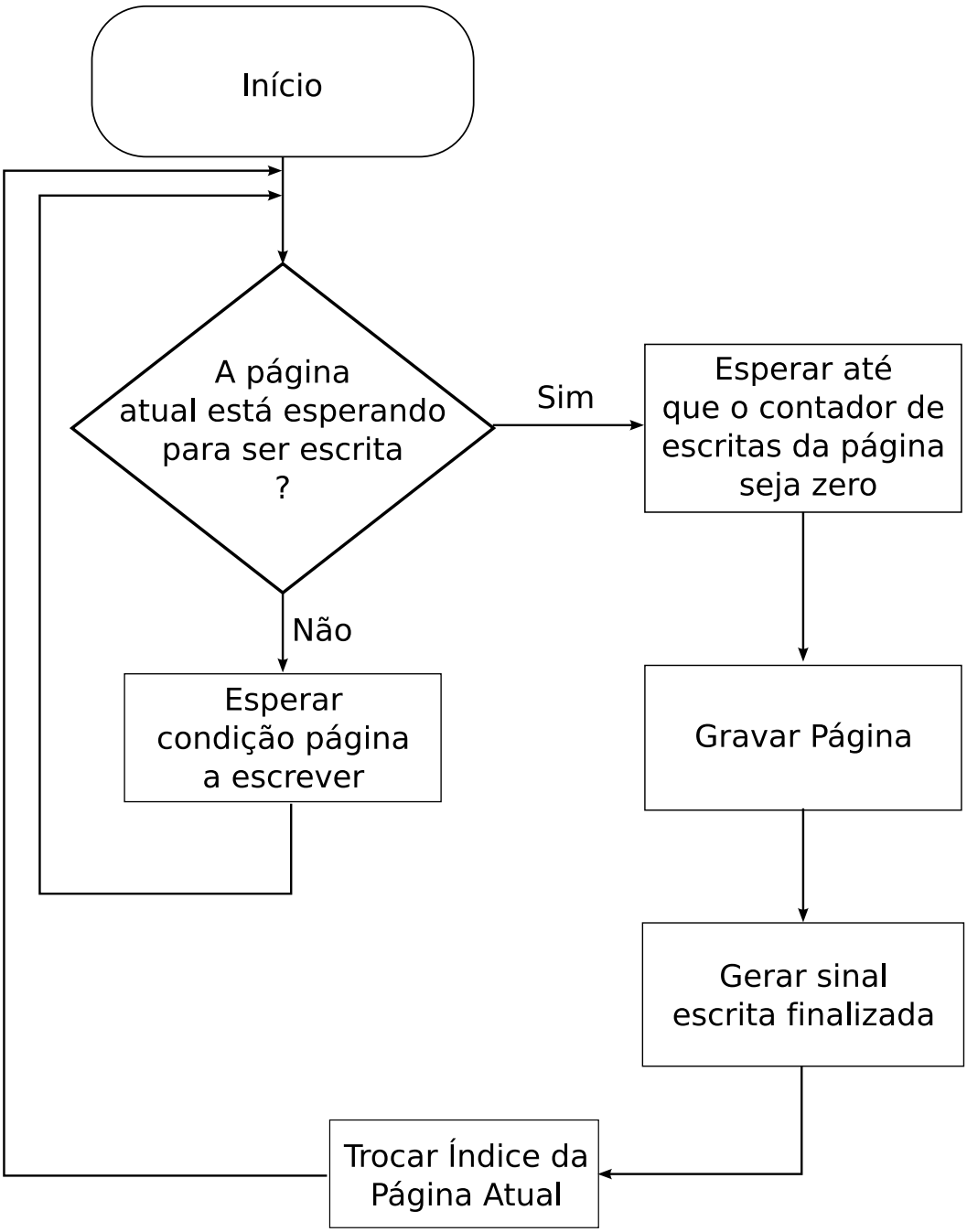

Figura 4.3: Funcionamento da *thread* de gravação em arquivo

5 AVALIAÇÃO E RESULTADOS

Para avaliar a implementação feita foram testados os tempos de execução de dois programas que executam a mesma tarefa. Um desses programas utiliza várias *threads* e outro é seqüencial. Todas as 'n' *threads* executam as mesmas operações, sendo que na versão seqüencial essas mesmas operações são executadas por 'n' iterações.

Para a realização dos testes foi utilizado um PC equipado com 2 processadores Intel Pentium III, com freqüência de operação de 1005,080 megahertz. Além disso, o sistema possui 1 *gigabyte* de memória RAM. A versão do sistema operacional Linux utilizada foi a 2.6.6.

Executou-se o programa seqüencial e o *multithread*, não instrumentados e instrumentados com dois tipos de eventos. O tipo de evento "1", registra apenas o tipo do evento. Já tipo "2" registra, além do tipo, um inteiro, um número de ponto flutuante de dupla precisão (*double*), e uma seqüência de caracteres de comprimento quinze.

Nos primeiros testes foram realizadas 10 execuções de cada versão do programa medindo-se seus tempos de execução e depois calculou-se a média aritmética desses tempos. O número de iterações ou *threads* foi 20. No caso dos programas instrumentados cada uma dessas registrou 500 mil de eventos. No total, então, foram registrados 10 milhões de eventos por execução.

A tabela 5.1 mostra as médias dos tempos de execução dos dois programas sem instrumentação. Nessa tabela e nas seguintes o símbolo Δ_{max} representa a diferença entre o maior e o menor tempos medidos. O programa *multithread* executou em média pouco mais da metade do tempo do seqüencial. Isso se deve à máquina utilizada nas medições possuir dois processadores.

As médias dos tempos de execução dos programas seqüenciais instrumentados com eventos dos tipos "1" e "2", citados anteriormente são mostradas na tabela 5.2. A cada

Programa	Tempo	Δ_{max}
Seqüencial	24,340 s	0,266 s
Multithread	14,211 s	0,0599 s

Tabela 5.1: Médias dos tempos de execução dos programas seqüencial e *multithread* de teste , sem instrumentação, em segundos

execução, foram gerados 10 milhões de eventos. São mostrados os tempos obtidos pela utilização da versão anterior da libRastro e da nova. Nessa tabela observa-se que os programas instrumentados com a nova versão foram mais rápidos com os dois tipos de evento. Deve-se levar em conta também que a nova versão gera um volume de dados maior, conforme mostra a tabela 5.3. Esse aumento no volume de dados se deve à gravação da datação com 64 bits e da identificação das *threads*, conforme explicado anteriormente no capítulo 4. Na versão anterior da ferramenta, o tamanho dos dados para o programa *multithread* é ligeiramente maior devido a ser gerado um arquivo para cada *thread* em que repete-se o início que é o evento de inicialização da libRastro. Na nova versão todos os processos geram apenas um arquivo.

Versão da libRastro	Tipo de Evento	"1"	"2"
Anterior	Média	32,311 s	37,712 s
	Δ_{max}	0,211 s	0,312 s
Nova	Média	28,166 s	33,177 s
	Δ_{max}	0,017 s	0,505 s

Tabela 5.2: Médias dos tempos de execução do programa seqüencial de teste, instrumentado, em segundos

Versão da libRastro	Eventos do tipo "1"	Eventos Tipo "2"
Anterior	Seqüencial: 80000044 B	Seqüencial: 360000044 B
	Multithread: 80000880 B (20 x 4000044 B)	*Multithread*: 360000880 B (20 x 18000044 B)
Nova	160000048 B	440000048 B

Tabela 5.3: Tamanho dos arquivos de rastro gerados na avaliação, em *Bytes*

Os mesmos dados mostrados para o programa seqüencial, aparecem na tabela 5.4 para a versão *multithread* do programa de teste. Nesse programa foram criadas 20 (vinte) *threads*, o que, na versão anterior da libRastro gerou 20 arquivos de rastro. Nesse caso, a nova versão foi mais lenta que a anterior. Isso provavelmente deve-se ao custo para trocar as páginas do *buffer*, pois essa rotina se utiliza bastante de mecanismos de sincronização. Essa hipótese é reforçada pelo fato de o programa de teste gerar um grande volume

de dados em um curto período de tempo. Isso faz com que as páginas do *buffer* fiquem freqüentemente cheias. Além disso, o fato de a nova versão gravar mais dados por rastro, para datação e identificação de *threads* pode também ser uma causa para seu pior desempenho nesse teste. Isso pode ser observado pela relação entre os tamanhos dos arquivos de rastro gerados. Na execução com rastros do tipo "1", por exemplo, a nova versão da libRastro gerou um arquivo com quase o dobro da quantidade de dados gerada pela outra versão.

Versão da libRastro	Tipo de Evento	"1"	"2"
Anterior	Média	17,062 s	19,898 s
	Δ_{max}	0,076 s	0,118 s
Nova	Média	20,105 s	25,490 s
	Δ_{max}	0,609 s	0,387 s

Tabela 5.4: Médias dos tempos de execução do programa *multithread* de teste, instrumentado, em segundos

Realizou-se também um teste com o intuito de ter uma melhor visualização do desempenho da ferramenta quando se registra eventos diferentes. Nesse teste, onde era registrado apenas um evento, foram colocados dois eventos, um de cada tipo dos utilizados nos testes anteriores. Cada *thread* ou iteração registrou 1 milhão de registros de cada um dos tipos de evento utilizados, totalizando 20 milhões de eventos. Com isso pretendia-se ter uma visão mais geral do tempo de execução. Os resultados estão na tabela 5.5. Pode-se observar que os tempos para o programa seqüencial foram menores com a nova versão da libRastro enquanto para o *multithread* foram maiores. Esse resultado é, portanto, compatível com os testes realizados anteriormente.

Versão da libRastro	Programa	Seqüencial	*Multithread*
Anterior	Média	45,421 s	23,957 s
	Δ_{max}	0,384 s	0,280 s
Nova	Média	37,340 s	30,367 s
	Δ_{max}	0,912 s	0,500 s

Tabela 5.5: Médias dos tempos de execução do programa instrumentado com os dois tipos de eventos de teste, em segundos

Foram realizados também testes para tentar estimar o tempo médio necessário para registrar um evento. Para isso foram executados programas de teste similares aos utilizados anteriormente mas que realizam como única operação o registro de 2 milhões de eventos. Foram realizadas 500 execuções de cada programa. Então foi calculada a média

dos tempos de execução em microssegundos e após, esse valor foi dividido por 2 milhões. Os resultados são apresentados na tabela 5.6.

Programa de teste		Seqüencial		*Multithread*	
Tipo de evento		Tipo "1"	Tipo "2"	Tipo "1"	Tipo "2"
Versão da libRastro	Anterior	0,775376 μs	1,286997 μs	0,407804 μs	0,693904 μs
	Nova	0,381463 μs	0,828881 μs	0,531137 μs	0,929344 μs

Tabela 5.6: Tempo médio para o registro de um evento, em microssegundos

Pode-se notar que os tempos médios para registrar um evento medidos com a nova versão da libRastro foram menores que os da original para os programas seqüenciais e maiores para os *multithread*. Isso é semelhante aos resultados observados pelos testes anteriores. Conclui-se portanto, a partir desses dados, que o custo médio para gravar um evento é maior na nova versão quando o programa instrumentado possui mais de uma *thread*.

Esse resultado também pode ser o reflexo da necessidade de bloquear as *threads* quando o *buffer* não possui páginas livres. Na versão seqüencial o custo é menor para a nova versão devido a, na libRastro original, haver a necessidade de o programa ficar bloqueado enquanto estão sendo escritos os dados no arquivo. Na nova versão, a *thread* de gravação é quem faz essa gravação e o programa não precisa ficar bloqueado a não ser que todas as páginas estejam esperando para serem escritas no arquivo.

Devido à nova versão ter se mostrado pior nos testes com programas *multithread*, resolveu-se realizar mais um teste. Nesse teste removeu-se da libRastro o comando responsável pela escrita do arquivo de rastros em disco, em ambas as versões da biblioteca. O objetivo desse teste foi analisar se o motivo do pior desempenho da nova versão estaria relacionado ao tempo que a ferramenta utiliza para gravar os dados. Os resultados para os programas seqüencias é mostrado na tabela 5.7 e para os *multithread* na tabela 5.8.

Versão da libRastro	Tipo de Evento	"1"	"2"	Misto
Anterior	Média	31,476 s	34,158 s	41,370 s
	Δ_{max}	0,183 s	0,001 s	0,402 s
Nova	Média	28,163 s	32,221 s	35,902 s
	Δ_{max}	0,020 s	0,365 s	0,405 s

Tabela 5.7: Médias dos tempos de execução do programa seqüencial de teste sem escrita em arquivo

Nota-se que para os programas seqüenciais a nova versão continuou sendo melhor.

Versão da libRastro	Tipo de Evento	"1"	"2"	Misto
Anterior	Média	16,646 s	18,006 s	21,539 s
	Δ_{max}	0,083 s	0,122 s	0,115 s
Nova	Média	19,144 s	21,214 s	23,476 s
	Δ_{max}	0,318 s	0,290 s	0,419 s

Tabela 5.8: Médias dos tempos de execução do programa *multithread* de teste sem escrita em arquivo

Assim também, os resultados para os *multithread* tiveram pouca variação. Conclui-se então que o menor desempenho para a nova versão com programas *multithread* deve ser causado pela sobrecarga gerada pelo mecanismo de troca de páginas do *buffer*. Portanto, se faz necessário o estudo de uma melhor implementação dessa funcionalidade.

6 CONCLUSÃO E TRABALHOS FUTUROS

A libRastro utilizava um método de datação pouco eficiente e preciso. Além disso, a mesma utilizava um *buffer* de memória por *thread* para armazenar os dados.

Foi implementado a datação utilizando instruções de baixo nível para ler o contador de ciclos do processador. Dessa maneira aumentou-se a precisão e diminuiu-se a influência dessa operação na execução do programa instrumentado.

Além disso, para simplificar a gerência de *buffers* foi implementado um *buffer* compartilhado entre as *threads*. Para garantir a exclusão mútua no acesso a esse *buffer* foi utilizada uma instrução atômica em linguagem de montagem para alocar espaço no mesmo. A utilização dessa instrução é menos custosa que o uso de mecanismos de exclusão mútua de alto nível. Também foram implementadas as modificações necessárias para possibilitar o uso desse *buffer* na escrita e leitura dos rastros e na geração dinâmica de funções de eventos.

Uma das principais modificações implementadas diz respeito à gravação dos dados do *buffer* no arquivo de rastros. Para isso foi criada uma *thread* que é responsável por essa gravação. Também foram utilizados mecanismos de sincronização necessários para evitar problemas relacionados ao acesso concorrente ao *buffer* durante essa operação.

Testes realizados mostraram que a nova versão apresenta problemas de desempenho quando usada para instrumentar programas *multithread*. Esses problemas provavelmente ocorrem devido aos mecanismos de sincronização utilizados na troca de páginas do *buffer* de rastros.

Como trabalho futuro seria interessante implementar uma técnica de gravação dos dados em arquivo que seja mais eficiente. Isso pode ser feito através de um melhor mecanismo de troca de páginas ou então utilizando alguma técnica que não necessite do mesmo. Também pode-se implementar um método eficiente de sincronização, para a ob-

tenção de tempo global na datação dos rastros. Isso facilitaria a utilização da libRastro para a análise de execução de programas distribuídos. Além disso, poderia-se estudar um método de obter dados sobre escalonamento de processos do núcleo do sistema operacional. Essa capacidade possibilitaria uma análise mais detalhada da execução.

REFERÊNCIAS

BOLDYSHEV, K.; RIDEAU, F.-R. **Linux Assembly HOWTO**. `http://www.tldp.org/HOWTO/Assembly-HOWTO/index.html`.

DANJEAN, V.; NAMYST, R.; WACRENIER, P.-A. **An Efficient Multi-level Trace Toolkit for Multi-threaded Applications**. Runtime / Futurs, France: INRIA, 2005. Submited to EuroPar2005. (RR-5513).

Intel Corporation. **IA-32 Intel Architecture Software Developers Manual**: basic architecture. [S.l.: s.n.], 2005. v.1.

Intel Corporation. **IA-32 Intel Architecture Software Developers Manual**: instruction set reference, a-m. [S.l.: s.n.], 2005. v.2A.

Intel Corporation. **IA-32 Intel Architecture Software Developers Manual**: instruction set reference, n-z. [S.l.: s.n.], 2005. v.2B.

MAILLET, E.; TRON, C. On efficiently implementing global time for performance evaluation on multiprocessor systems. **J. Parallel Distrib. Comput.**, Orlando, FL, USA, v.28, n.1, p.84–93, 1995.

RUSSEL, R. D.; CHAVAN, M. Fast Kernel Tracing: a performance evaluation tool for linux. **Proceedings of the 19th IASTED International Symposia on Applied Informatics**, [S.l.], 2001.

SAMUEL THIBAULT, R. D. R. **Developing a Software Tool For Precise Kernel Measurements**. [S.l.]: Unité de recherche INRIA Futurs, 2005. `ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-5602.pdf`.

SILVA, G. J. da; STEIN, B. Uma Biblioteca Genérica de Geração de Rastros de Execução para Visualização de Programas. **Anais do I Simpósio de Informática da Região Centro, Santa Maria**, [S.l.], 2002.

STALLMAN, R. M.; GCC DEVELOPER COMMUNITY the. **Using the GNU Compiler Collection (GCC)**. [S.l.: s.n.], 2005. `http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc.pdf`.

TANENBAUM, A. S. **Modern Operating Systems**. second.ed. [S.l.]: Prentice Hall, 2001.