JOÃO CLAUDIO MUSSI DE ALBUQUERQUE

ANÁLISE DO COMPORTAMENTO DA HIERARQUIA DE MEMÓRIA COM OPROFILE ESTENDIDO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Roberto André Hexsel

CURITIBA

2009

JOÃO CLAUDIO MUSSI DE ALBUQUERQUE

ANÁLISE DO COMPORTAMENTO DA HIERARQUIA DE MEMÓRIA COM OPROFILE ESTENDIDO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Roberto André Hexsel

CURITIBA

2009

DEDICATÓRIA

À minha avó, Palmira. Aos meus pais, João Liro e Maria Ana. À minha sobrinha e afilhada, Valentina. Aos meus irmãos Humberto Luis e Paulo Henrique. À minha irmã Ana Maria. Aos demais familiares que *também* não lerão esta dissertação. A qualquer pessoa que justifique a existência deste trabalho, lendo-o.

AGRADECIMENTOS

Aos meus familiares, pela paciência e apoio incondicionais. Aos grandes amigos, amigas e demais pessoas especiais, pela torcida e compreensão durante a minha ausência: Aristeu, Daniel, Cassio, Bruno, Ike, Rapha, Betam, Enrico, Tati, Nissin, Carlos, Anna, Maurício, Maria, Felipe, Kika, Jepi, Zé e, especialmente à Mell, a auxiliar de encadernação.

Ao meu orientador Roberto, principalmente pela paciência e pela fé no meu trabalho. À Jucélia, secretária do DINF, pela prestatividade. Aos amigos e colegas do SIMEPAR pela compreensão e apoio, Sato em especial. Aos doutores Marcelo Oliveira, Erasto Cichon e Julio Batista, pela minha segunda chance. A você, pelo seu interesse e atenção.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O *Oprofile* é um programa de monitoramento de desempenho cujo funcionamento é baseado na amostragem de dados dos *contadores de desempenho em hardware* (CDHs). Esta dissertação descreve uma extensão ao *Oprofile*, o *Oprofile Estendido*, que adiciona uma referência de tempo absoluta e o estado de todos os contadores de desempenho às amostras periodicamente coletadas. A referência de tempo, obtida do *Time Stamp Counter*, permite observar o comportamento temporal dos eventos monitorados na resolução definida pelo usuário, desde que suportada pelo projeto da CPU.

Com um conjunto de amostras, cada uma contendo o valor de todos os contadores de desempenho e a referência de tempo, é possível estabelecer relações entre as interações de eventos que ocorrem em frequências distintas.

Três experimentos mostram a utilização do *Oprofile Estendido*: (i) um programa de teste que provoca um número controlado de faltas nos dois níveis da TLB do sistema; (ii) programas de multiplicação de matrizes com diferentes níveis de otimização; e (iii) a comparação de desempenho, com relação à hierarquia de memória, da ordenação com duas versões do algoritmo de ordenação Mergesort: uma implementação simplória, e outra que divide o vetor em faixas do tamanho das caches e emprega o Quick sort nestas faixas.

Os resultados dos experimentos mostram que: (i) o *Oprofile Estendido* fornece dados importantes sobre o desempenho do sistema estudado; (ii) o *Oprofile Estendido* ajuda na compreensão do modo como os programas monitorados utilizam a hierarquia de memória; (iii) o *Oprofile Estendido* apresenta uma interferência próxima à do *Oprofile* em alguns experimentos, cuja intensidade se altera de acordo com a taxa de amostragem utilizada; e (iv) o *Oprofile Estendido* proporciona uma análise mais completa das sequências dos eventos ao longo da execução dos testes da que é possível de se obter com o *Oprofile*.

ABSTRACT

Oprofile is a performance monitoring tool based on *hardware performance counters* (HPC) data sampling. This dissertation describes an extension to *Oprofile*, *Oprofile Estendido*, which adds an absolute time reference and the state of all performance counters to the periodically collected samples. The time reference allows observing the temporal behavior of the monitored events at a user defined resolution, provided that resolution is supported by the processor. The time reference is acquired from the *Time Stamp Counter* (TSC).

With a set of samples, each containing the values of all performance counters and the time reference, it is possible to establish relations between the interactions of the different events that occur at distinct frequencies.

Three experiments show *Oprofile Estendido*'s usage: (i) a simple test program that causes a controlled number of faults on both levels of the system's TLB; (ii) matrix multiplication programs with different optimization levels; and (iii) a performance comparison, from the memory hierarchy point of view, between two Mergesort algorithms: a simple implementation and a complex one, which divides the vector in cache-sized tiles and applies Quick sort on these tiles.

The experiments show that: (i) *Oprofile Estendido* provides invaluable data on the performance of the system under study; (ii) *Oprofile Estendido* helps to improve the understandig on the way that monitored programs use the memory hierarchy; (iii) *Oprofile Estendido*'s interference on the system under study is similar to that caused by *Oprofile*, and the level of interference is related to the sampling rates; and (iv) *Oprofile Estendido* allows a more thorough analysis of the sequences of events than is possible with *Oprofile*.

CAPÍTULO 1

INTRODUÇÃO

Contadores de Desempenho em *Hardware* (CDHs) são dispositivos de monitoramento disponíveis em CPUs de alto desempenho. Estes dispositivos possibilitam monitorar e contabilizar eventos ocorridos no processador durante a execução de programas e são amplamente utilizados na otimização de aplicações.

As ferramentas de depuração de desempenho disponíveis até o advento dos CDHs eram capazes de fornecer a ordem das chamadas das funções e o tempo aproximado de execução de cada uma, bem como o número de vezes em que elas eram invocadas [15]. Com base nesses dados, observava-se o comportamento do programa nos trechos críticos de código e deduzia-se o que era possível otimizar no programa para melhorar o seu desempenho.

Informações mais detalhadas relacionadas à execução de um programa também podem ser adquiridas através de simuladores [3, 30], que são difíceis de ajustar adequadamente [34], são demorados para executar, além de gerarem um grande volume de dados e de não serem capazes de reproduzir tudo o que ocorre num processsador real durante a execução do programa.

A implementação dos CDHs viabilizou o monitoramento de eventos ocorridos no processador durante a execução de programas, o que desencadeou o desenvolvimento de novas ferramentas de depuração de desempenho, que são capazes de fornecer informações complementares às já existentes, com a finalidade de facilitar a identificação de gargalos presentes nos sistemas.

Dentre as ferramentas disponíveis para a utilização dos CDHs encontram-se: (i) drivers de acesso aos contadores [29]; (ii) bibliotecas para leitura e escrita dos CDHs [6, 16]; (iii) aplicativos de monitoramento [5, 2, 10, 7] – que funcionam sem nenhuma alteração no programa estudado; e (iv) ferramentas para a visualização dos dados coletados [11].

Cada processador que suporta CDHs provê um certo conjunto de eventos observáveis,

que incluem acessos à cache, desvios tomados, instruções completadas e faltas na cache da tabela de páginas (TLB) entre outros. A quantidade de eventos monitoráveis simultaneamente, bem como as formas de acesso aos contadores são também distintas para cada CPU.

Apesar dessas restrições, os instrumentos disponíveis possibilitaram a realização de pesquisas que resultaram em técnicas para otimização do desempenho de aplicações e compiladores baseadas no uso dos CDHs [8, 9], estudos relacionados à acurácia e confiabilidade dos contadores [33, 26, 18, 35] e pesquisas referentes aos métodos de coleta de dados dos CDHs e dos programas monitorados [27, 4, 34, 28].

O *Oprofile* é um instrumento de depuração de desempenho baseado na utilização de CDHs capaz de observar (ou monitorar) simultaneamente tantos eventos quanto permitidos pelo processador [24] sem causar perturbações significativas na execução do programa monitorado. As amostras coletadas pelo *Oprofile* não contém informação referente ao instante em que um evento ocorreu, nem informações a respeito do estado dos demais contadores naquele mesmo instante.

Após a execução de um programa monitorado pelo *Oprofile* obtém-se informações estatísticas, similares a um histograma, a respeito dos eventos ocorridos durante a medição, tais como: (i) o número de eventos causados pelo programa observado; (ii) o número de eventos causados pelos demais binários em execução; (iii) estatísticas das medidas; (iv) estatísticas de eventos sobre os símbolos de programas compilados com a opção de depuração '`-g`' do `gcc` (*GNU Compiler Collection*).

A primeira parte do trabalho aqui apresentado consistiu em expandir as funcionalidades do *Oprofile*, de forma que cada amostra coletada passasse a conter o ciclo de relógio *(clock)* em que ocorreu a coleta do dado e o estado dos demais contadores naquele mesmo instante. Esta informação complementar – do ciclo em que ocorre a coleta dos dados – permitiu observar a evolução da contagem dos eventos ao longo do tempo.

Outras extensões consistiram em disponibilizar, em arquivos diferentes dos utilizados pelo *Oprofile*, os registros de todas as amostras coletadas durante uma sessão de monitoramento com o *Oprofile Estendido*, nomes dos binários que causaram os eventos e seus

respectivos PIDs *(Process ID)* e GIDs *(Group ID)*.

As novas informações coletadas precisaram ser contextualizadas para viabilizar a sua interpretação correta. Para isso foi utilizada uma função independente do *Oprofile* que lê o ciclo de relógio do processador (`readtsc()`). Esta função foi inserida no código dos programas monitorados e foi utilizada para delimitar o momento em que se inicia um trecho da execução do programa cujo comportamento se queria observar.

A segunda parte do trabalho consistiu em utilizar essa versão modificada do *Oprofile* e a função `readtsc()` para: (i) estudar a distribuição temporal dos eventos monitorados durante uma medição; (ii) investigar o comportamento de programas com base na observação das faltas na L1, L2 e TLB; e (iii) demonstrar as perturbações que tanto o Sistema Operacional (SO) quanto o *Oprofile* podem causar em uma aplicação durante a sua execução.

Os programas que foram investigados com o auxílio da ferramenta, apresentados em ordem crescente de complexidade, são: (i) programas que acessam dados em memória para estimar o tamanho das TLB's; (ii) multiplicação de matrizes com diferentes níveis de otimização; e (iii) programas de ordenação.

Os estudos descritos em (i) e (ii) destinam-se a refinar a aplicação da metodologia e eventualmente ajustar as ferramentas. Com (iii) pretende-se estudar o comportamento da hierarquia de memória L1 e L2 *cache* durante a execução de programas baseados em recursão trabalhando com conjuntos de dados de tamanhos distintos.

Parte dos resultados dos estudos contidos neste trabalho foi utilizada no artigo "OProfile Estendido para Depuração de Desempenho" [1], apresentado no VI Workshop de Sistemas Operacionais (WSO'2009).

O restante do trabalho está organizado como descrito a seguir: o Capítulo 2 apresenta uma discussão sobre os trabalhos relacionados a esta dissertação e uma descrição mais detalhada do *Oprofile*. As alterações feitas no *Oprofile* e os detalhes a respeito das informações complementares coletadas estão descritas no Capítulo 3. O Capítulo 4 descreve a metodologia para a utilização do *Oprofile Estendido* enquanto o Capítulo 5 apresenta o ambiente de testes utilizado. Os Capítulos 6, 7 e 8 descrevem e discutem os experimentos

e resultados obtidos, enquanto o Capítulo 9 discute a interferência causadas pelo *Oprofile Estendido* nas medidas. O Capitulo 10 apresenta uma discussão a respeito da divergência encontrada entre os valores obtidos com o *Oprofile* e com o *Oprofile Estendido* em cada experimento. Finalmente, o Capítulo 11 apresenta as conclusões e estudos futuros enquanto os Apêndices A, B, C e D contém a listagem dos eventos suportados pelo Athlon, os códigos de programas e funções utilizadas no trabalho, e um exemplo de arquivo de configuração para o *Oprofile Estendido*.

CAPÍTULO 2

REVISÃO BIBLIOGRÁFICA

2.1 Trabalhos Relacionados

Existem duas modalidades de programação e leitura dos CDHs durante o monitoramento da execução de um programa, que são: contagem (*counting*) e amostragem (*sampling*). *Contagem* consiste na leitura dos CDHs antes e depois da execução do programa. Nesta abordagem, a quantidade total de eventos é obtida através da subtração dos valores registrados em cada uma das leituras.

Amostragem é baseada na leitura de um CDH a cada *overflow* do mesmo. Neste modo, quando o contador atinge o seu limite, uma interrupção de *hardware* permite a leitura do valor do contador e o disponibiliza para a aplicação de monitoramento, que por sua vez, contabiliza este evento nas estatísticas das medidas.

Em [27] é apresentado um estudo a respeito dessas duas modalidades de uso dos contadores e das implicações de cada uma na eficiência e acurácia das medidas. O estudo foi realizado com a ferramenta de medida PAPI, inicialmente descrita em [7]. O trabalho cita alguns processadores as modalidades de coleta por eles suportada. O trabalho conclui que ambos os modos são importantes e devem ser suportados em todas as plataformas de hardware possíveis. Conclui ainda, que é necessário mais trabalho para se determinar as características desejáveis em uma interface multi-plataforma de acesso a CDHs, além de sugerir estudos relacionados à calibração dos mesmos, a fim de melhorar a precisão das medidas.

A acurácia dos CDHs é estudada em [33]. Para estimar a precisão das medidas foram criados *microbenchmarks* capazes de causar uma quantidade previsível de eventos monitoráveis por CDHs, como faltas na TLB (*Translation Lookaside Buffer*). Os *microbenchmarks* foram executados em um processador MIPS R12000 monitorado pelos CDHs e em um simulador deste processador.

No processador, os dados dos CDHs foram lidos por intermédio das ferramentas de acesso disponíveis para a plataforma SGI MIPS, *perfex* e *libperfex*. A quantidade de eventos registrada nos CDHs foi comparada com os valores das simulações e com os valores estimados. Os resultados demonstram que, para cada tipo de *microbenchmark* e método de acesso aos CDHs, distintos níveis de acurácia eram atingidos. Além disso, as leituras realizadas com *perfex* e *libperfex* divergiram nos mesmos testes.

Dentre os eventos medidos estão: (i) instruções decodificadas; (ii) leituras em memória; (iii) escritas em memória; e (iv) desvios condicionais resolvidos. Destes, apenas (iii) não atingiu um índice de erro inferior a 10% depois dos *microbenchmarks* serem executados por tempo suficiente para que os valores lidos convirjam até os os valores estimados ou obtidos nas simulações. Não são analisadas as causas das diferenças entre as medidas obtidas com *perfex* e *libperfex*.

Uma continuação do estudo de acurácia descrito em [33] é apresentada em [26]. Neste trabalho foi utilizada uma metodologia semelhante à apresentada em [33], além da ferramenta *PAPI* [7] para realizar medidas em diferentes processadores. Algumas das diferenças observadas entre a previsão de eventos e a leitura dos CDHs nos *microbenchmarks* foram atribuídas à busca antecipada de dados em alguns dos processadores. Este problema foi contornado alterando-se nos *microbenchmarks* a sequência de acesso aos dados em memória, de linear para aleatória.

Os resultados obtidos permitiram tipificar as divergências entre as estimativas e os valores obtidos nas medições da seguinte forma: (i) *overhead* – representa uma diferença constante entre a medida prevista e a medida adquirida, possivelmente causada pela ferramenta de monitoramento, ou por uma perturbação, atribuída pelos autores a algum outro código não identificado; (ii) multiplicativa – o resultado obtido é o produto da medida esperada por um fator; (iii) aleatória – diferença que se apresenta apenas no início da medição e que tende a desaparecer à medida que o *microbenchmark* prossegue a sua execução; e (iv) desconhecida – não se enquadra nas demais categorias.

Diante desses resultados os autores propõem a inclusão de fatores de correção para os contadores nas ferramentas de análise dos dados para melhorar a precisão das medidas.

Tanto este estudo quanto o seu precedente não detalham o método de leitura dos registradores nem qual o *overhead* que este processo causa de acordo com a sua implementação nas diferentes plataformas analisadas, possivelmente pelo trabalho ter sido realizado em plataformas proprietárias.

Estes trabalhos também não consideram que algumas das perturbações observadas nas medidas podem ser causadas pelo próprio Sistema Operacional, uma vez que as leituras dos CDHs são diretamente comparadas com os resultados das simulações e cálculos de previsão, os quais não contemplam as interferências causadas pelo SO.

Em [14], os autores demonstram como a ferramenta de monitoramento para *clusters* RVision [13] teve sua funcionalidade extendida para suportar a leitura dos CDHs. Para isso foi criada uma biblioteca de acesso aos contadores que utiliza o driver de acesso aos CDHs disponível em [29]. Esta extensão do aplicativo foi utilizada para monitorar a execução de uma versão paralela do *benchmark* SPEC *Swim* [32].

Após a análise do resultado das medições de taxas de faltas na *cache* durante execução do *Swim*, um problema de conflito de mapeamentos em cache foi detectado e resolvido através de uma pequena alteração no código do programa. Após essa alteração, o tempo de execução melhorou em 25%.

Outras informações relevantes são apresentadas neste trabalho [14], como a análise da intrusão das medidas no desempenho dos programas monitorados, além da listagem de algumas métricas derivadas dos eventos monitoráveis, tais como operações completadas por ciclo, taxa de escritas e leituras por ciclo entre outras.

Duas técnicas para contornar algumas das limitações dos CDHs são apresentadas em [4]. A primeira consiste aumentar o número de contadores disponíveis para uma medição, multiplexando os eventos monitoráveis entre os contadores existentes no processador durante a execução da medida. Para isso, é criado um contexto contendo os dados dos contadores para cada processo em execução na máquina. Cada contexto acumula as informações obtidas dos contadores à medida que os eventos monitoráveis são multiplexados aleatoriamente ao longo da medição.

Os contextos de contadores são trocados a cada troca de contexto do SO de forma

que, para cada processo seja levantado o seu perfil de execução, sem que este perfil sofra a interferência dos demais processos existentes. Uma comparação entre os dados medidos através dessa técnica e os dados medidos sem a multiplexação dos eventos nos contadores indicou que a diferença entre as medidas obtidas era de aproximadamente 15%.

A segunda metodologia proposta associa especulativamente as paradas (*stalls*) do processador a um componente que seja o potencial causador da parada. Essa metodologia foi implementada a partir da ferramenta de multiplexação de CDHs e os resultados dos experimentos demonstraram que as medidas obtidas através desse método são razoavelmente precisas.

Além das duas metodologias descritas, o trabalho [4] também apresenta algumas relações entre a taxa de amostragem dos registradores e o *overhead* observado no sistema, assim como uma descrição detalhada das características de funcionamento dos processadores superescalares.

A ferramenta para análise de desempenho PAPI é descrita em [7]. Essa ferramenta disponibiliza uma série de métricas que podem ser utilizadas na depuração de desempenho de aplicações. Sua arquitetura é implementada em camadas, sendo que apenas a camada de aquisição dos dados dos contadores é dependente da arquitetura do processador e do SO, enquanto as demais são facilmente portáveis entre Sistemas Operacionais distintos.

A ferramenta PAPI contém conjuntos de contadores de eventos pré-definidos, além de suportar multiplexação de contadores, fazer medições estatísticas e medições em *threads*. PAPI suporta as plataformas Intel Pentium Pro/II/III no Linux, SGI/MIPS R10000/R12000 no IRIX 6.x, IBM Power 604/604e/630 no AIX 4.3, Compaq Alpha EV4/5/6 no Unix Tru64 e Cray T3e/Ev5 no Unicos/mk.

O trabalho [7] também cita as várias ferramentas de visualização dos dados obtidos através da PAPI como o *performeter* e *profometer*, além de mencionar o fato desses dados serem compatíveis com outras ferramentas de análise de desempenho como o *vprof* [20] e o *SvPablo* [11].

Uma abordagem alternativa à análise de desempenho baseada em eventos ocorridos nos CDHs é apresentada em [10]. Nesse trabalho é demonstrado como efetuar uma análise

do desempenho de um programa a partir do perfil de execução de pares de instruções escolhidos aleatoriamente. Para isso, é necessário *hardware* adicional não disponível em todos os processadores.

A função destes circuitos adicionais é registrar o que ocorre no processador durante a execução de uma determinada instrução. Os eventos observados incluem: (i) faltas nas caches de instruções e dados; (ii) faltas nas TLBs de instruções e dados; (iii) desvios previstos tomados; (iv) erro da previsão de desvios tomados; (v) instruções completadas; e (vi) latência, em ciclos, de cada estágio de execução da instrução no *pipeline*.

Com essas informações e mais algumas métricas a elas associadas, é possível encontrar gargalos nos sistemas examinados. Outras potenciais aplicações da ferramenta citadas pelos autores são: otimizações de compiladores, melhorias no escalonamento das instruções e aumento nas taxas de acertos das caches e TLBs.

Outra técnica alternativa à análise de desempenho baseada nos CDHs é a *instrumentação no kernel* estudada em [22, 23]. Neste trabalho é desenvolvida uma ferramenta de depuração de desempenho, denominada *PANALYSER*, que adquire as amostras a partir das chamadas de sistema `ptrace()`, `getrusage()` e `wait4()`. O *PANALYSER* é comparado às demais ferramentas de monitoramento do sistema – que adquirem as informações a partir do pseudo sistema de arquivos `/proc` – e concluiu-se que a instrumentação do *kernel* demanda menos processamento e, consequentemente, apresenta um *overhead* inferior ao das ferramentas baseadas na leitura do `/proc`. Apesar da instrumentação do kernel causar menos perturbação no sistema do que a leitura do `/proc`, ela ainda causa um *overhead* superior ao da leitura dos CDHs.

Uma análise dos eventos contados em processadores x86 é apresentada em [35]. Neste trabalho, vários processadores dessa mesma arquitetura são utilizados para medir instruções completadas e ciclos em que o processador não está parado. Uma vez que todos os processadores são da mesma arquitetura, o número de intruções necessárias para executar o mesmo programa não deve variar entre os processadores.

As medidas iniciais demonstraram, a priori, uma divergência de aproximadamente 1% entre os processadores testados. Foi então realizado um estudo que incluiu características

de *layout* de memória, análise de instruções dos *benchmarks*, análise do modo de execução – entre 32 e 64 *bits*, gerenciamento de memória e a própria implementação dos contadores.

Com os resultados e padronizações obtidas a partir dos dados deste estudo, os experimentos foram realizados novamente e a margem de erro obtida entre os processadores baixou para menos de 0,002%. Apesar da alta precisão da medida, ela apenas se refere à quantidade de instruções executadas. Os demais eventos disponíveis nos processadores não são testados.

Em [18] é feita uma comparação entre resultados de medidas obtidas com contadores de desempenho de diferentes processadores para os mesmos *benchmarks*. Paralelamente, características dos *benchmarks* tais como previsibilidade de desvios, conjunto de trabalho (*working set*), quantidade de memória utilizada foram analisados.

Comparando os resultados das medidas dos contadores com as características de cada programa examinado, foi constatado que *benchmarks* com características diferentes podem apresentar comportamentos similares em algumas arquiteturas, se analisados apenas pelo viés dos contadores de desempenho. Destas comparações, foi observado que as conclusões referentes ao perfil de execução de programas, quando obtidas a partir apenas dos dados dos CDHs em uma determinada arquitetura, não devem ser generalizadas para as demais.

Dados coletados de contadores de desempenho são utilizados para definir o melhor conjunto de *flags* de compilação para programas de *benchmark* em [9]. O processo consiste na utilização de redes neurais para fazer o mapeamento entre o comportamento dos eventos e as *flags* que devem ser ligadas ou desligadas para a compilação. Após a realização de um treinamento exaustivo na rede neural com vários tipos de programas, o mapeamento é obtido e utilizado como base para estabelecer o melhor conjunto de *flags* para um programa qualquer. Normalmente o conjunto de *flags* é definido em aproximadamente três compilações e execuções monitoradas.

As medidas de desempenho realizadas nos programas compilados com o conjunto de *flags* estabelecido pela ferramenta demonstraram que houve um ganho de desempenho superior a 10% em relação ao melhor conjunto de *flags* de otimização pré-definidos em um compilador comercial.

De acordo com [34], simuladores não-comerciais são difíceis de utilizar em pesquisas por serem mal documentados e implementados sobre bibliotecas velhas. Além disso, os modelos de processadores suportados são defasados e estão longe de simular a arquitetura dos processadores atuais.

Partindo deste cenário, três tipos de análises são realizadas e comparadas para um conjunto de *benchmarks*: execuções com simuladores, execuções com ferramentas de Instrumentação Dinâmica Binária (IDB) e execuções com contadores de desempenho. Os testes foram realizados no simulador mais parecido com o processador utilizado nos experimentos e os valores obtidos com os contadores de desempenho foram a referência de medida confiável.

A conclusão indica que, apesar de trabalhosas e imprecisas na maioria das ocasiões, as simulações ainda são necessárias. Além disso, foi concluído também que de acordo com o grau de detalhe necessário para a análise dos programas, as ferramentas de IDB podem oferecer vantagens por serem mais simples e mais rápidas, apesar de fornecerem menos detalhes sobre as execuções.

2.2 OProfile

O *Oprofile* é uma ferramenta de monitoramento de desempenho baseada no uso dos CDHs composta por: (i) componentes dependentes de arquitetura (DA) – código responsável pelo gerenciamento, configuração e manipulação dos CDHs em cada arquitetura suportada; (ii) *oprofilefs* – pseudo *filesystem* que armazena os arquivos responsáveis por informar e receber configurações do espaço de usuário, além de outras informações referentes aos eventos monitoráveis; (iii) *CPU Buffer* e *Event Buffer*– áreas de armazenamento que mantém as amostras em espaço de SO até que elas sejam transferidas para a área de usuário; (iv) *driver* genérico para o *kernel* – recebe as amostras enviadas pelo componente DA e armazena-as no *buffer* apropriado até transferí-las para o *daemon* no espaço de usuário; (v) *OProfile daemon* – responsável por receber as amostras do kernel e escrevê-las em disco, sendo as amostras separadas e salvas em diferentes arquivos; (vi) ferramentas de pós processamento – selecionam as amostras requeridas pelo usuário associando-as aos

binários relevantes, apresentando a informação num formato compreensível.

Durante a execução do *Oprofile*, cada CDH disponível no processador é inicializado e configurado para monitorar um determinado evento que, quando ocorre, causa um incremento do contador apropriado. No momento em que o contador atinge o valor limite, uma interrupção causa a execução do código do *driver* (DA), que lê o valor do PC e o número do contador que entrou em *overflow*, reiniciando-o e inserindo os valores do contador e do PC no *CPU Buffer*. O código do *driver* DA também registra no *CPU Buffer* as trocas de processos em execução, assim como as trocas de modo de execução entre usuário e sistema.

Periodicamente, o *CPU Buffer* é sincronizado com o *Event Buffer* pelo *driver* genérico do *OProfile*, que além de converter o valor do PC em um *offset* relativo ao binário causador do evento, insere os dados das amostras e do *offset* referente ao PC no *Event Buffer*, junto com um identificador do binário em execução no momento do registro da amostra. Os dados são então transferidos para o espaço de usuário pelo *OProfile daemon*, que trata os dados coletados criando e mantendo em disco arquivos de amostras para cada binário em execução.

Para cada binário é mantido um conjunto de arquivos, um arquivo para cada evento monitorado. O arquivo contém uma série de registros com o valor do PC no momento em que a interrupção do contador foi atendida, além do número de interrupções já atendidas para cada um dos valores de PC registrados. O número de registros, multiplicado pelo número de ocorrências, corresponde ao número de interrupções associadas ao contador e é portanto proporcional à contagem de eventos. No pós-processamento é possível associar o valor de cada PC a um símbolo do binário, caso este tenha sido compilado com a opção `-g` do `gcc`.

A figura 2.1 descreve o trajeto percorrido por uma amostra durante uma medição do *Oprofile*. Primeiramente, os contadores são configurados e inicializados pelas chamadas `setup()` e `start()`, contidas na porção dependente de arquitetura do driver do *Oprofile*. Após algum tempo, o contador 0 entra em *overflow*. Neste momento, uma interrupção de *hardware* executa o código que descobre qual contador entrou em *overflow*, qual o valor

do PC, qual a tarefa em execução e envia essas informações para o *driver* genérico através da função `oprofile_add_sample()`.

A amostra e demais informações são então armazenadas no *CPU Buffer*. No *Oprofile* há um *CPU Buffer* para cada processador no sistema. O *CPU Buffer* também armazena informações a respeito da troca de processos em execução e da troca de modo de operação entre modo usuário e modo sistema ao longo de uma sessão de monitoramento.

Periodicamente, os dados são transferidos para o *buffer* principal, denominado *Event Buffer*. O código contido no arquivo `buffer_sync.c` é responsável pela sincronização dos *buffers*, pela criação de identificadores únicos *(dcookie)* para cada binário executado no sistema e pelo armazenamento de informações complementares.

As amostras são então transferidas para o espaço de usuário através do *OProfile daemon*, responsável por essa transferência, pela criação de novos arquivos à medida que amostras de novos binários lhe são entregues na sincronização periódica dos arquivos. Para cada binário executado, um arquivo por evento monitorado é criado e mapeado em memória. Cada um desses arquivos de dados contém os totais de amostras para cada valor de PC que tenha causado um *overflow*.

O *daemon* organiza os ponteiros para os arquivos numa lista encadeada tanto em LRU *(Last Recently Used)* quanto pelo índice gerado a partir do *hash* do nome do binário monitorado. A sincronização dos arquivos de amostras mapeados com o disco ocorre periodicamente até o término da execução do *Oprofile*. Maiores informações a respeito desse processo e das ferramentas de pós-processamento disponíveis no *Oprofile* podem ser encontradas em [25].

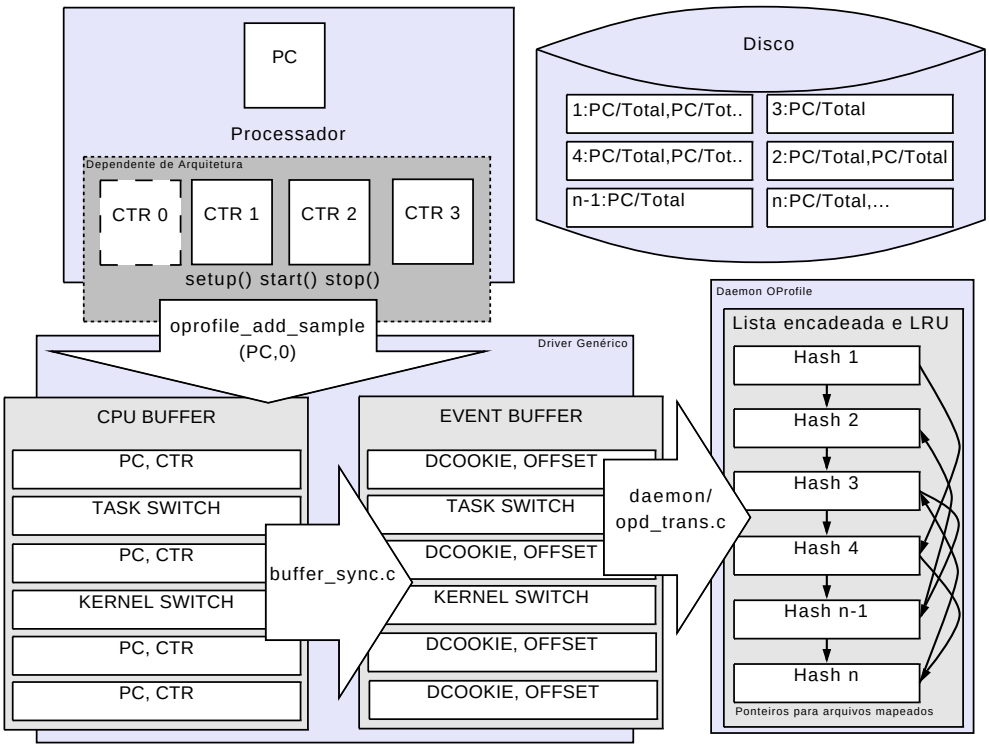

Figura 2.1: Esquema simplificado do OProfile.

CAPÍTULO 3

OPROFILE ESTENDIDO

O código do *Oprofile* foi alterado pelo autor, de forma que cada amostra coletada contenha uma referência de tempo, além do estado dos demais contadores no momento da interrupção. Este capítulo descreve as alterações realizadas no código do *Oprofile* que resultaram no *Oprofile Estendido*.

3.1 Alterações na coleta dos dados

As primeiras alterações realizadas foram inseridas no código do *driver* DA do *Oprofile*, e são executadas no momento em que ocorre a interrupção que permite a leitura dos contadores de desempenho e determina o contador em *overflow*. A referência de tempo foi obtida implementando neste trecho de código uma função que lê o *Time Stamp Counter* (TSC)[1] no momento em que este código é executado. A informação referente ao estado dos demais contadores foi obtida alterando o código que verifica os contadores, de forma que os valores de todos fossem copiados para um *array*.

O valor do TSC e o *array* contendo os valores de todos os contadores foram incluídos nos parâmetros da chamada da função `oprofile_add_sample()`, para que pudessem ser inseridos no *CPU Buffer* junto aos demais dados. Os códigos das estruturas *CPU Buffer* e *Event Buffer* foram alterados para incluir os valores do TSC e dos demais contadores a cada nova amostra. Os arquivos `buffer_sync.c` e `daemon/opd_trans.c` também foram alterados para incluir esses valores nas operações de sincronização e transferência entre os *buffers*.

No código do *daemon*, foram inseridos dois novos arquivos `opd_logfile.c` e `opd_logtable.c`. O código contido em `opd_logfile.c` é responsável por mapear um arquivo de registro contendo, para cada amostra, o valor do TSC, as informações sobre o estado da

[1] TSC é um contador de 64 bits que conta o número de ciclos de relógio do processador a partir do instante da inicialização do computador.

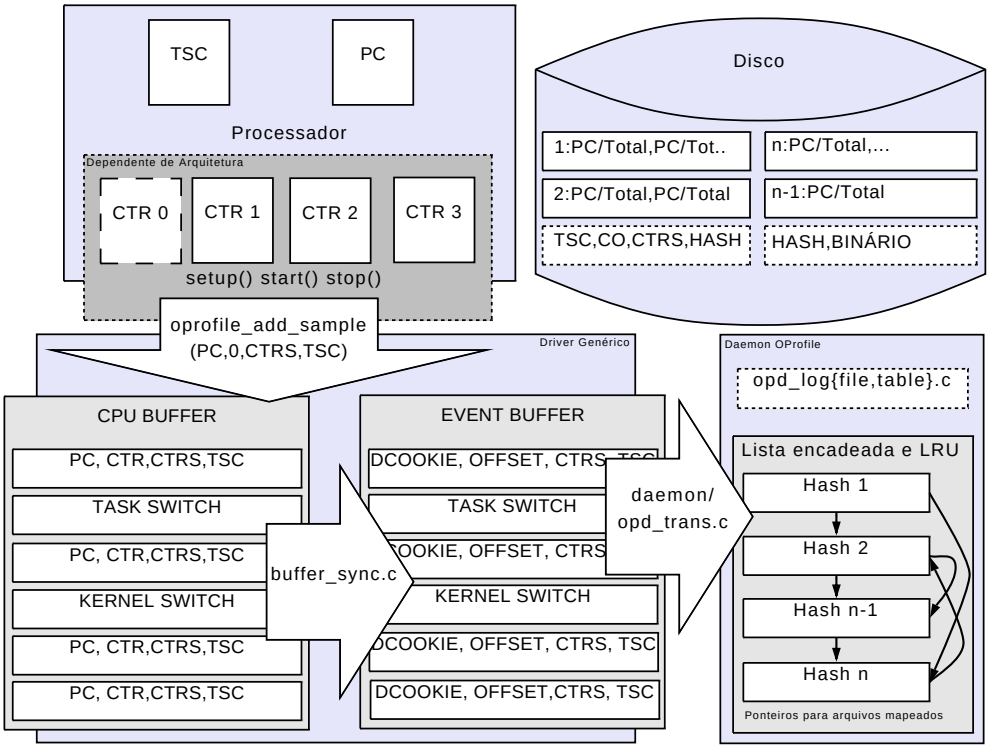

Figura 3.1: Alterações realizadas no OProfile.

contagem dos CDHs e o valor do *hash* do binário que estava sendo executado, bem como seus respectivos PIDs e GIDs. As amostras são inseridas no arquivo de log mapeado em memória à medida que são processadas e o arquivo em disco é atualizado a cada sincronização do SO.

O arquivo `opd_logtable.c` contém o código para manter um arquivo de índice contendo o *hash* e o nome do binário de cada programa em execução que tenha gerado dados de amostras durante a monitoração do *Oprofile*. O arquivo de índice é atualizado a cada novo nome de binário observado pelo *Oprofile*. A figura 3.1 contém o diagrama simplificado do *Oprofile* original com as alterações realizadas no trabalho.

3.2 Pós-processamento e análise das amostras

A análise dos resultados obtidos com o *Oprofile Estendido* é baseada no registro gerado a partir os dados coletados que, pós-processados, podem fornecer gráficos, contagens dos eventos em trechos da execução, etc. Todas as rotinas relacionadas ao pós processamento dos dados foram desenvolvidas à parte do código original do *Oprofile* e as principais

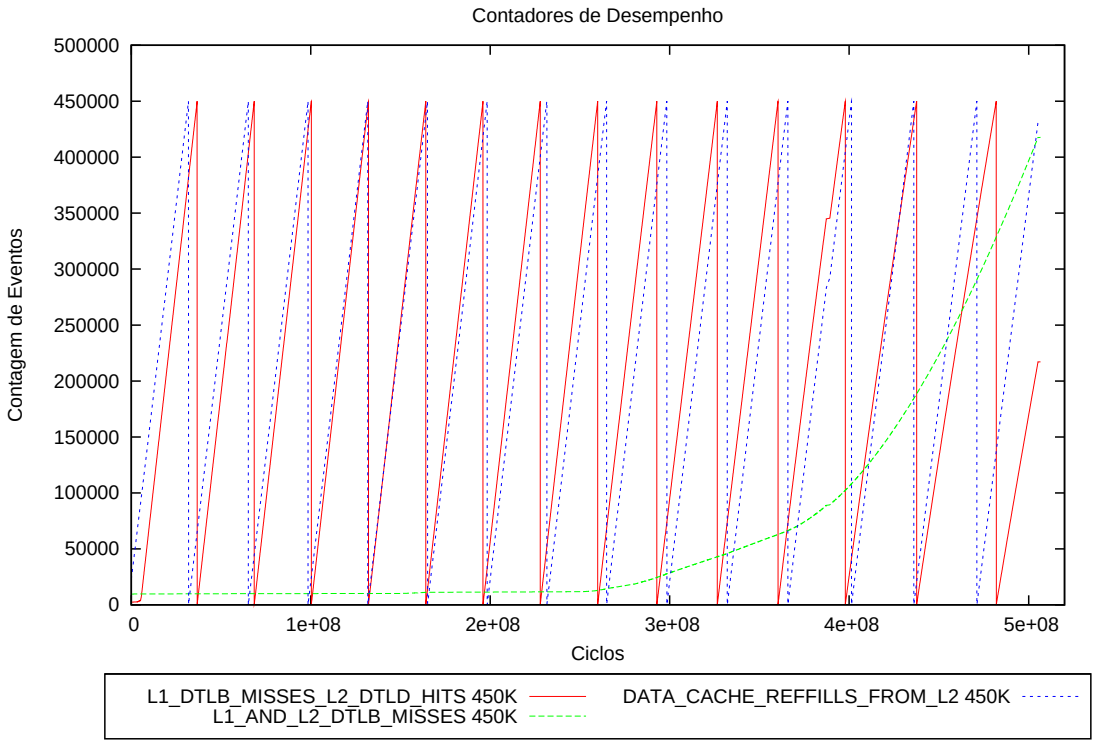

Figura 3.2: Ocorrência de diferentes eventos medidos pelo OProfile Estendido.

características das ferramentas de pós processamento dos dados estão descritas a seguir.

A coleta de amostras do *Oprofile Estendido* é controlada somente pelo contador que acumula os ciclos de relógio em que o processador está ativo (`cpu_clk_unhalted`). Dependendo do grau de resolução que se queira na medição, o número de eventos registrado por unidade de tempo pode ser muito elevado por conta da frequência do relógio dos processadores atuais, o que torna mais lenta a geração de gráficos de eventos $\times$ tempo. A visualização do eixo do gráfico que representa os ciclos de relógio é também prejudicada em função do tamanho dos valores inicial e final inseridos na escala.

Para melhorar a visualização desses dados nos gráficos a escala de ciclos foi deslocada para zero, subtraindo-se o valor do primeiro ciclo registrado pelo *Oprofile Estendido* dos demais ciclos. Dessa forma, além de melhorar a visualização do gráfico, é possível estabelecer um ponto inicial na execução de um programa, o que torna possível a comparação entre gráficos de medidas de diferentes execuções do mesmo programa.

O *Oprofile* é capaz de monitorar vários eventos. De acordo com a natureza do evento monitorado, é provável que ele ocorra numa frequência mais alta que a dos demais, fa-

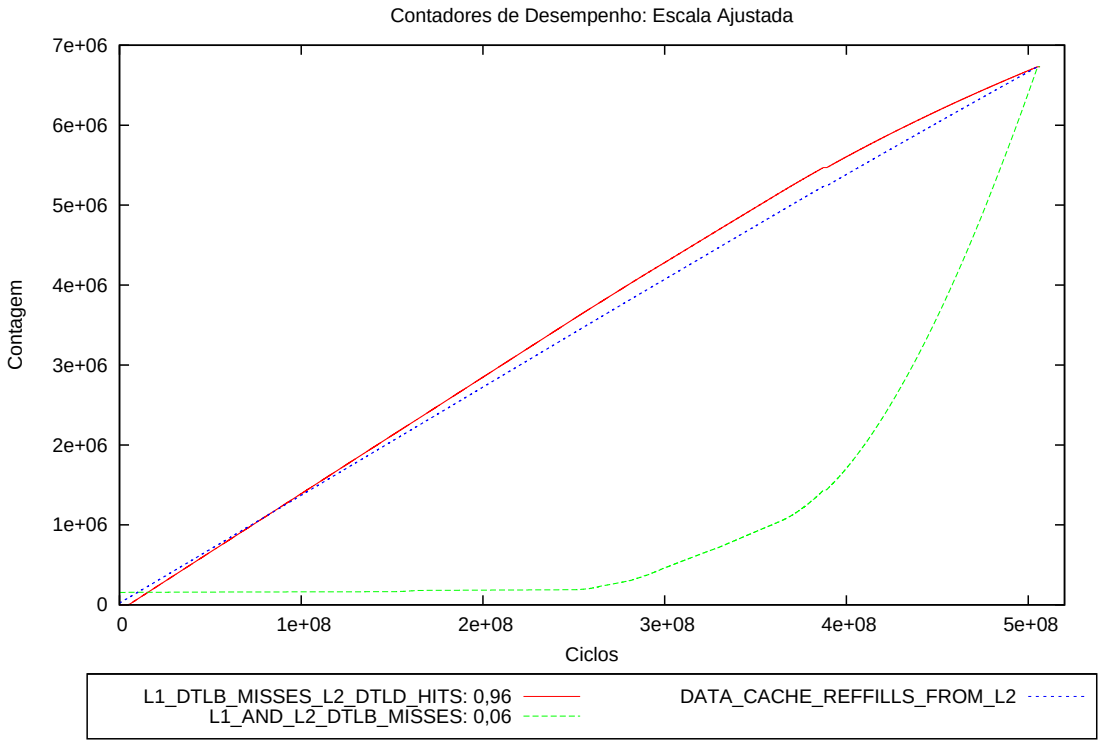

Figura 3.3: Eventos ajustados para comparação visual.

zendo com que um contador atinja seu limite e seja reinicializado mais vezes do que os outros. Isso causa um efeito de serra no gráfico da contagem dos eventos, como mostra a Figura 3.2. Observa-se na figura que, enquanto `Data_Cache_Refills_from_L2` ocorre aproximadamente na mesma frequência que `L1_DTLB_Misses_L2_DTLD_Hits` – ambos atingindo o pico do gráfico aproximadamente quatorze vezes – `L1_and_L2_DTLB_Misses` atinge o pico do gráfico apenas uma vez, denotando uma frequência de ocorrência aproximadamente quatorze vezes inferior.

Para corrigir esse problema, foi implementada uma rotina que soma, a cada valor lido, o valor total da contagem até a última re-inicialização do contador. Deste modo, a linha em forma de serra é plotada como uma linha que cresce de forma monotônica. Para evitar que a inclinação da linha do evento que ocorre em maior quantidade torne invisíveis as variações na linha do evento que ocorre em menor quantidade, ao final da rotina de soma, todos os pontos da linha são ajustados em relação à escala do evento que ocorreu o maior número de vezes. Desta forma, as linhas do gráfico são plotadas em escalas diferentes mas facilmente comparáveis, como mostrado na figura 3.3. O fator de ajuste de cada curva é

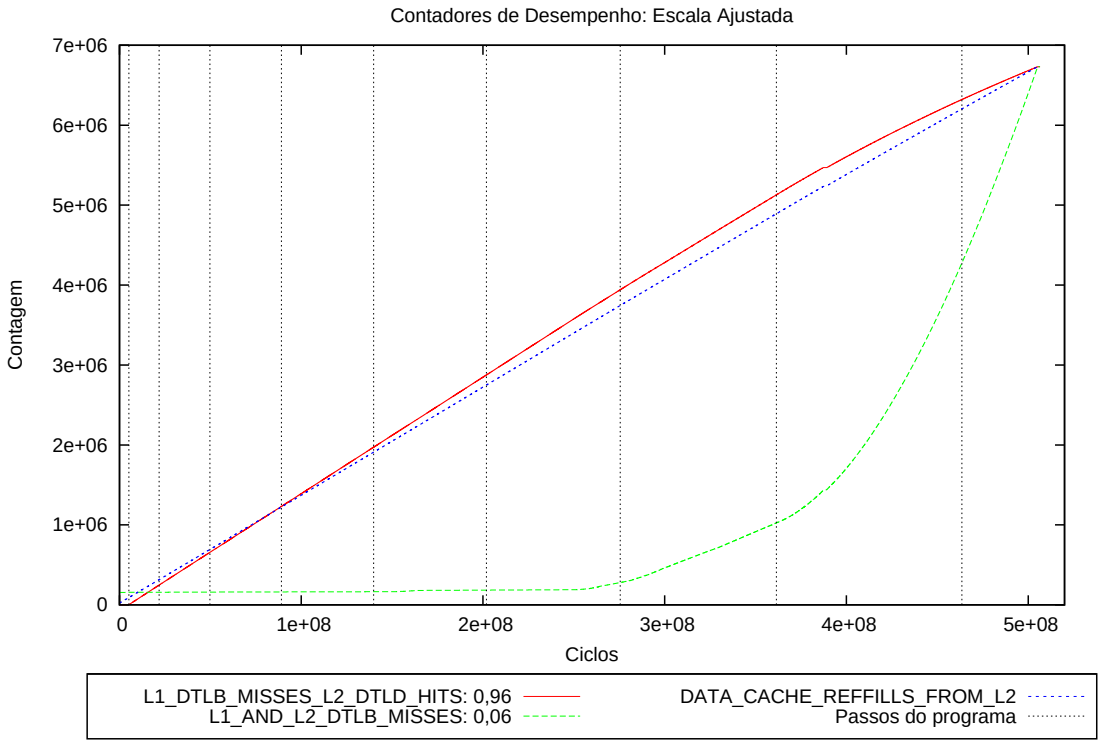

Figura 3.4: Leitura do ciclo do processador durante a execução do programa

mostrado na legenda da figura.

Para delimitar pontos interessantes da execução do programa, assim como o início e o término da execução do mesmo, foi escrita uma função – `readtsc()`, cujo código está listado no Apêndice C – que lê o ciclo do processador registrado no *Time Stamp Counter*. O resultado da inserção da função `readtsc()` no código do programa monitorado pode ser visualizado na figura 3.4. As linhas verticais denotam os instantes em que a função `readtsc()` é invocada.

Estas são as ferramentas utilizadas para facilitar a análise dos dados coletados pelo *Oprofile Estendido*. Na medida que mais testes foram concebidos e realizados, mais ferramentas se tornaram necessárias e foram desenvolvidas. Neste mesmo processo, novas potencialidades do *Oprofile Estendido* foram enxergadas e exploradas, o que será demonstrado no Capítulo 4, nas formas de visualização apresentadas.

CAPÍTULO 4

METODOLOGIA

Para analisar o comportamento de um determinado programa com o *Oprofile Estendido* a seguinte metodologia deve ser utilizada.

Primeiramente, é necessário fazer uma análise da natureza da aplicação monitorada e definir o tipo de evento que pode estar relacionado aos problemas de desempenho inerentes à aplicação[1]. Por exemplo, analisar as faltas nas *caches* e TLBs em um programa que executa operações matemáticas em grandes quantidades de dados pode ser mais interessante a *priori* do que medir as falhas nas previsões de desvios do mesmo programa.

Monitorar diferentes eventos durante uma mesma execução permite identificar as relações entre os eventos. É possível, por exemplo, saber a proporção das faltas na *cache* L1 que causam faltas na L2, ou mesmo nas demais estruturas presentes nos outros níveis da hierarquia de memória. É possível também relacionar um período de muitas faltas nas TLBs a uma queda na taxa de instruções completadas e assim por diante.

Definidos os eventos a serem monitorados, é necessário verificar a taxa de amostragem ideal para coletar os dados. Durante uma medição com o *Oprofile Estendido*, o primeiro contador é configurado para contar os ciclos de relógio do processador (`Cpu_Clk_Unhalted`). Cada vez que aquele entra em *overflow*, é coletada a amostra do programa em execução e o estado dos demais contadores. A frequência com que o primeiro contador entra em *overflow* é definida como taxa de amostragem e pode ser alterada pelo usuário.

A taxa de amostragem utilizada depende da resolução requerida pela aplicação em teste e do tempo de execução da mesma. Por exemplo, eventos monitorados na *cache* L1 podem requerer uma taxa de amostragem mais alta do que eventos monitorados na TLB, uma vez que os eventos da TLB são menos frequentes. Por outro lado, monitorar eventos na TLB com a mesma frequência que se monitora eventos na *cache* resulta em

[1] Os eventos suportados pelo Athlon estão listados no Apêndice A

um *overhead* de processamento desnecessário.

Da mesma forma, monitorar os eventos – ainda que na frequência recomendada – durante muito tempo gera uma quantidade de dados enorme, dos quais – de acordo com o caso – grande parte podem ser seguramente descartada sem que a visualização do gráfico seja prejudicada. Este caso também ilustra uma situação onde boa parte do *overhead* da medida poderia ser evitado.

Definido o valor ideal da taxa de amostragem, o código do programa a ser medido deve ser analisado e a função `readtsc()` inserida no código para delimitar os pontos mais interessantes a serem investigados. Também é interessante inserir uma chamada desta função no início e no final da execução do programa, de forma que o *overhead* da carga e da destruição do processo medido não interfira nos dados analisados.

Com o *Oprofile Estendido*, é possível visualizar os dados para a análise de três formas distintas: (i) a *contagem dos eventos*; (ii) a *taxa dos eventos*; e (iii) a *contagem discriminada dos eventos*. Cada um dos métodos de visualização pode exigir opções de configuração diferentes no momento da coleta de dados. Além disso, o pós-processamento dos dados das amostras também são distintos para cada um dos casos.

Na *visualização da contagem* dos eventos, o conjunto de dados é tratado com o ajuste das curvas e a demarcação dos pontos interessantes do programa com o auxílio da função `readtsc()`, como visto no Capítulo 3. A Figura 4.1 mostra a evolução da contagem dos eventos `ICache_Fetches` e `ICache_Misses` durante a carga de um programa. A linha vertical pontilhada, delimita o início da execução do programa e foi obtida com a execução da função `readtsc()`.

A *visualização da taxa* é obtida dividindo-se as variações (Δ) de um evento entre duas amostras consecutivas pela variação de outro evento nas mesmas duas amostras. Os eventos são escolhidos de acordo com a informação que se queira visualizar. Na Figura 4.2, o mesmo trecho de carga de programa é apresentado, mas as linhas plotadas representam as taxas dos eventos Δ`ICache_Accesses`/Δ`Ciclos` e Δ`ICache_Misses`/Δ`ICache_Accesses`.

A taxa dos eventos apresenta oscilações em alguns trechos da execução da medida. Isto

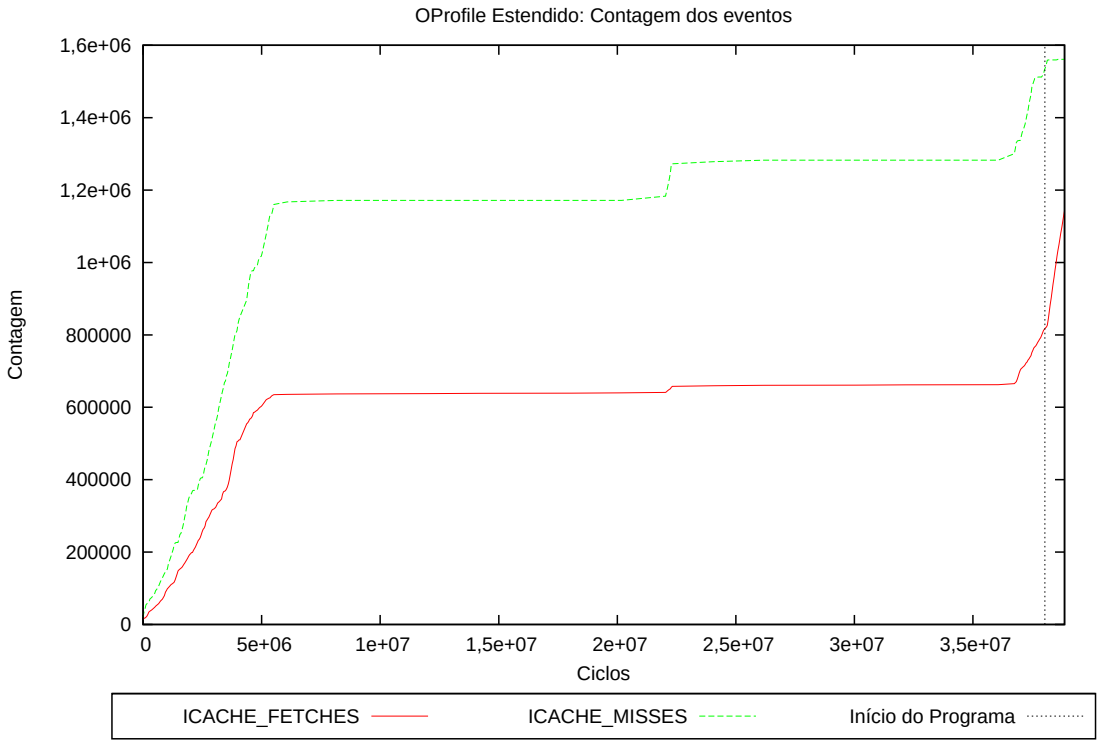

Figura 4.1: Contagem dos eventos do OProfile Estendido.

acontece porque a execução da interrupção que lê a amostra do *Oprofile Estendido* não ocorre instantaneamente, uma vez que ela depende do atendimento pelo processador, que pode demorar de centenas a milhares de ciclos de relógio de acordo com a carga da CPU. Devido a estas variações no tempo de atendimento da coleta da amostra, a contagem dos eventos monitorados pode sofrer alguma alteração entre momento em que ocorre o *overflow* e o momento da leitura do contador. Esta variação é praticamente invisível na visualização da contagem mas torna-se bastante evidente na taxa.

A *visualização discriminada* apresenta, além da contagem dos eventos, o binário que estava sendo executado no momento em que foi realizada a leitura dos contadores e outras informações. É possível gerar este tipo de visualização porque o *Oprofile* é capaz de distinguir as amostras dos binários monitorados entre: (i) *lib* – código de bibliotecas compartilhadas, (ii) *kernel* – código de *kernel* executado, (iii) *cpu* – em qual processador o código foi executado, (iv) *thread* – caso o mesmo programa seja executado em *threads*, separa as amostras por *thread* e (v) *all* – todas as opções.

De acordo com as opções de configuração escolhidas na coleta das amostras, cada uma

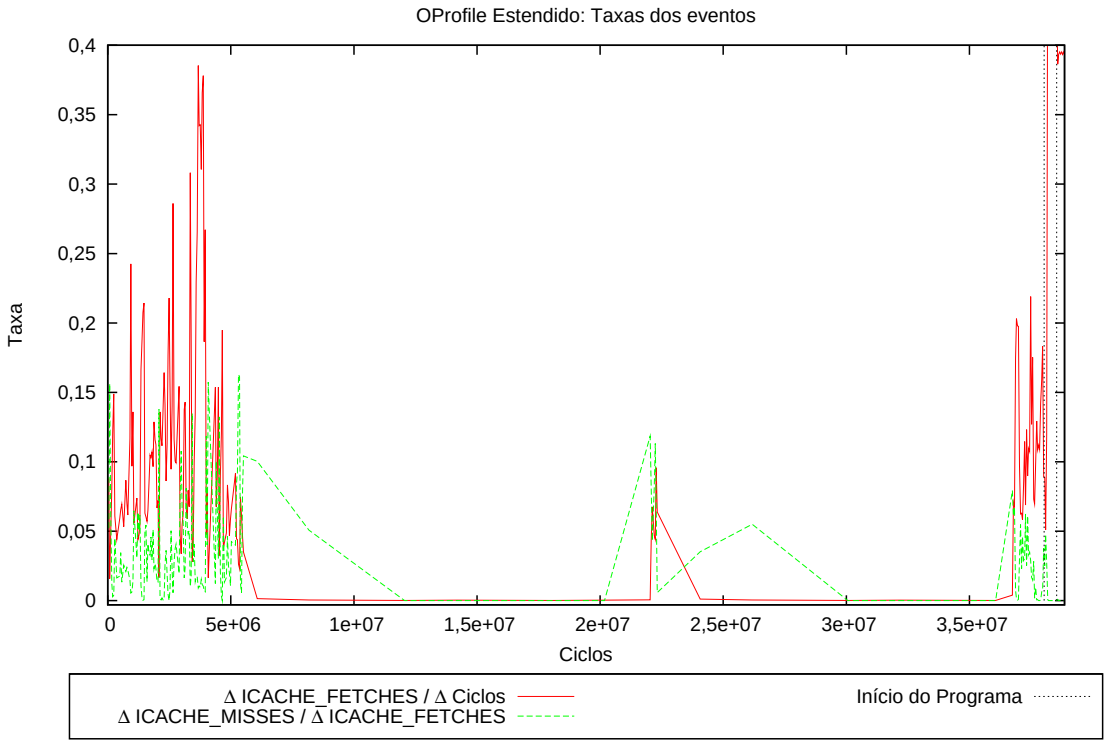

Figura 4.2: Taxas das contagens dos eventos do OProfile Estendido.

delas conterá informações complementares como nome do binário, nome da biblioteca, número do PID e número do GID. Considerando a quantidade de programas rodando em um sistema, as bibliotecas utilizadas na implementação de cada um, além do respectivo PID ou GID que diferentes instâncias do mesmo programa podem possuir, nota-se que é inviável plotar um gráfico contendo todas estas informações. Neste caso, as amostras coletadas tem de ser reagrupadas de forma que a visualização dos dados se torne possível, além de manter-se apenas as informações relevantes ao experimento em questão.

Um exemplo desta visualização é demonstrada na Figura 4.3 – que mais uma vez apresenta o mesmo trecho de carga do programa `dinamico` – no qual todas as amostras foram separadas e reagrupadas de acordo com a conveniência de visualização. Cada padrão de ponto representa a sequência da contagem de um evento. Cada cor atribuída aos pontos equivale a um binário detectado pelo *Oprofile Estendido* durante a monitoração. Os dados coletados que não fazem parte do conjunto que se quer observar (*oprofile, ld, libc, libm* e *dinamico*) foram agrupados em um grupo e plotados como "resto". Os trechos em que há uma grande distância entre os pontos representam *stalls* (paradas) do processador.

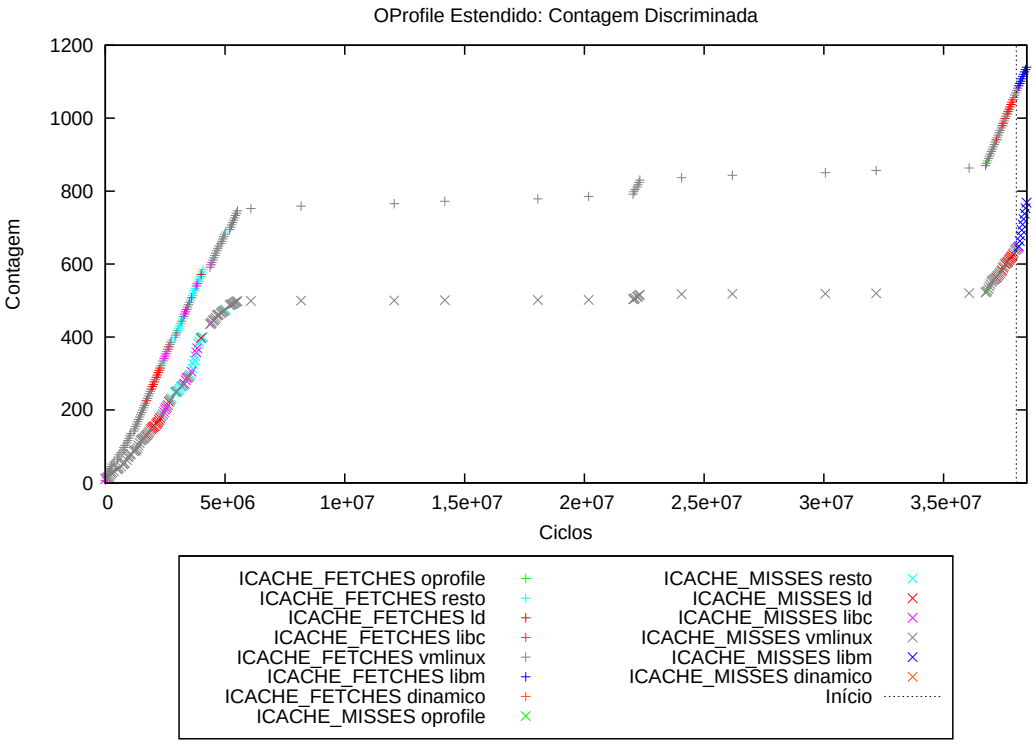

Figura 4.3: Contagem discriminada dos eventos do OProfile Estendido.

CAPÍTULO 5

AMBIENTE DE TESTES

Todos os testes apresentados neste trabalho foram realizados na mesma máquina. Os experimentos realizados com o *Oprofile Estendido* foram executados no GNU Linux-2.6.8 com as alterações nos *drivers* do *Oprofile* original. A versão do *Oprofile* usada como base para o desenvolvimento do *Oprofile Estendido* foi a versão 0.9.2.

O processador utilizado nos experimentos é o AMD Athlon^{TM} XP 2400+, família 6, modelo 8. A maioria das informações aqui descritas foram obtidas no seu manual [19], assim como a Figura 5.1, que contém um diagrama do processador com os elementos da sua hierarquia de memória destacados em vermelho. As informações não encontradas na documentação do Athlon foram complementadas com os dados fornecidos pelo programa `x86info` [21].

A hierarquia de memória do Athlon tem dois níveis de memória *cache*. A *cache* primária é separada entre *cache* de dados e *cache* de instruções. A *cache* de dados tem capacidade de 64 K*bytes* com 512 conjuntos, cada conjunto contendo dois blocos (associatividade binária), cada um com 64 *bytes* e a reposição ocorre sobre o elemento *menos recentemente utilizado* (*Least Recently Used*). A escrita de dados funciona com escrita preguiçosa (*write back*) e alocação em falta na escrita.

A *cache* de instruções tem capacidade para 64 K*bytes*, associatividade de duas vias e armazena 64 *bytes* por bloco. A reposição dos blocos é *Least Recently Used* (LRU) e cada busca de instrução recupera da L2, ou da memória principal, os 64 *bytes* que preenchem uma linha inteira, além dos 64 *bytes* da linha seguinte, tirando proveito da localidade espacial.

O segundo nível da *cache* é unificado para dados e instruções, com associatividade de 16 vias, 64 *bytes* por bloco e capacidade de 256 K*bytes*.

A hierarquia de TLBs é organizada em dois níveis, sendo o primeiro nível separado para

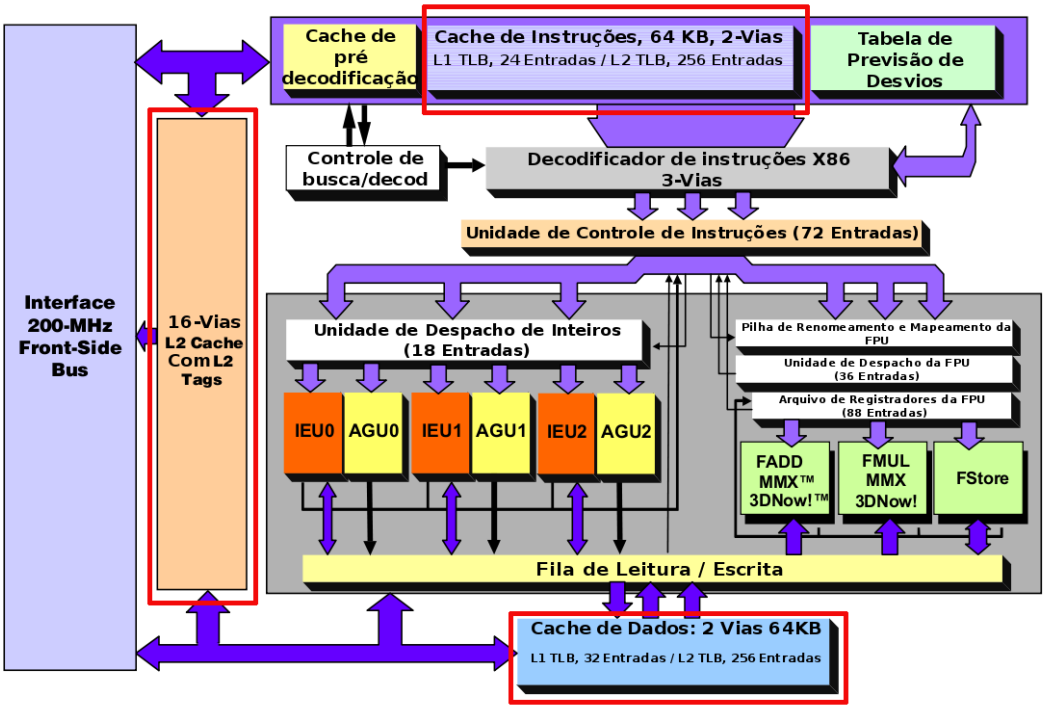

Figura 5.1: Diagrama do processador.

instruções e dados. A TLB de dados do nível 1 contém 32 mapeamentos para páginas de 4 K*bytes* e 8 mapeamentos para páginas de 2 ou 4 M*bytes*, além de ser totalmente associativa e com reposição LRU. Na TLB de instruções do primeiro nível há 16 blocos para páginas de 4 K*bytes* e 8 blocos para páginas de 2 ou 4 M*bytes*, a reposição é LRU e a associatividade é total.

O segundo nível da TLB deste processador também é separado para dados e instruções. Tanto a TLB de instruções quanto a de dados do nível 2 mantém 256 mapeamentos com associatividade de 4 vias. As duas TLBs possuem apenas mapeamentos para páginas de 4 K*bytes* [19].

CAPÍTULO 6

ANÁLISE DAS FALTAS NAS TLBS

Este capítulo descreve um experimento que visa a monitorar as faltas na TLB durante a execução de um programa que causa um número definido de faltas nesta estrutura. O objetivo dos experimentos é observar o comportamento das faltas na L1 e L2 TLB à medida que o tamanho do *buffer* percorrido na execução do programa supera a capacidade do nível da TLB testado.

6.1 Descrição do programa

Para causar a quantidade de faltas esperadas durante a sua execução, a cada 'passo' do teste, o programa aloca um conjunto de n páginas e as percorre 500 vezes, efetuando uma leitura em cada página. Dessa forma espera-se que a cada passo ocorra ao menos uma falta a mais que no passo anterior. O algoritmo do programa é mostrado em linguagem 'C' na Figura 6.1.

Uma área de tamanho $n/5$ [1] é alocada para separar a área original de tamanho n daquela que será alocada com tamanho $n+1$. As áreas de tamanho n e $n/5$ são liberadas somente após a alocação da nova área, que é então percorrida 500 vezes. O separador de tamanho $n/5$ diminui a probabilidade de que dois testes consecutivos usem o mesmo conjunto de páginas, o que reduziria a ocorrência de faltas pelo reaproveitamento dos mapeamentos na TLB.

6.2 Preparação do Experimento

Esta seção descreve os valores e padrões utilizados nos programas para cada teste, além dos parâmetros de configuração utilizados no *Oprofile Estendido*.

[1] $n/5$ foi o valor escolhido. Entretanto, este poderia ser qualquer outro valor, tal como $n/3$, $n/7$, etc.

```
...

/* aloca o tamanho inicial do buffer */
buffer1 = malloc (n);
do {
    /* repete 500 iterações */
    for (i = 0; i < 500; i ++)
        /* acessa todo o buffer, página por página */
        for (size = 0; size < n; size + pagesize)
            tmp = buffer1[size];

    /* incrementa o buffer em uma página para a próxima sequência de 500
       acessos */
    n+=pagesize;
    /* separa duas alocações consecutivas */
    buffer2 = malloc(n/5);
    /* aloca um buffer novo */
    buffer3 = malloc(n);
    free(buffer2);
    free(buffer1);
    /* aponta buffer1 para o buffer recém alocado */
    buffer1 = buffer3;
} while (n < max);

...
```

Figura 6.1: Código 'C' do programa que causa faltas controladas nas TLBs.

6.2.1 Parâmetros dos Programas

Os tamanhos inicial e final do *buffer* são definidos pelo usuário. O programa também aceita o número de iterações que o *loop* deve executar e o incremento da quantidade de páginas a cada sequência de iterações do *loop*. Em cada experimento, o programa executou 500 iterações do *loop* acrescentando 1 página ao *buffer* a cada sequência de iterações.

Três medições foram realizadas com este programa. A primeira ao percorrer um *buffer* variando entre 0 e 349 páginas, na qual foram medidas as faltas nas TLBs de primeiro e segundo níveis. A segunda, percorrendo um *buffer* variando entre 22 e 41 páginas em que foram medidas as faltas na L1 TLB e a terceira, percorrendo um *buffer* variando de 250 a 279 páginas, na qual foram medidas as faltas na L2 TLB.

6.2.2 Configuração do OProfile Estendido

A configuração utilizada no *Oprofile Estendido* foi a mesma em todos os testes: (i) `Cpu_Clk_Unhalted`, 150.000 eventos; (ii) `Data_Cache_Accesses`, 500.000 eventos; (iii) `L1_Dtlb_Misses_L2_Dtld_Hits`, 500.000 eventos; e (iv) `L1_And_L2_Dtlb_Misses`, 500.000 eventos.

A taxa de amostragem é definida pelo evento `Cpu_Clk_Unhalted`. Os demais eventos estão configurados apenas para ajustarem os contadores aos eventos a se monitorar. Apesar de todos estes eventos serem monitorados em todos os testes, apenas os eventos relevantes a cada teste são apresentados nos resultados.

6.3 Resultados dos Experimentos

6.3.1 Buffer de 0 a 349 páginas

O gráfico da Figura 6.2 apresenta as faltas ocorridas durante a execução do programa percorrendo *buffers* com tamanho de 0 a 349 páginas. A primeira linha vertical sólida demarca a capacidade do *buffer* em 32 páginas, a segunda linha vertical sólida representa o ponto em que o *buffer* atinge 256 páginas e as duas faixas verticais pontilhadas demarcam o início e o final da execução do programa. A linha superior representa as faltas na L1d-TLB e a inferior as faltas na L2-TLB – com fator de escala de $0,186$.

O degrau nas duas linhas que medem as faltas nas TLBs próximo ao ciclo $2{\cdot}10^9$ decorre da execução de código do *kernel*, *libc* e *Oprofile Estendido*. Este degrau nas contagens dos eventos associado à execução de código do OProfile indica o momento em que o SO pára a execução do programa monitorado e passa a executar o código do OProfile para sincronizar os *buffers* do OProfile bem como salvar dados das amostras coletadas.

A Tabela 6.1 mostra o que é executado pelo processador no trecho do log de amostras do *Oprofile Estendido* extraído do intervalo em que ocorre o degrau. Na primeira coluna da Tabela 6.1 estão os valores referentes ao ciclo em que foi realizada a leitura dos contadores, na segunda coluna estão os nomes dos binários que estavam sendo executados no momento da leitura, na terceira, quarta e quinta coluna estão os valores ajustados dos contadores que

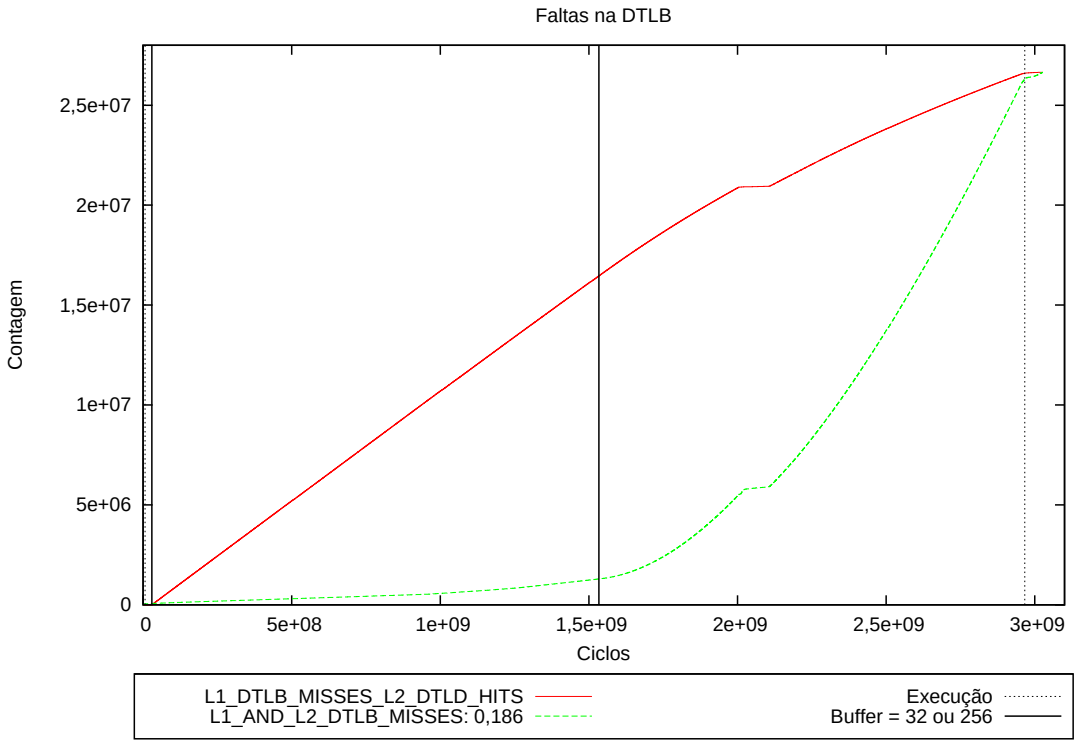

Figura 6.2: Faltas na L1 e L2 DTLB percorrendo *buffers* entre 0 e 349 páginas.

registram os eventos `Cpu_Clock_Unhalted`, `L1_Dtlb_Misses_L2_Dtld_Hits` e `L1_and_L2_DTLB_Misses` respectivamente. Os nomes dos binários na segunda coluna mostram que o programa `tlb_teste` parou de ser executado durante este intervalo da medida, enquanto o *Oprofile Estendido* realizou a sincronização dos seus registros.

Sobre o comportamento das faltas na hierarquia de memória apresentado na Figura 6.2, durante a metade esquerda do gráfico as faltas na L1 TLB aumentam a partir do ponto em que o *buffer* atinge 32 páginas – a capacidade da L1 TLB – na primeira linha vertical sólida. A partir deste ponto, as faltas nesse nível continuam ocorrendo na medida em que mais e mais páginas são alocadas.

Quando o tamanho do *buffer* se aproxima da capacidade da L2 TLB, o número de faltas naquela aumenta lentamente. Quando a capacidade da L2 é ultrapassada – centro da figura – o número de faltas na L2 TLB aumenta gradativamente a medida em que mais páginas são alocadas para o *buffer*, enquanto a contagem de faltas na L1 TLB parece diminuir.

A redução na contagem das faltas na L1 TLB se deve à própria definição do evento

Ciclo	Binário	Clock Unhalted	L1 M L2 H	L1 & L2 M
2004291143	tlb_teste	235494181	20996488	990284
2004442118	oprofile	235548720	20996736	990402
2004455914	oprofile	235555491	20996736	990404
2004592743	oprofile	235627403	20996736	990410
2004743619	oprofile	235710063	20996736	990416
2004894172	oprofile	235795961	20996736	990423
2005044853	vmlinux	235876816	20996737	990430
2005195523	vmlinux	235960173	20996738	990448
2005346466	vmlinux	236042444	20996739	990457
2005371143	oprofile	236055612	20996739	990458
		...		
2014107167	vmlinux	237882634	20997489	1016162
2014258021	libc-2.3.6.so	237931057	20997527	1016803
2014408780	vmlinux	237976297	20997547	1017637
2014559754	libc-2.3.6.so	238020696	20997582	1018400
2014645408	vmlinux	238056054	20997605	1018955
2014710594	oprofiled	238071539	20997610	1019272
2014861908	vmlinux	238133560	20997652	1020043
2015012886	vmlinux	238179766	20997664	1021159
2015163807	vmlinux	238229001	20997702	1021916
2015314592	vmlinux	238274002	20997721	1023022

Tabela 6.1: Trecho do *log* do OProfile no qual ocorre o degrau da Figura 6.2.

monitorado, uma vez que ele conta faltas na L1 com acertos na L2 TLB. A partir do momento em que as faltas na L1 não correspondem a acertos na L2, a contagem desses eventos na L1 tende a diminuir uma vez que aumentam as faltas na L2 TLB.

Outro fator que colabora para a redução das faltas na L1 TLB na segunda metade do experimento é o custo mais elevado das faltas na L2: o número de ciclos decorridos entre duas faltas na L2 TLB é superior ao de duas faltas na L1 TLB com acertos na L2 TLB. As faltas na L2 TLB causam uma redução na taxa de faltas por ciclos, mas não no total de faltas.

O resultado deste experimento coincide com o esperado: enquanto as páginas percorridas são todas mapeadas pela L2 TLB, a taxa de faltas na L1 TLB é aproximadamente constante. Quando o número de páginas percorridas excede a capacidade da L2 TLB, o número de faltas nesta aumenta gradativamente, assim como diminui a contagem de faltas na L1 com acertos na L2 TLB. Outro efeito do aumento das faltas na L2 TLB é o aumento do custo relativo de cada falta na L1 TLB.

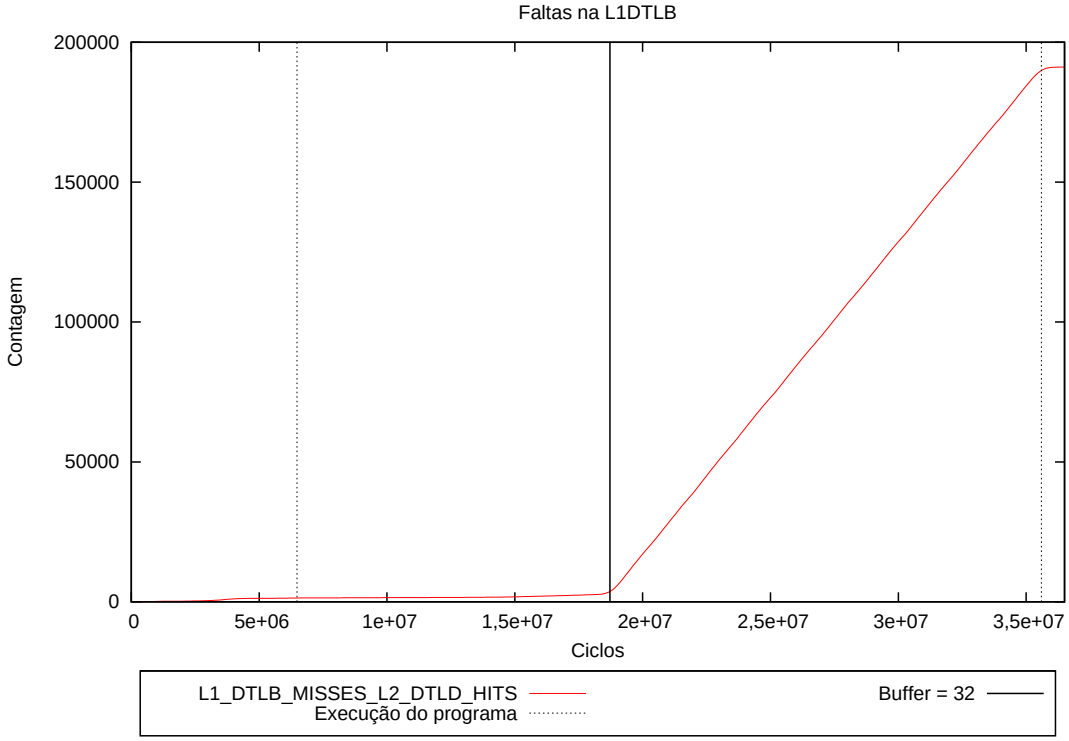

Figura 6.3: Faltas na L1 DTLB percorrendo *buffers* entre 22 e 41 páginas.

6.3.2 Buffer de 22 a 42 Páginas

A Figura 6.3 mostra as faltas ocorridas na L1 DTLB enquanto o programa percorre *buffers* com tamanhos entre 22 e 41 páginas. As linhas verticais pontilhadas representam o intervalo de tempo em que o programa foi executado e a linha vertical sólida marca o ponto em que o *buffer* atinge o tamanho de 32 páginas, que é a capacidade da L1 TLB. A linha plotada apresenta a contagem de faltas na L1 TLB ao longo da execução do programa.

Como visto no Capítulo 5, a hierarquia de TLBs do Athlon possui dois níveis. A TLB de primeiro nível, L1d-TLB contém 32 mapeamentos, é totalmente associativa com reposição *least recently used* (LRU) [19]. É esperado que, antes da utilização de 32 páginas, cada ciclo de 500 acessos acumule apenas o número de faltas correspondente ao espaço alocado: um *buffer* de 27 páginas causa 27 faltas na primeira volta do *loop*, enquanto que nas outras 499 voltas nenhuma falta ocorre porque os 27 mapeamentos já estão carregadas na TLB. Acima do limite de 32 páginas, espera-se que todos os acessos realizados em todas

as iterações do *loop* causem faltas, o que caracteriza o estado de *thrashing* na L1d-TLB. Isso é esperado porque, sendo a reposição por LRU, cada nova página mapeada além das 32 expurga um mapeamento que será reutilizado em breve.

Considerando que programa usa ao menos uma página adicional para sua pilha, pode-se considerar que o limite de *thrashing* na L1d-TLB é 31 páginas. Desta forma, o número de faltas esperado quando o programa de testes percorre de 22 a 31 faltas é o somatório de todos estes valores: 265 faltas. Na segunda metade do experimento, quando são percorridas de 32 a 41 páginas, o valor estimado é a soma dos valores do intervalo multiplicado pelo número de iterações do *loop*: 182.500 faltas. O total de faltas esperado é 182.765 faltas, que é a soma dos valores estimados nas duas etapas do teste.

A Figura 6.3 mostra a contagem de faltas na medida em que o tamanho do bloco percorrido aumenta de 22 a 41 páginas. Antes da medição atingir o ponto de interesse, após a carga do programa, faltas decorrentes da alocação da pilha e de dados do programa são registradas, totalizando 1.258 faltas. No momento em que o teste propriamente dito se inicia, as faltas crescem linearmente com o tamanho da área alocada pois ainda há registros livres na L1-TLB. Quando são alocadas mais de 28–30 páginas, a L1d-TLB entra em *thrashing*, como esperado. Ao final do teste, desprezados os valores anteriores à execução do programa, são contabilizadas 188.953 faltas, 3% acima do esperado.

É importante observar que a estimativa para o número de faltas não modela fielmente o que ocorre no processador, enquanto este executa o programa sobre um SO multitarefa, concorrentemente às demais aplicações. A deficiência da estimativa torna-se evidente quando se observa a ampliação do gráfico na Figura 6.4: cada flecha indica o tamanho do *buffer* a ser percorrido (número de páginas). É possível observar que entre o regime normal e o de *thrashing*, há um regime intermediário – entre 29 e 31 páginas – no qual a taxa de faltas aumenta muito, possivelmente em virtude dos conflitos de mapeamento entre o programa de teste e os demais processos, mas ainda não o suficiente para caracterizar o *thrashing*, evidenciado a partir do ponto em que o *buffer* atinge o tamanho de 32 páginas, quando a inclinação do gráfico se aproxima da vertical.

É possível observar degraus no gráfico da Figura 6.4, o que nos permite contar o

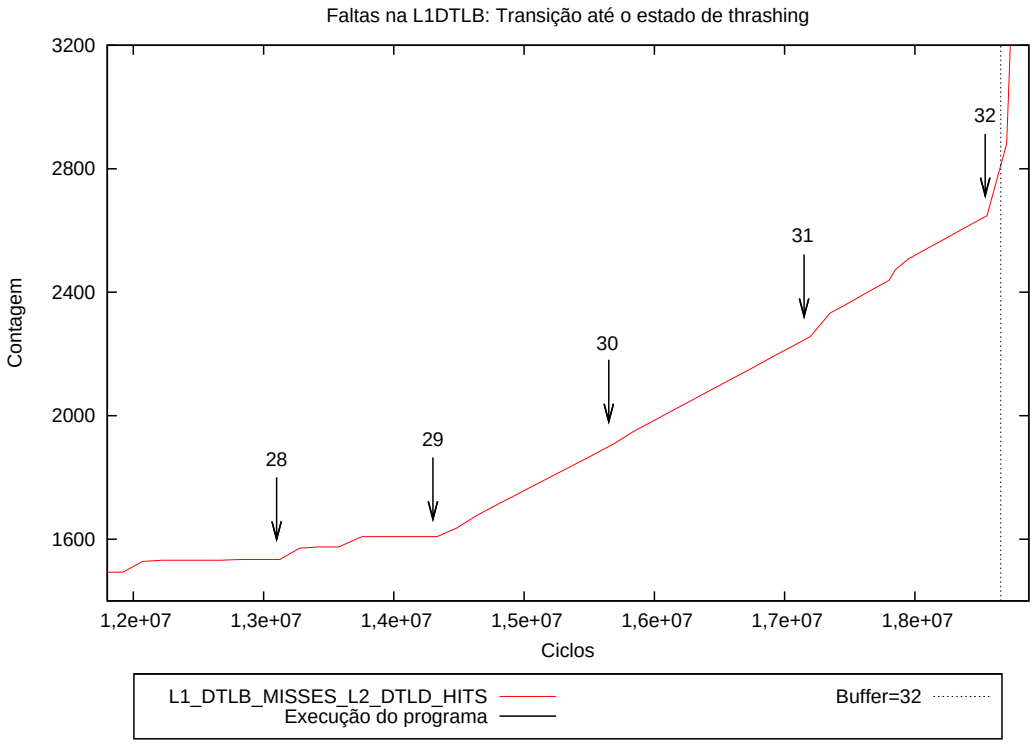

Figura 6.4: Transição entre o regime de funcionamento normal e o *thrashing* na L1 DTLB.

número de faltas para cada tamanho de *buffer* percorrido. A Tabela 6.2 apresenta a comparação entre valores contados nos degraus e os valores estimados para alguns trechos da execução do programa, e o total de faltas medidas no experimento.

Intervalo [pág]	Medido	Estimado
23	42	23
24	33	24
36	18.427	18.000
37	20.197	18.500
Total	188.953	182.765

Tabela 6.2: Faltas na L1d-TLB.

6.3.3 Buffer de 250 a 280 Páginas

O gráfico na porção superior da Figura 6.5 mostra as faltas ocorridas na L2 DTLB enquanto o programa percorre *buffers* com tamanho variando de 250 a 279 páginas. As linhas verticais pontilhadas demarcam o intervalo em que o programa foi executado enquanto a linha vertical sólida marca o ponto em que o *buffer* atinge o tamanho de 256

páginas.

No início da execução do programa, quando são alocadas 250 páginas, a contagem de faltas na L2 TLB aumenta de forma gradativa, antes mesmo do *buffer* atingir 256 páginas. Isso ocorre porque o programa começa a execução utilizando um tamanho de *buffer* próximo do limite da TLB, o que gera conflitos de mapeamento entre os endereços utilizados pelo *buffer* no programa e os endereços utilizados pelo SO ou pelos demais programas em execução na máquina.

A partir do momento em que o *buffer* atinge o tamanho da L2 TLB, a inclinação da linha aumenta a cada incremento no tamanho do *buffer*. A porção inferior da Figura 6.5 apresenta a ampliação de um trecho da execução no qual é possível identificar os pontos em que ocorrem aumentos no tamanho do *buffer*. No gráfico, cada degrau da linha equivale a um novo *buffer*, de tamanho $n + 1$, percorrido pelo programa.

Confirmando a relação entre os degraus no gráfico e o tamanho do *buffer* no programa, nesta figura é possível contar seis degraus na linha vermelha antes do ponto em que o número de páginas do *buffer* chega a 256, lembrando que o programa inicia a sua execução com um *buffer* de 250 páginas. Existem 30 degraus ao longo de toda a extensão da mesma linha, na porção superior da Figura 6.5.

Era esperado que a taxa de faltas na L2 TLB apresentasse um comportamento similar ao da L1 TLB na Figura 6.3. Entretanto, comparando as estruturas das duas TLBs, apresentadas no Capítulo 5, e verificando o gráfico de cada uma, conclui-se que a diferença entre o comportamento das duas TLBs ocorre porque, enquanto a L2 TLB possui associatividade quaternária, a L1 TLB possui associatividade total e reposição LRU.

Enquanto a L1 TLB se comporta como descrito na Seção anterior, a L2 TLB mapeia 256 páginas com associatividade 4, distribuindo-as em 64 linhas, cada uma com 4 possibilidades de lugar para inserir um dado mapeamento. O local de inserção varia de acordo com o método de reposição utilizado. No início da execução, por ainda haverem registros livres na TLB, quase não ocorrem faltas. À medida que o tamanho do *buffer* aumenta, a quantidade de faltas cresce lentamente.

Quando o tamanho do *buffer* supera o tamanho da L2 TLB, o reaproveitamento pro-

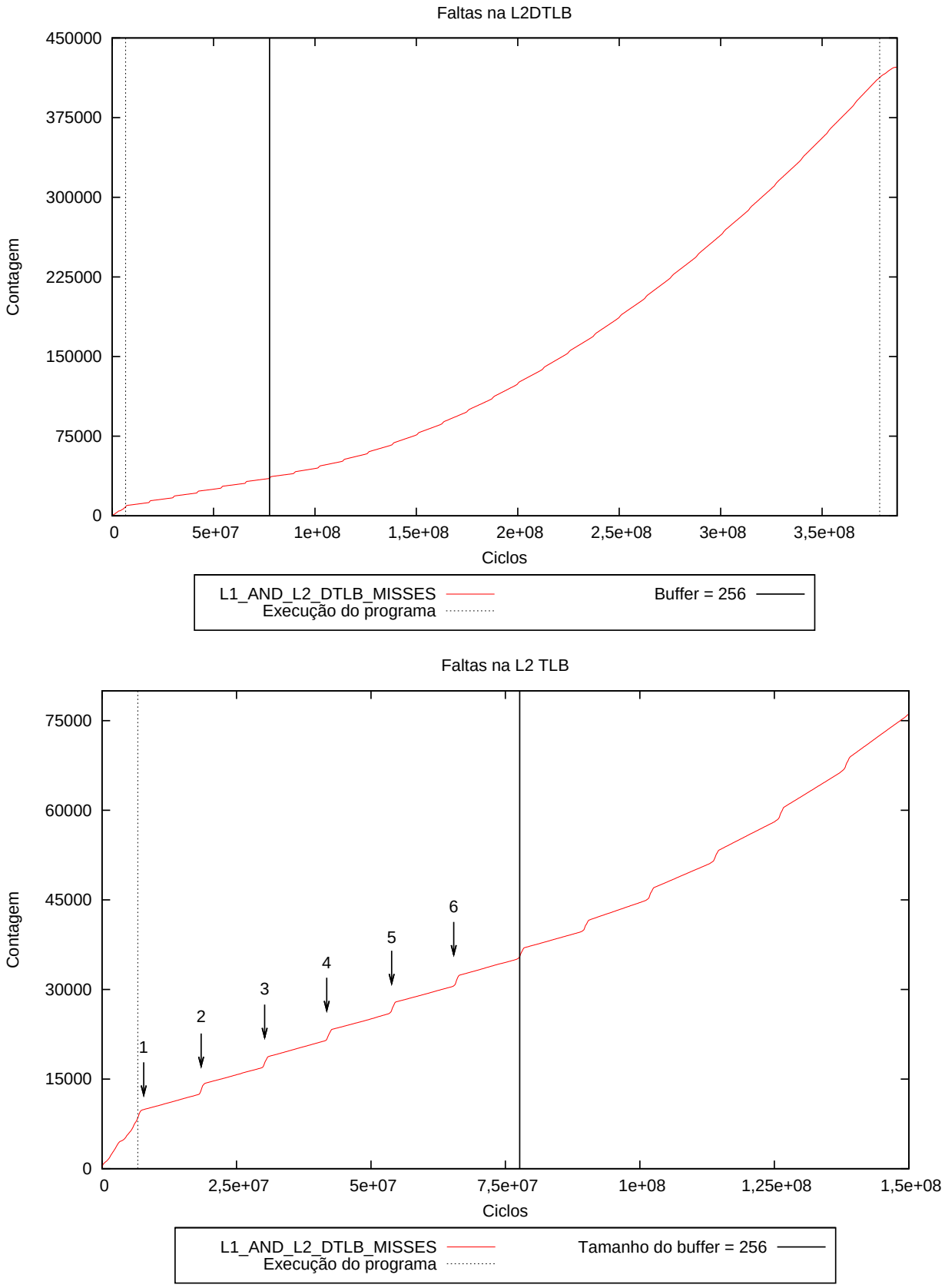

Figura 6.5: Acima: faltas na L2 DTLB percorrendo *buffers* entre 250 e 279 páginas. Abaixo: ampliação do gráfico demontrando a transição entre *buffers*.

porcionado pela associatividade quaternária da L2 TLB piora e a quantidade de faltas cresce gradativamente. Caso este experimento fosse executado para um intervalo maior de páginas, o ponto a partir do qual a linha plotada no gráfico se tornasse uma linha reta, similar àquela da Figura 6.3, representaria o ponto da execução em que a L2 TLB entrou em *thrashing*.

Assim como apresentado para a L1 TLB na Figura 6.3, o estado de *thrashing* na L2 TLB é caracterizado por um aumento no tamanho do *buffer* sem um aumento proporcional na inclinação da reta que representa a contagem de faltas nesta estrutura. Isto ocorre quando o limite da TLB é excedido ao ponto de não se conseguir mais reaproveitar nenhum endereço presente nesta estrutura. Deste instante em diante, cada acesso à página do *buffer* causa uma falta e a contagem de faltas é aproximadamente a mesma que a contagem de acessos até o término do teste.

CAPÍTULO 7

ANÁLISE DA MULTIPLICAÇÃO DE MATRIZES

Programas de multiplicação de matrizes são simples e têm um padrão de acessos à memória ruim. O acesso aos elementos dispostos em colunas causa faltas na *cache* a cada elemento acessado. No melhor caso, ao se multiplicar matrizes pequenas, o acesso aos elementos da primeira coluna de uma matriz faz com que os elementos seguintes sejam carregados automaticamente no bloco da *cache*, reduzindo as faltas compulsórias nos acessos subsequentes.

À medida que o tamanho das matrizes aumenta, os elementos dispostos nas colunas que já estão carregados na *cache* conflitam com novos elementos da mesma coluna quando a capacidade da *cache* é atingida. Por este motivo, o desempenho do programa piora drasticamente. Para evitar este comportamento indesejado, existem diferentes otimizações que reduzem conflitos de mapeamento e tiram proveito da hierarquia de memória a fim de melhorar o tempo de execução desses programas.

O experimento descrito nesta seção tem como objetivo demonstrar a medição de faltas nas *caches* L1 e L2 do sistema, durante a execução de programas de multiplicação de matrizes com diferentes níveis de otimização. Este experimento também avalia a interferência nas medidas causada pelo *Oprofile Estendido* durante a execução dos testes. A interferência é discutida em mais detalhe no Capítulo 9.

Apesar do experimento medir apenas eventos nas *caches* do sistema, a importância das TLBs no desempenho dos programas não foi ignorada. Para que a eficiência dos programas otimizados seja evidenciada é necessária a utilização de matrizes de tamanho superior à capacidade da L2 TLB do sistema. Dessa forma, o programa que fizer o melhor uso da hierarquia de memória do processador obterá desempenho superior.

Neste experimento foram utilizadas matrizes de 1.024 linhas por 1.024 colunas com elementos do tipo *double*, ocupando cada matriz $8 Mbytes$. A L2 TLB do Athlon possui

256 elementos [19] para páginas de $4Kbytes$, totalizando aproximadamente $1Mbyte$ de capacidade de mapeamento. A L1 *cache* do processador possui 512 linhas com dois blocos de 64 *bytes* cada, totalizando $64Kbytes$ de capacidade. No caso, uma linha da matriz ocupa 128 blocos da *cache* em linhas distintas.

Quatro programas para a multiplicação de matrizes foram selecionados para a realização deste teste, são eles: (i) Multiplicação de matrizes convencional, não otimizado; (ii) Multiplicação de matrizes com *padding*; (iii) Multiplicação de matrizes com blocagem; e (iv) Multiplicação de matrizes com *padding* e blocagem.

7.1 Estudo das Otimizações Testadas

Esta seção descreve o funcionamento dos programas de multiplicação de matrizes e suas otimizações, sob o ponto de vista da hierarquia de memória.

7.1.1 Multiplicação Convencional

Boa parte do mau desempenho da multiplicação de matrizes é decorrente de conflitos de endereçamento entre as matrizes: as posições de uma matriz que estão sendo acessadas num determinado instante podem estar mapeadas na mesma região da *cache* que as posições equivalentes das duas outras matrizes envolvidas na multiplicação. Nesse caso, cada acesso a um determinado elemento de cada matriz pode causar uma falta em qualquer das estruturas da hierarquia de memória, o que anula os benefícios de acesso rápido proporcionados por esta hierarquia.

A Figura 7.1 apresenta o padrão de referências acessados durante a multiplicação de matrizes quadradas de ordem 64 pelo programa normal em que $A \times B = C$. Cada linha dessas matrizes ocupa 8 linhas da *cache* e os pontos em vermelho, verde e azul indicam endereços das matrizes 'A', 'B' e 'C' respectivamente. O eixo vertical indica a linha da *cache* L1 acessada e o eixo horizontal representa o tempo: a cada elemento multiplicado, os endereços dos operandos são impressos. Todas as demais figuras referentes a acessos de endereços seguirem este padrão e utilizaram as mesmas matrizes.

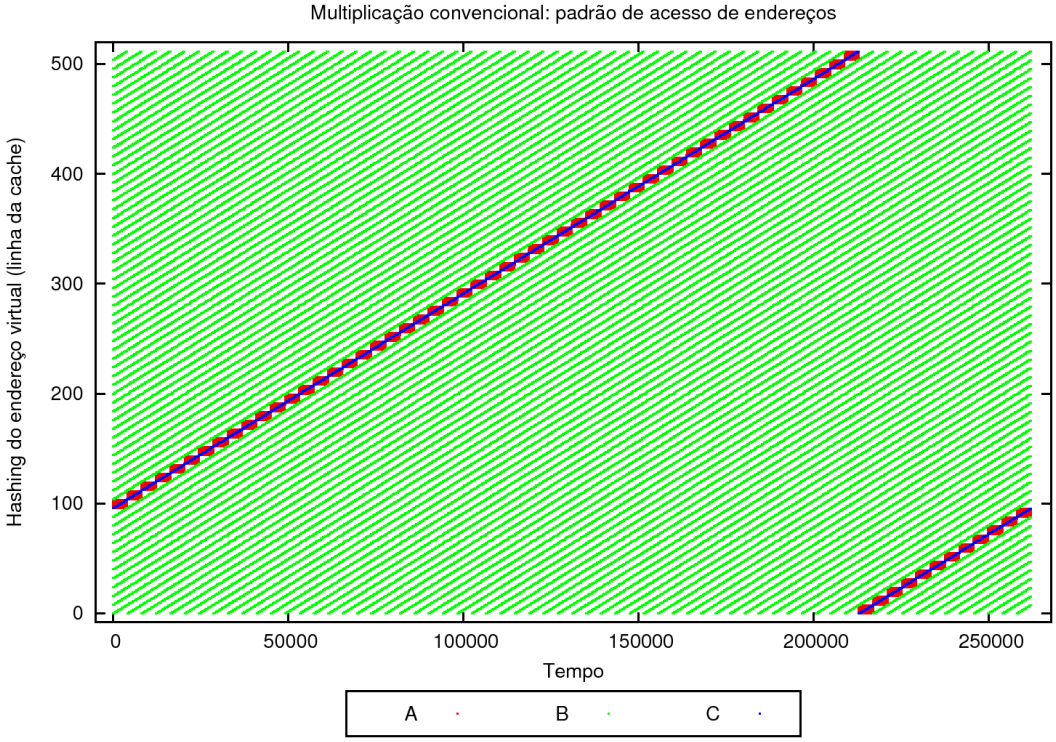

Figura 7.1: Endereços acessados na multiplicação de matrizes convencional.

No programa, para obter o valor de um elemento da coluna 'C', é preciso que uma linha inteira de 'A' multiplique uma coluna inteira de 'B'. Na Figura 7.1 observa-se que enquanto as colunas de 'B' são acessadas, os pontos do gráfico que representam o endereço de 'B' são plotados na vertical pois cada coluna se encontra numa linha de *cache* diferente.

Enquanto isso, à medida que a mesma linha de 'A' é acessada, a variação vertical dos pontos de 'A' é de 8 linhas, ou a região ocupada por uma linha da matriz na cache. A cada ciclo de multiplicação de linha de 'A' por coluna de 'B', um endereço de 'C' é acessado. Finalmente, a cada 8 endereços sequenciais acessados para 'C', a linha de cache do endereço de 'C' muda.

Este padrão de acessos se repete durante toda a execução do programa e os pontos do gráfico nos quais os endereços das três matrizes são plotados no mesmo lugar indicam conflitos de endereçamento. Neste gráfico especificamente, nota-se as matrizes 'A' e 'C' estão mapeadas nas mesmas linhas da *cache*, o que causa um constante conflito de endereços entre as duas matrizes. Dependendo do ciclo da multiplicação também ocorrem conflitos de mapeamento entre as três matrizes.

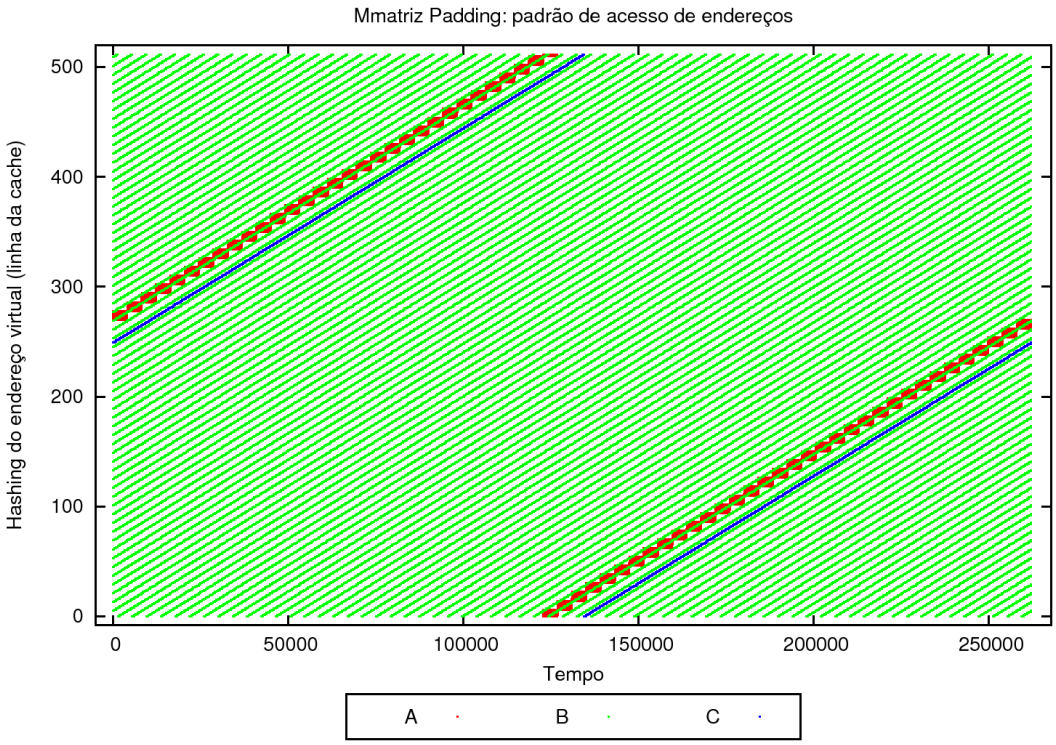

Figura 7.2: Endereços acessados na multiplicação de matrizes com *padding*.

7.1.2 Multiplicação com Padding

Para resolver o problema dos conflitos de mapeamento, utiliza-se a técnica de *padding*, que consiste em separar as faixas de endereços das matrizes de forma que elas sejam mapeadas em diferentes linhas de *cache*. Para isso, entre cada declaração de matriz que será multiplicada é declarado um vetor do tamanho de uma ou mais linhas de *cache* que não é utilizado durante a execução do programa. Os vetores vazios deslocam o mapeamento dos endereços das matrizes, alterando a linha de cache onde elas serão mapeadas e reduzindo os conflitos de memória e, consequentemente, o desempenho do programa melhora. A Figura 7.2 apresenta o padrão de acesso de endereços do programa que realiza a multiplicação de matrizes com *padding*. Nesta figura é possível observar que a matriz 'C' está mapeada em linhas de *cache* distintas das de 'A', devido à separação das diagonais que representam os endereços dessas matrizes, diferentemente da Figura 7.1.

7.1.3 Multiplicação com Blocagem

Outra forma de melhorar o desempenho da multiplicação de matrizes é aumentar o aproveitamento das *caches* do sistema. Para se obter um linha de 'C', uma mesma linha de 'A' multiplica cada coluna de 'B'. Para a próxima linha de 'C', a mesma linha de 'A' multiplica cada coluna de 'B' novamente. Considerando o caso em que as matrizes sejam grande demais para a L1, periodicamente os dados das matrizes são expurgados da *cache* e têm de ser recuperados na memória principal a cada ciclo de acessos, o que piora o desempenho já que acessos à memória são custosos.

A *blocagem* tira proveito da hierarquia de memória dividindo as matrizes em blocos de tamanhos menores, compatíveis com o da *cache*, otimizando a quantidade de operações realizadas entre cada busca de elementos das matrizes na memória principal. O código da multiplicação com blocagem utilizado é o mesmo apresentado em [17].

O comportamento dos acessos aos endereços é similar ao da multiplicação de matrizes convencional mas o espaço de memória percorrido é menor. Na Figura 7.3, é possível observar que o espaço vertical representado pelos acessos às colunas de 'B' ocupa apenas 256 linhas a cada bloco utilizado ao invés de toda a cache. Isso ocorre porque apenas um bloco da matriz 'B' está sendo acessado, coluna por coluna, para cada linha dos blocos de 'A' e 'C' que são percorridas. Nota-se também que o tamanho das marcas no gráfico que representam os acessos a 'A' e 'C' são menores do que as marcas apresentadas na Figura 7.1, também por causa da blocagem já que a linha do bloco é mais curta do que a linha da matriz. Ainda é possível visualizar a troca de blocos em 'C' durante a execução do programa.

7.1.4 Multiplicação com Padding e Blocagem

É possível combinar as duas técnicas, separando a faixa de endereços das matrizes na declaração para evitar conflitos de mapeamento e utilizando a técnica de blocagem para otimizar a utilização da hierarquia de memória. O padrão de acessos à memória durante a multiplicação das matrizes com a combinação de *padding* e blocagem pode ser observado na Figura 7.4, na qual é possível verificar o mesmo comportamento da Figura 7.3 mas

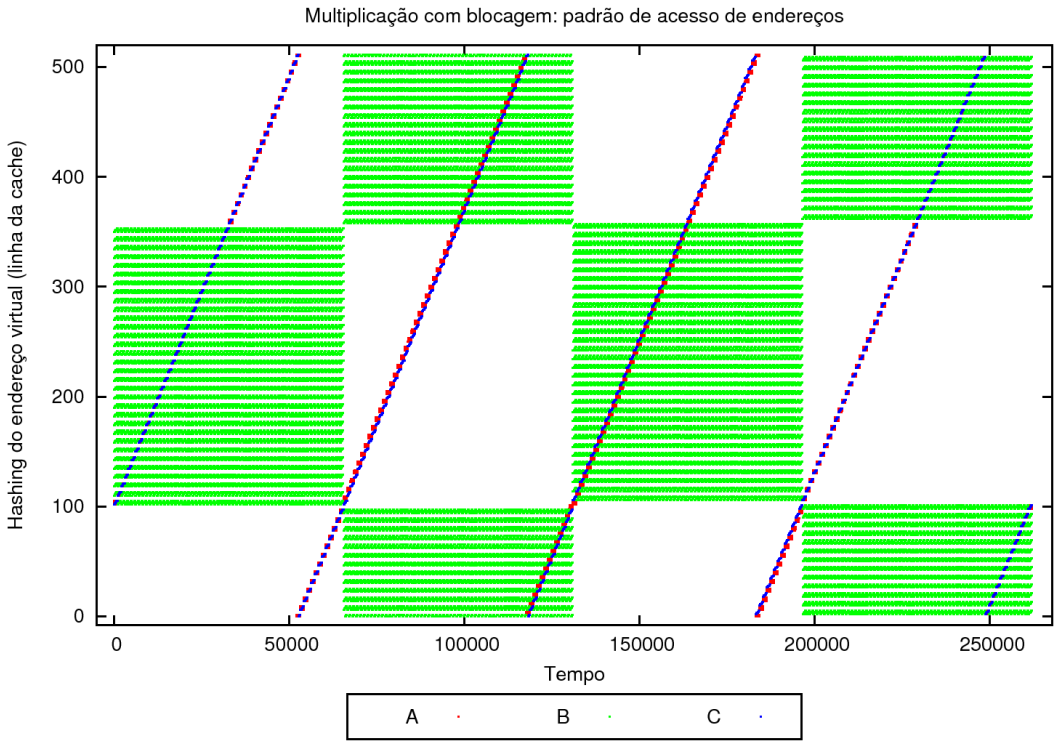

Figura 7.3: Endereços acessados na multiplicação de matrizes com blocagem.

com os endereços das linhas de bloco da matriz 'C' separados dos da matriz 'A'. O código do programa de multiplicação com blocagem e *padding* está no Apêndice B.

7.2 Preparação do Experimento

Esta seção descreve os valores de bloco e *padding* utilizados nos experimentos de medida que definiram os casos a serem estudados, assim como as configurações utilizadas no *Oprofile Estendido* para cada programa monitorado.

7.2.1 Parâmetros dos Programas

Três fatores foram levados em conta para a preparação dos testes: o tamanho de uma linha da *cache* do sistema, o tamanho da *cache* do sistema e o número de linhas de *cache* que uma linha de matriz ocupa.

O tamanho da linha da *cache* do sistema foi utilizado para determinar o tamanho mínimo do *padding* utilizado na separação das matrizes: o tamanho do *padding* deve ser

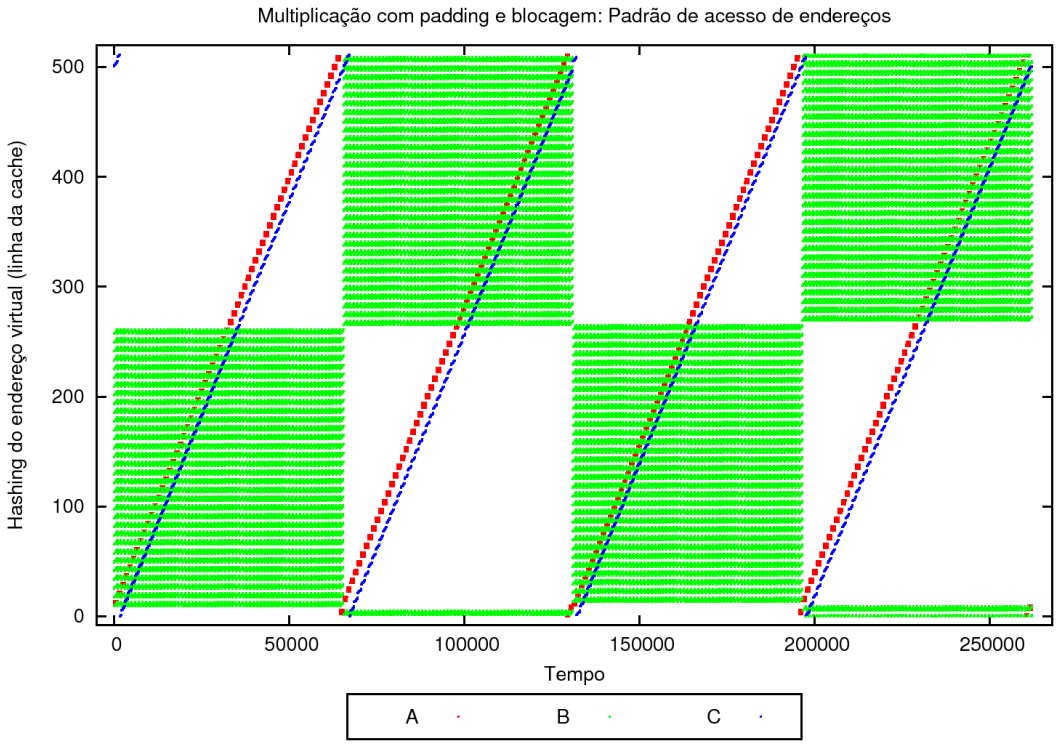

Figura 7.4: Endereços acessados na multiplicação de matrizes com *padding* e blocagem.

sempre um múltiplo da linha de *cache*, que tem 64 *bytes* ou 8 *doubles*. Dessa forma, cada início de linha da matriz coincide com um início de linha de *cache*, o que otimiza o acesso aos dados, uma vez que a carga do primeiro elemento de uma linha de *cache* resulta na carga antecipada de todos os elementos da mesma linha. Este valor também foi utilizado para determinar o passo do aumento do bloco no teste de blocagem: blocos de tamanho múltiplo da linha de *cache* também tiram proveito da busca antecipada e obtém um desempenho superior.

O tamanho da *cache* do sistema foi utilizado para determinar o melhor tamanho de bloco para o programa. O ideal é que a *cache* do sistema comporte os três blocos acessados durante cada etapa da multiplicação. Neste caso, o tamanho máximo que um bloco pode ter é 48 elementos (48 linhas $\times$ 48 colunas $\times$ 3 blocos $\times$ 8 *bytes* por elemento = 55.296 *bytes*). O próximo valor de bloco a ser testado seria de 56 elementos, o que totaliza um espaço de 75.264 *bytes*, maior que capacidade da L1 que é de 65.536 *bytes*.

O número de linhas de *cache* ocupadas por uma linha da matriz pode ser utilizado para definir um tamanho de *padding* que seja mais apropriado do que o de apenas uma linha

da *cache*. Se o *padding* não for grande o suficiente para separar as linhas das matrizes 'A' e 'C', haverá na cache uma área de sobreposição entre as duas matrizes que ocasionará faltas por conflito durante a multiplicação. Se cada linha da matriz ocupa 128 linhas de *cache*, este valor seria suficiente para evitar essa sobreposição. Porém, sendo 128 uma potência de 2, existe a possibilidade dos mapeamentos das matrizes coincidirem na mesma região da *cache*. Por isso, foram utilizados os valores de *padding* de 133 linhas entre 'A' e 'B' e 135 linhas entre 'B' e 'C'.

Definidos os valores, blocos variando de 0 a 48 elementos e *padding* escolhido dentre 0, 1 e 133 *e* 135 linhas, o desempenho dos programas foi medido para cada combinação de parâmetros. Medidas de tempo foram realizadas durante vinte execuções de cada um dos programas com o utilitário `time`. A média dessas medidas serviu para a definição do tamanho ideal de *padding* e de bloco utilizados nos programas, de forma que fossem obtidos os menores tempos de execução. A partir desses resultados, os programas com melhor e pior desempenho foram selecionados para a análise com o *Oprofile Estendido*.

Os resultados das medidas de tempo podem ser vistos na Tabela 7.1, em que a primeira coluna apresenta o tamanho dos blocos utilizados, a segunda coluna apresenta os resultados de tempo para execuções sem *padding*, a terceira coluna apresenta os resultados para *padding* de uma linha de *cache* entre cada matriz e a quarta coluna o resultado para *padding* de 133 linhas de *cache* entre a primeira e a segunda matriz e de 135 entre a segunda e a terceira matriz.

A primeira linha da Tabela 7.1 apresenta os tempos em segundos para a multiplicação de matrizes convencional, sem blocagem, para os diferentes valores de *padding* utilizados. Dos demais valores da tabela é possível concluir que nem sempre há ganho de desempenho utilizando *padding* diferente de uma linha e que o tamanho ideal de bloco está entre 16 e 32 elementos: valores de bloco maiores ou menores que estes degradam o desempenho do programa neste processador.

Para a análise com o *Oprofile Estendido*, cada experimento foi realizado 20 vezes e os resultados apresentados nos gráficos e tabelas das próximas seções representam a média das 20 execuções, exceto quando explicitamente mencionado. Os casos selecionados

	Padding		
Bloco	0	1	133 *e* 135
0	214,7	99,8	98,9
8	30,6	30,1	30,3
16	26,6	24,1	24,2
24	25,5	22,7	22,8
32	25,7	22,8	22,3
40	69,6	51,8	25,4
48	107,3	97,7	101,3

Tabela 7.1: Tempo de execução da multiplicação de matrizes variando bloco e *padding*.

para a análise com o *Oprofile Estendido* pelo critério do melhor e pior tempo foram a multiplicação convencional, sem otimizações e a multiplicação com *padding* de 133 *e* 135 linhas e blocos de 32 elementos.

7.2.2 Configurações do OProfile Estendido

Os eventos selecionados para a observação nesses experimentos foram: (i) `Data_Cache_Accesses`; (ii) `Data_Cache_Refills_from_L2` e; (iii) `Data_Cache_Refills_from_System`. O contador de ciclos de relógio `Cpu_Clock_Unhalted` foi configurado de acordo com o tempo de execução de cada um dos experimentos de forma que a quantidade de informação registrada pelo *Oprofile Estendido* seja equivalente.

Sendo o tempo médio de execução do programa de multiplicação de matrizes normal aproximadamente 9 vezes maior do que o tempo médio de execução da multiplicação de matrizes otimizada, o contador de ciclos foi configurado para interromper o processador a cada 1.800.000 ciclos ativos durante a multiplicação normal. Na multiplicação com *padding* e blocagem, o valor de *overflow* do contador foi ajustado em 250.000 ciclos.

A diferença entre o valor de leitura da função `readtsc()` e os valores dos contadores registrados no ciclo de relógio mais próximo – do início ou do término da execução – do trecho do programa que se quer monitorar não apresenta problemas para a visualização dos dados nos gráficos. No entanto, dependendo da frequência da amostragem dos contadores, pode-se perder precisão na medida da contagem total dos eventos.

Para contornar esse problema e obter as contagens dos eventos monitorados no ponto

em que ocorreu a chamada à função `readtsc()`, é feita uma aproximação calculando a variação do número de eventos monitorados em relação à variação do número de ciclos registrados. A variação do número de eventos é então multiplicada pela diferença entre a última leitura do TSC ocorrida antes da chamada à `readtsc()` e o valor retornado por esta função. Dessa forma, é esperado que se mantenha um grau razoável de confiabilidade na medida mesmo utilizando taxas de amostragem distintas.

7.3 Resultados dos Experimentos

Esta seção apresenta os resultados dos experimentos assim como algumas soluções adotadas na apresentação dos mesmos.

7.3.1 Comportamento dos Programas

A multiplicação convencional apresentou um comportamento bastante estável em todas as execuções. Aparentemente, o uso da hierarquia de memória por esse programa é ruim a ponto do seu desempenho não ser prejudicado por interferências do Sistema Operacional (SO). No caso da multiplicação com blocagem e *padding*, o tempo de execução varia bastante entre as medidas, em razão deste programa tirar um bom proveito da hierarquia de memória e consequentemente ser mais sensível às interferências do SO.

Para contornar o problema da instabilidade nas medidas de tempo da multiplicação com *padding* e blocagem, foram realizadas, ao invés de 20, 60 medidas de execução deste programa com o *Oprofile Estendido*. A execução com os valores de ciclos e contagem de faltas mais próximas às das médias das 60 medidas foi utilizada para plotar os gráficos apresentados nessa e nas seções que referenciem este experimento. Os demais dados apresentados, contagens e taxas, são referentes à média das contagens das 60 execuções.

Nos gráficos que apresentam as medidas do *Oprofile Estendido* para cada programa monitorado, a primeira linha da legenda representa os acessos à *cache* L1, a segunda linha representa as faltas ocorridas na L1 que corresponderam a *acertos* na L2, a terceira linha representa as faltas da L1 que ocasionaram *faltas* na L2, a quarta linha indica o início e

o final da execução do programa e os eixos X e Y representam os ciclos de relógio e a contagem de eventos respectivamente.

Os valores apresentados ao lado de cada linha na legenda indicam os fatores de ajuste de cada uma, que é o valor pelo qual a linha deve ser multiplicada para que ela entre na sua escala normal. A linha de evento sem fator de ajuste apresenta a contagem do evento obtida diretamente através do *Oprofile Estendido*.

7.3.2 Contagem dos Eventos

Apesar do programa de multiplicação otimizado ser 8,5 vezes mais rápido do que o programa normal, os gráficos plotados a partir do registro do *Oprofile Estendido* para os dois programas são praticamente iguais, a menos dos valores das contagens de eventos e os fatores de ajuste apresentados em cada legenda dos gráficos da Figura 7.5. Em princípio, não há evidência visual que justifique a diferença de desempenho entre os programas além da discreta separação entre a linha que conta os acessos à *cache* e as linhas que contam as faltas nessa estrutura, que pode ser observada na porção inferior da Figura 7.5.

Os valores aproximados das medidas obtidas no registro do *Oprofile Estendido* são mostrados na Tabela 7.2. Nessa tabela, a primeira coluna apresenta os tipos de programas estudados, a segunda coluna o total de ciclos, a terceira coluna os acessos à *cache* (`Data_Cache_Accesses`), a quarta e quinta colunas apresentam as recargas oriundas da L2 (`Data_Cache_Refills_from_L2`) e do sistema (`Data_Cache_Refills_from_System`) respectivamente. O total de faltas na L1 é calculado a partir da soma das recargas provenientes da L2 e do sistema.

Analisando as contagens dos eventos, a multiplicação normal realiza pouco mais do que a metade dos acessos à *cache* efetuados pela multiplicação otimizada. Ainda assim, a multiplicação normal precisa de uma quantidade de ciclos 8,5 vezes maior que a utilizada

Programa	Ciclos	Acessos à L1	Faltas na L1	L1 ← L2	L1 ← sistema
Convencional	$4{,}32 \cdot 10^{11}$	$1{,}66 \cdot 10^{10}$	$2{,}15 \cdot 10^{9}$	$1{,}08 \cdot 10^{9}$	$1{,}07 \cdot 10^{9}$
Otimizado	$5{,}05 \cdot 10^{10}$	$2{,}96 \cdot 10^{10}$	$1{,}06 \cdot 10^{9}$	$1{,}03 \cdot 10^{9}$	$3{,}4 \cdot 10^{7}$

Tabela 7.2: Contagem dos eventos para os programas de multiplicação de matrizes.

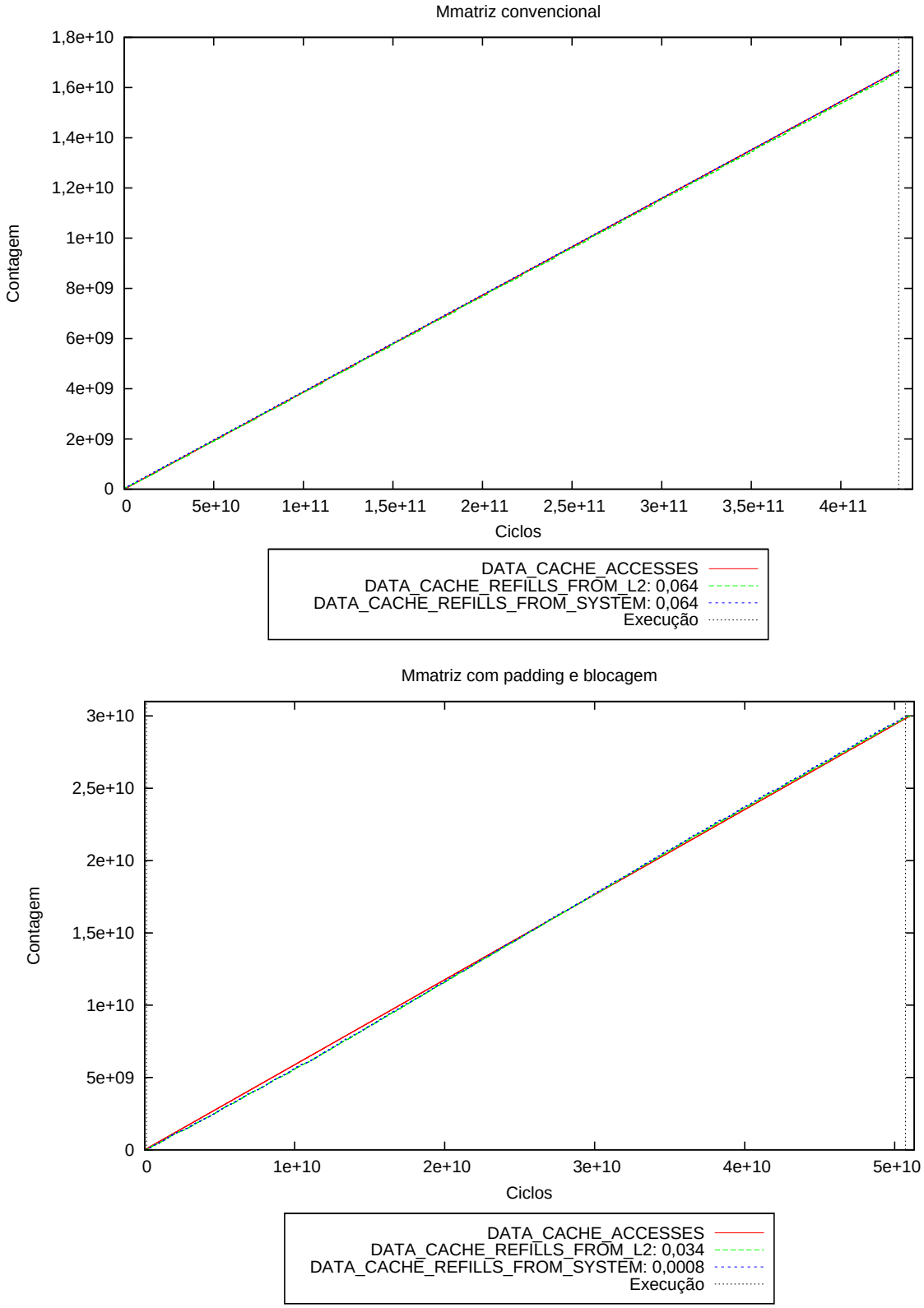

Figura 7.5: Monitoramento da cache de dados durante a multiplicação convencional (acima) e a multiplicação com *padding* e blocagem.

pelo programa otimizado. No caso, a multiplicação normal realiza acessos à *cache* a uma taxa de 0,038 acessos para cada ciclo de relógio enquanto a multiplicação otimizada atinge a taxa de 0,586 acessos por ciclo. Este aumento na taxa de acessos ocorre porque, na multiplicação otimizada, além dos dados serem acessados de maneira diferente, a maioria das faltas na *cache* L1 resultam em acertos na *cache* L2. Sendo os acessos à L2 mais rápidos do que os acessos à memória principal, o número de ciclos em que o processador fica parado aguardando os dados da memória é menor no programa otimizado.

Na multiplicação normal, 12,9% dos acessos à *cache* L1 realizados resultam em faltas. Deste percentual, 50,4% são repostos pela L2 e 49,6% pelo sistema de memória. No caso do programa otimizado, 3,5% dos acessos à L1 resultam em faltas, das quais 97,1% são repostos pela L2 e 2,9% pelo sistema, indicando uma melhora significativa na utilização da L2 e, consequentemente, no desempenho do programa. O ganho de desempenho ocorre porque o processador é capaz de continuar executando outras instruções, independentes dos dados faltantes, enquanto espera por uma recarga da L2 [17].

Os dados apresentados nesta seção foram obtidos a partir do registro do *Oprofile Estendido*. As mesmas informações também podem ser obtidas com o *Oprofile* convencional. Entretanto, há uma variação entre as medidas destas ferramentas – que ocorre em função da natureza do método de amostragem de cada uma – que pode alterar os resultados, comprometendo a precisão da medida do *Oprofile*. As diferenças entre as medidas realizadas com o *Oprofile Estendido* e o *Oprofile*, bem como as suas causas são discutidas no Capítulo 10.

7.3.3 Utilização da Hierarquia de Memória

Ampliando um pequeno trecho da execução de cada um dos gráficos da Figura 7.5, é possível observar as diferenças entre o comportamento de cada programa durante a fase da execução mostrada no intervalo da ampliação. Na multiplicação convencional, mostrada na Figura 7.6 (topo) nota-se que enquanto os acessos à cache são relativamente constantes, os *refills* provenientes da L2 ocorrem numa taxa mais baixa do que os 6,4% indicados no fator de ajuste, salvo quando ocorrem pequenos degraus verticais na linha que representa

este evento.

A linha que representa as recargas à L1 provenientes do sistema apresenta uma elevação pouco superior aos 6,4% indicados na legenda, mas ela apresenta discretos degraus horizontais nos mesmos pontos em que a linha que representa as recargas da L2 apresenta os degraus verticais. À exceção dessa perturbação periódica, o comportamento dos eventos não apresenta qualquer variação devido ao padrão de acessos à memória realizado pelo programa. A quantidade de conflitos de endereços ocorridos ao longo da execução do programa é constante.

Uma análise do registro do *Oprofile Estendido* nos trechos onde ocorrem as perturbações indica que esta alteração no comportamento dos programas é decorrente de uma descarga (*dump*) dos *buffers* do *Oprofile Estendido*. Nesta seção, esta interferência será utilizada como referência na comparação entre a utilização da memória dos dois programas. No Capítulo 9 será apresentada uma análise mais detalhada da interferência.

No caso da multiplicação otimizada, enquanto os acessos à *cache* ocorrem com taxa aparentemente constante, as recargas provenientes da L2 oscilam na faixa dos 3,4% (porção inferior da Figura 7.6). É possível perceber um padrão na oscilação da linha, ocasionado pelas fases da blocagem durante a execução do programa. Um comportamento semelhante é apresentado pela linha que representa as recargas da L1 pelo sistema, que oscila na faixa de 0,08% em relação aos acessos à *cache*.

Esta oscilação reflete o comportamento das faltas na cache à medida que as colunas de um mesmo bloco $b1$ são multiplicadas pelas linhas de outro bloco $b2$. Enquanto a faixa de endereços ocupada por $b1$, que está sendo acessada coluna a coluna permanece constante, o bloco $b2$, que é percorrido linha por linha, ocupa da primeira à última linha da matriz. Dessa forma, o tempo gasto para cada linha de $b2$ que é carregada na *cache* é compensado multiplicando-se a linha pelas colunas de *b1* que já estão carregadas na *cache*.

Devido ao comportamento do programa e ao tamanho das matrizes, ocorrem conflitos de endereçamento entre as próprias colunas de $b1$. Pode também ocorrer de a linha de $b2$ estar no mesmo endereço de algum outro dado utilizado. Esses conflitos de endereçamento refletem os trechos com maior inclinação nas linhas que indicam as recargas na L1. Por

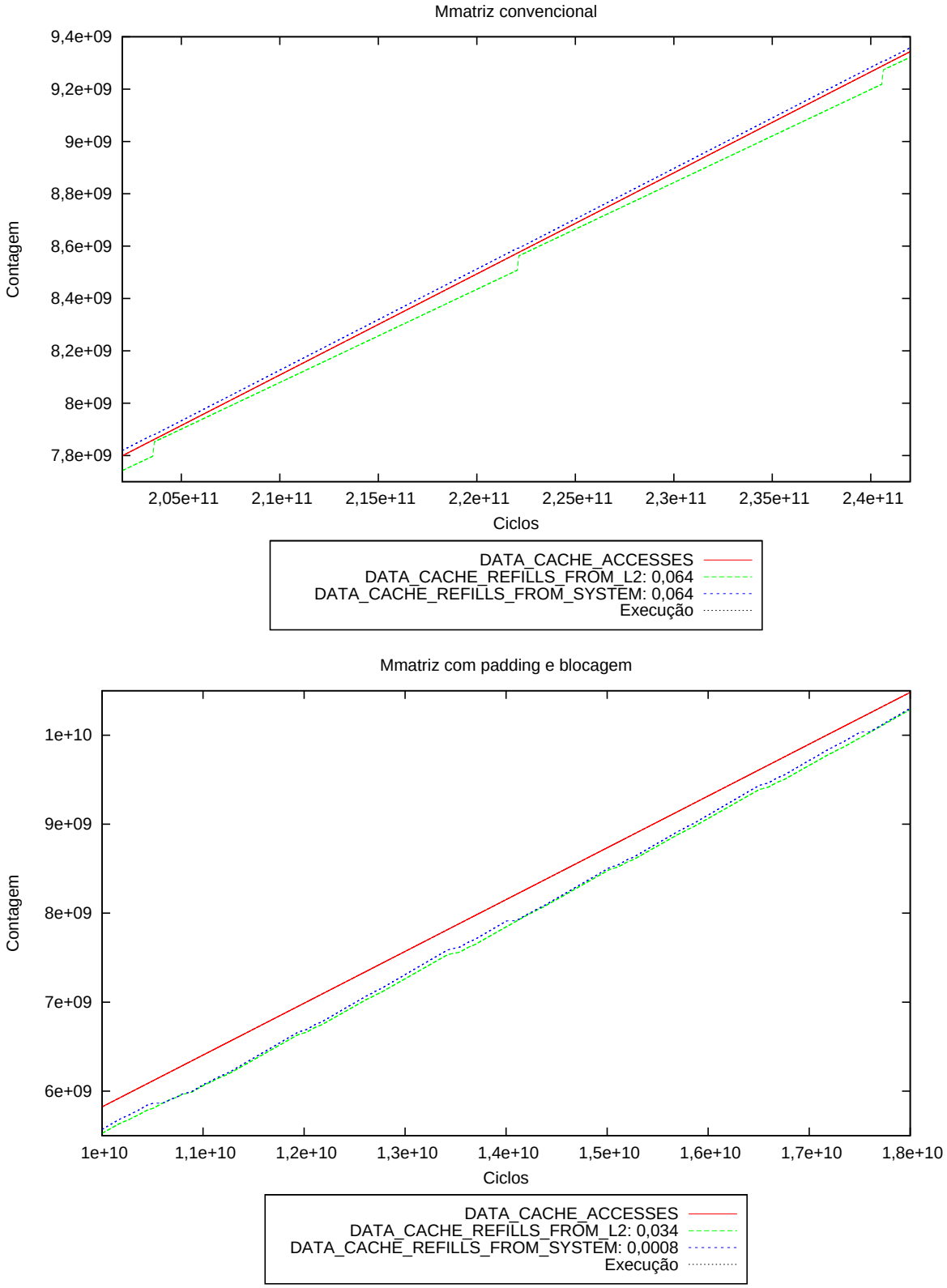

Figura 7.6: Comportamento dos eventos durante a multiplicação convencional (acima) e a otimizada.

serem usados conjuntos pequenos de dados na blocagem, a grande maioria das faltas na L1 são preenchidas por dados da L2.

Além desses conflitos, à medida que os blocos da matriz que estão sendo percorridos linha a linha são trocados por outros, podem ocorrer sobreposições entre os endereços de *cache* ocupados pelos os novos blocos da matriz e os endereços ocupados por $b1$. Ao longo da execução do programa, à medida que os blocos das matrizes se movimentam pelas linhas da *cache*, as faltas causadas pelos blocos sobrepostos causam a discreta ondulação nas linhas que medem as reposições das faltas na L1, mostrada na porção inferior da Figura 7.5.

É possível ainda observar que, além do padrão ondulado apresentado nas linhas que contam as recargas na L1, há também a interferência periódica causada pelo *Oprofile Estendido* – semelhante à interferência da multiplicação convencional – representada por um degrau horizontal na linha das recargas vindas do sistema e por uma inclinação contínua na linha das recargas pela L2 nos mesmos pontos. A porção superior da Figura 7.7 mostra a ampliação de um trecho da interferência ocorrida no programa de multiplicação convencional, onde é possível observar claramente que enquanto os acessos à *cache* permanecem estáveis, o número de recargas provenientes da L2 aumenta rapidamente e as recargas do sistema permanecem estáveis durante o mesmo período.

Uma ampliação semelhante para a multiplicação otimizada é mostrada na parte inferior da Figura 7.7. Os eventos nesse gráfico aparentemente apresentam o mesmo comportamento: há um aumento, ainda que leve, na inclinação da linha que mede os acertos na L2 e uma redução na inclinação da linha que mede as faltas na L2. No trecho em que ocorre essa oscilação, a linha que mede os acessos à *cache* permanece constante.

Analisando a Figura 7.7 é possível comparar visualmente a utilização da *cache* L2 pelos dois programas, tomando como referência a perturbação causada pelo *Oprofile Estendido*. Observando-se o aumento da inclinação na linha que mede as recargas provenientes da L2 durante o *dump* do *Oprofile Estendido*, constata-se que utilização da L2 na multiplicação convencional é pior do que a utilização da L2 durante a execução do *Oprofile Estendido*. No caso da multiplicação otimizada, ocorre uma perturbação nas recargas da L2, mas a

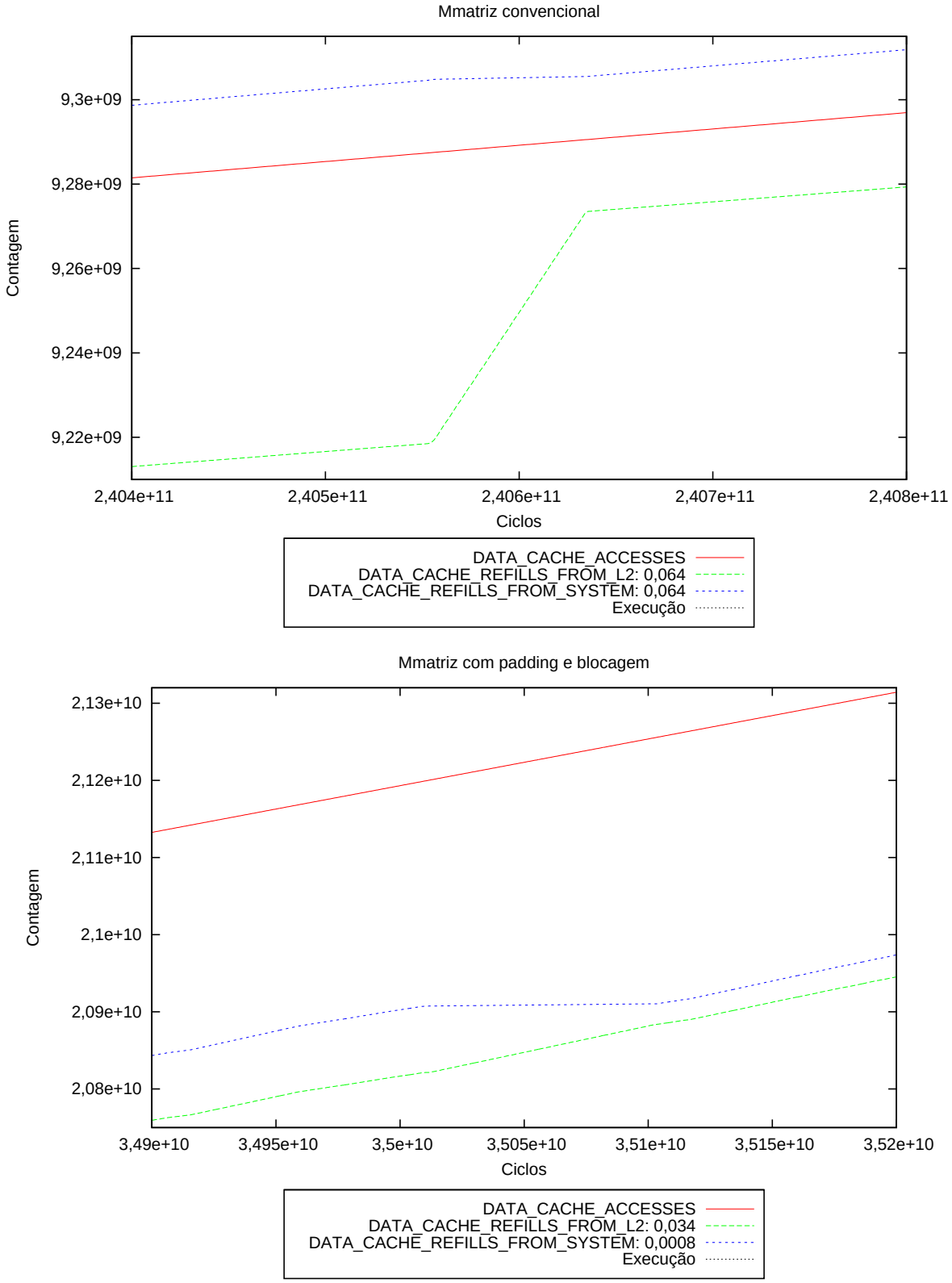

Figura 7.7: Interferência durante a multiplicação de matrizes convencional (topo) e a otimizada.

inclinação da linha que mede estas recargas quase não se altera. Disto constata-se que a utilização da L2 é similar entre o programa otimizado e o *Oprofile Estendido*. Maiores detalhes referentes aos valores das taxas dos eventos medidos em cada programa são apresentados na próxima seção.

Todos os dados e figuras apresentados nesta seção foram obtidos a partir do registro do *Oprofile Estendido* – não sendo possível obter estes mesmos dados com o *Oprofile* – e auxiliam a compreensão do funcionamento da hierarquia de memória do processador durante a execução de cada um dos programas de multiplicação de matrizes. Com a distribuição temporal da contagem de eventos, é possível observar e deduzir os estágios da multiplicação das matrizes bem como o comportamento da hierarquia de memória em cada um destes estágios.

7.3.4 Análise das Taxas dos Eventos Medidos

Os resultados apresentados na Seção 7.3.2 mostram as contagens de acessos, de faltas e de algumas taxas obtidas durante a execução dos dois experimentos sem apresentar a variação delas ao longo do tempo. Os resultados da Seção 7.3.3 apresentam essa variação nos eventos ao longo do tempo, mas devido ao fator de ajuste das linhas, é difícil estabelecer valores mais precisos nas variações dos eventos ao longo da execução dos programas.

Para complementar as informações das seções anteriores, as taxas das medidas dos experimentos foram calculadas, dividindo as variações (Δ[1]) de um evento por outro, seguindo as mesmas métricas dos resultados já apresentados: a taxa dos acessos à cache é Δ`Data_Cache_Accesses`/Δ`Ciclos`, a de acertos na L2 é Δ`Data_Cache_Refills_from_L2`/Δ`Data_Cache_Accesses`, enquanto a taxa das faltas na L2 é Δ`Data_Cache_Refills_from_System`/Δ`Data_Cache_Accesses`.

Em todos os gráficos apresentados nesta seção, o eixo Y apresenta a variação da taxa da medida, o eixo X representa a contagem dos ciclos de relógio ao longo do tempo, os pontos correspondem aos valores da métrica para cada Δ ciclo amostrado e as linhas verticais pontilhadas indicam o início e o fim da execução do programa.

[1] $\Delta x = ...x_2 - x_1, x_3 - x_2...$, $\Delta y = ...y_2 - y_1, y_3 - y_2...$, sendo x e y eventos plotados nos eixos x e y.

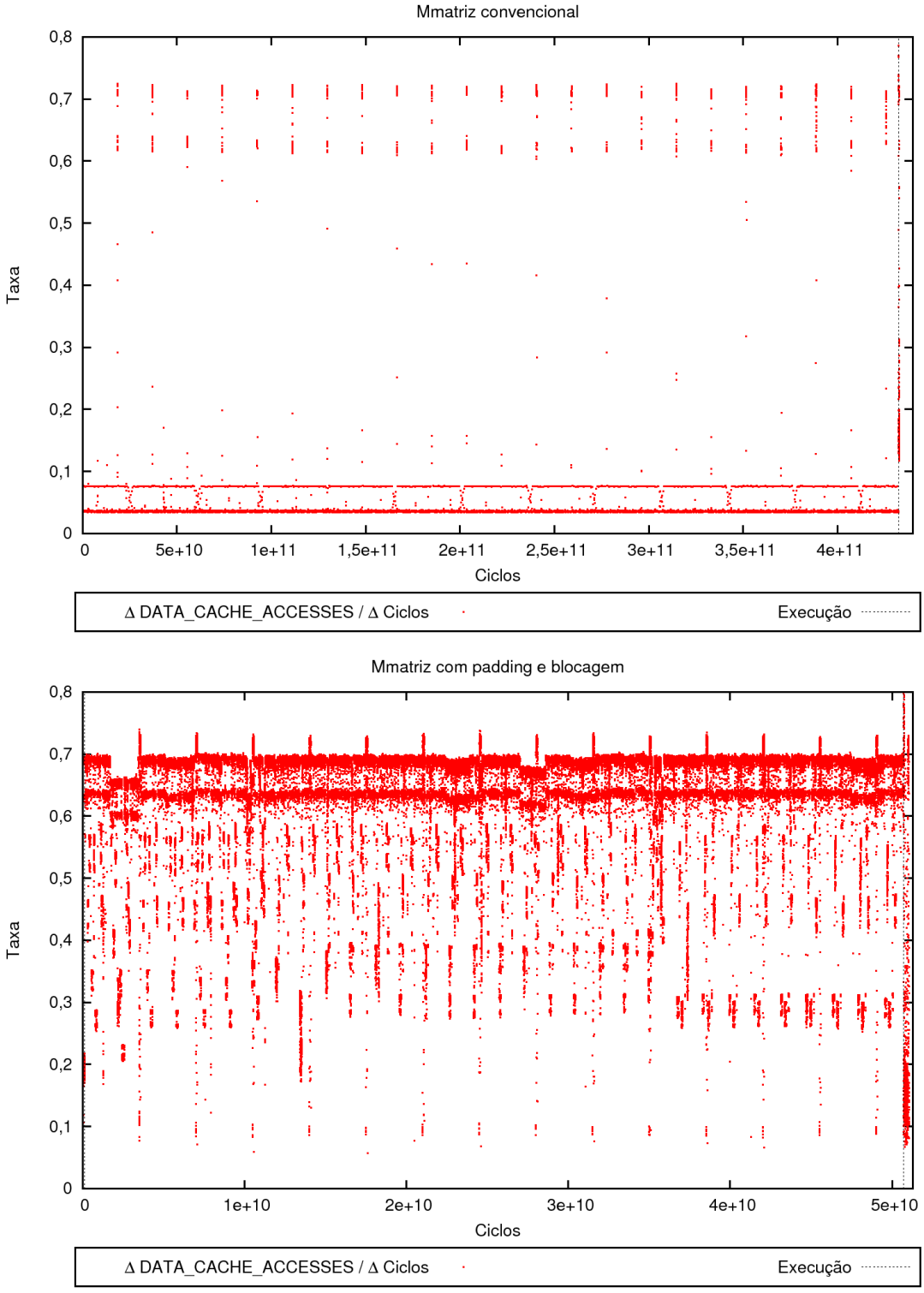

Figura 7.8: Acessos à *cache* por ciclos de relógio na multiplicação convencional (acima) e na com *padding* e blocagem.

A análise da taxa dos acessos à *cache* por intervalo pode ser observada no topo da Figura 7.8 para a multiplicação convencional. O gráfico, mostra que na maior parte do tempo de execução do programa a taxa de acessos por ciclo é de 0,035. Numa frequência bem mais baixa ocorre também uma convergência do valor da taxa de acessos em 0,076. A taxa mais baixa representa os acessos às colunas da matriz enquanto a taxa mais alta representa os acessos às linhas da matriz, situação em que é possível obter algum reaproveitamento dos dados nas *caches*. Nos curtos intervalos em que a taxa oscila entre 0,6 e 0,75 ocorre a interferência causada pelo *dump* do *Oprofile Estendido*.

Para o caso da multiplicação com *padding* e blocagem, apresentado na Figura 7.8, a maior parte dos acessos à L1 ocorrem na faixa entre 0,68 e 0,7 acessos por ciclo. Logo abaixo há um outro patamar menos denso entre 0,62 e 0,63. Possivelmente esses patamares também indiquem os acessos da linha do bloco da matriz e da coluna do bloco da matriz durante cada fase da multiplicação. As demais concentrações de pontos abaixo desses valores indicam as interferências entre os blocos que causam as oscilações das faltas na L1 e na L2. Os intervalos com pontos acima de 0,7 são causados também pela interferência do *daemon* do *Oprofile Estendido*.

A redução da taxa de acertos na L2 por acessos à *cache* pode significar duas coisas: (i) que a taxa de *acertos* na L1 aumentou; (ii) que as *faltas* na L2 aumentaram. Por isso, para complementar o estudo do comportamento dos acertos na L2, os gráficos contidos na Figura 7.9 apresentam as faltas por acesso na L1, plotadas em vermelho, e os acertos na L2, plotados em verde. Dessa forma, é possível estabelecer a origem da causa da diminuição nas faltas na L2.

Os acertos por acesso na L2 – plotados em verde – para a multiplicação convencional se mantém na faixa dos 0,07 como mostrado no topo da Figura 7.9. Também é possível observar mais três faixas de valores nas quais ocorrem convergências dos pontos: a primeira em 0,033, a segunda em 0,072 e a terceira em 0,085.

Em razão da densidade das três faixas de valores ser parecida, além de baixa, é difícil estabelecer uma correlação entre elas e as fases da multiplicação das matrizes. Comparando-se a taxa de acertos na L2, que oscila em 0,033, com a taxa de faltas na L1 (em vermelho),

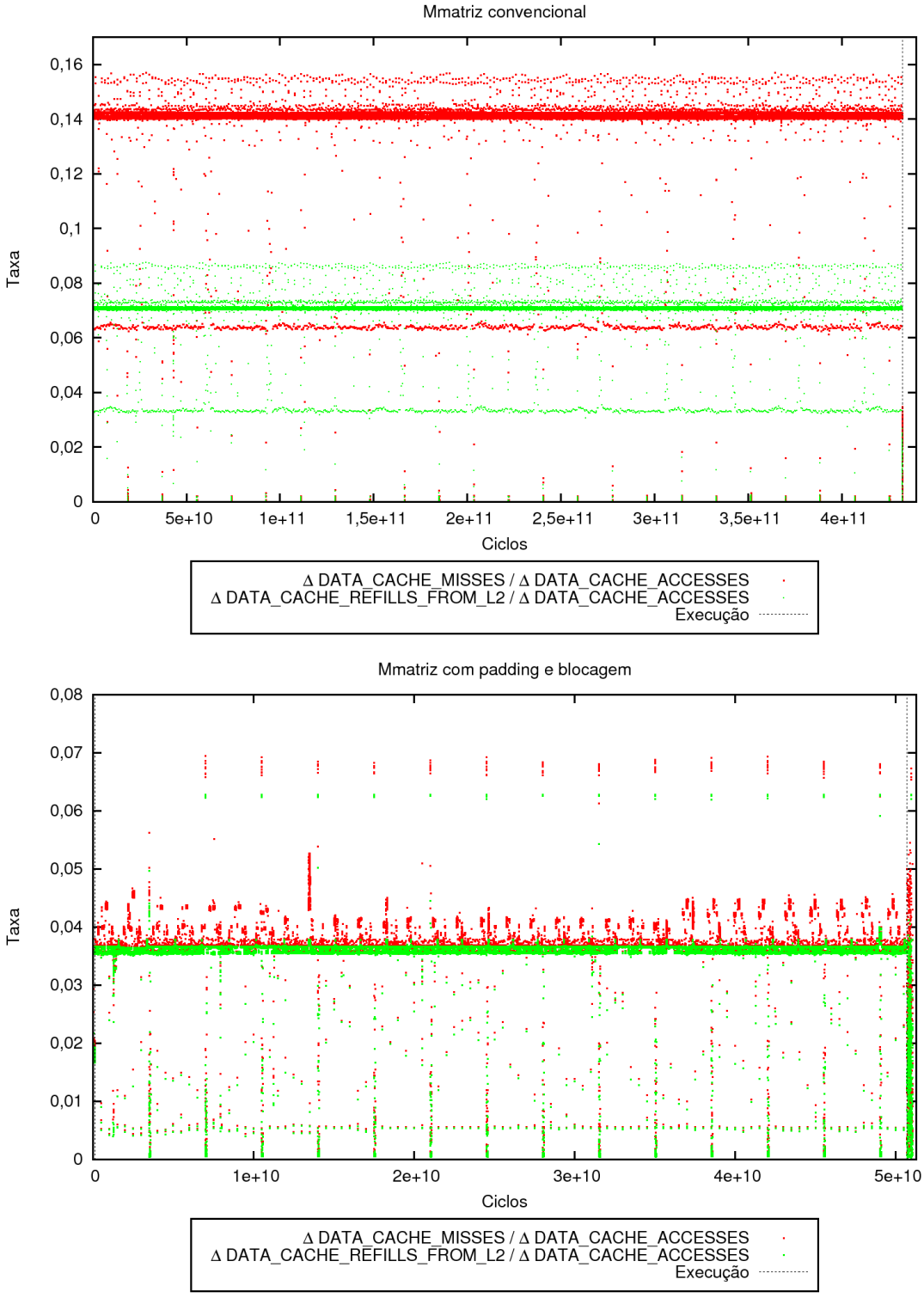

Figura 7.9: Acertos na L2 por acessos à *cache* na multiplicação convencional (acima) e na multiplicação com *padding* e blocagem.

em 0,064, é possível associar a redução dos acertos na L2 com a redução das faltas na L1. Dessa associação, é possível relacionar a faixa mais baixa de valores de recargas da L2 a acessos às linhas da matriz e a faixa superior aos acessos às colunas da matriz.

As faltas na L1 (em vermelho), que oscilam em torno de 0,154, causam a elevação da taxa dos acertos na L2 (em verde) para 0,085. Considerando o tamanho da L2 e o tamanho do conjunto de trabalho utilizado para acessar as colunas da matriz, é muito difícil atribuir os acertos na L2 ao acesso de colunas e, portanto, essas taxas também devem ser atribuídas aos acessos às linhas da matriz.

Os demais acessos e faltas realizados pelo programa, faltas na L1 na faixa de 0,142 com acertos na L2 na faixa de 0,072 representam os acessos às colunas da matriz. A pequena separação entre na faixa plotada em 0,072 pode representar alguma melhoria no reaproveitamento de dados das colunas, mas ainda nada que represente um ganho de desempenho significativo.

O valor da taxa de acertos na L2 oscila entre 0,035 e 0,036 para a multiplicação otimizada, na porção inferior da Figura 7.9. Os picos pouco acima dessa faixa não são causados por interferências, mas pelo próprio programa. Ocorre ainda uma pequena concentração em 0,005, decorrente de um aumento nos acertos da L1, como pode ser verificado pela redução da taxa de faltas.

Ainda sobre a taxa de faltas, é possível perceber que apesar da sua variação, os acertos na L2 permanecem em um nível constante. Os picos na taxa de faltas entre 0,04 e 0,05 causam uma redução quase imperceptível na taxa de acertos da L2. As faltas na L1 causadas pelo *Oprofile Estendido* que elevam a taxa acima de 0,065 causam um aumento na taxa dos acertos na L2 para 0,062 enquanto as que reduzem a taxa de faltas para quase 0 também reduzem os acertos.

A análise da taxa das recargas provenientes do sistema por acesso à *cache* pode ser observada no topo da Figura 7.10 para a multiplicação convencional. O gráfico mostra que, na maior parte do tempo, a taxa de faltas na L2 é de 0,07. Há também um patamar na taxa de acessos em 0,03. A taxa mais alta representa os acessos às colunas da matriz enquanto a taxa mais baixa representa os acessos às linhas da matriz, nos quais é possível

obter algum reaproveitamento dos dados na L2. As falhas periódicas na faixa inferior de pontos representam a interferência do *Oprofile Estendido*, comprovada comparando o número de falhas com a ocorrência dos *dumps* indicada no registro do *Oprofile Estendido*.

Para a multiplicação com *padding* e blocagem, na parte inferior da Figura 7.10, a maior parte das faltas na L2 ocorrem numa faixa muito próxima de 0. Pouco acima dessa primeira faixa há outro patamar discreto. Estes patamares indicam os acessos da linha do bloco da matriz e da coluna do bloco da matriz durante cada fase da multiplicação. As concentrações de pontos acima desses valores indicam as interferências entre os blocos que causam as faltas na L2.

Os dois gráficos da Figura 7.10 apresentam um comportamento inverso aos gráficos correspondentes, apresentados na Figura 7.8. As faltas na L2 interferem diretamente na taxa dos acessos à cache, uma vez que a penalidade de uma falta nesse nível da hierarquia de memória pode bloquear os acessos subsequentes até que o dado que ocasionou a falta seja recuperado da memória principal.

Na porção inferior da Figura 7.10 é possível identificar três fases distintas na concentração de pontos entre 0,005 e 0,01. Estas mesmas fases podem ser observadas na porção inferior da Figura 7.5, em que a discreta variação na inclinação das linhas que medem os eventos `Data_Cache_Refills_from_L1` e `Data_Cache_Refills_from_System` representa a oscilação da taxa de faltas ao longo das três fases. Esta variação pode ocorrer devido a um padrão de sobreposição dos blocos das matrizes na hierarquia de memória do sistema, ou por um padrão de interferência causada pelo SO ao longo da execução do programa.

Um estudo dos padrões de acesso aos endereços em memória, similar ao apresentado na Figura 7.4 (página 44), pode revelar o que ocorre na memória durante as diferentes fases no comportamento das faltas. Entretanto, tal estudo em uma matriz de 1.024×1.024 elementos resulta em $1.024^3 \times 3$ endereços para plotar no gráfico – aproximadamente 24G*bytes* – o que inviabiliza a geração do gráfico.

Os dados e figuras apresentados nesta seção complementam os dados apresentados nas seções anteriores e também foram obtidos a partir do registro do *Oprofile Estendido*. Não conhecemos nenhuma outra ferramenta que disponibilize dados suficientes para plotar

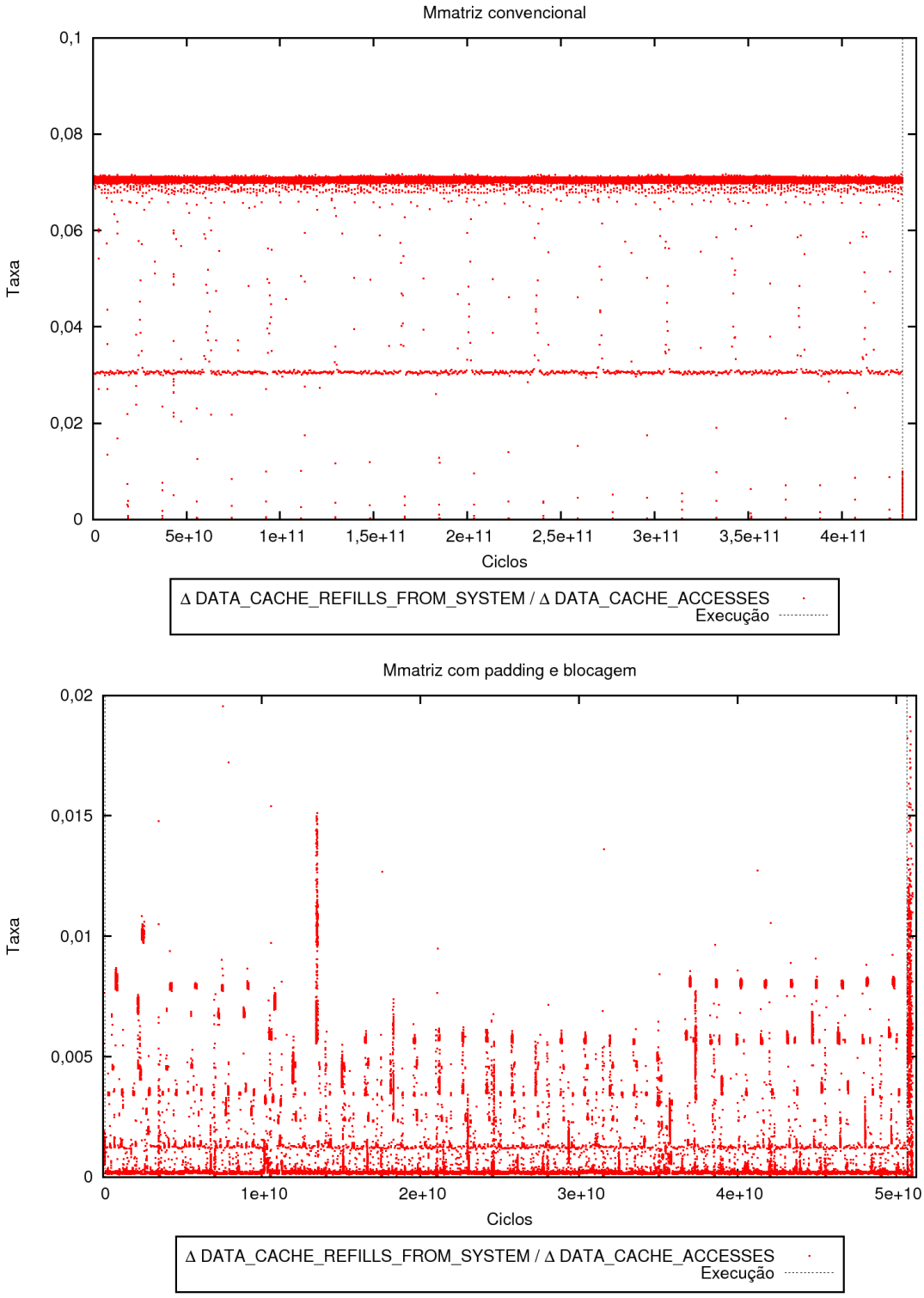

Figura 7.10: Faltas na L2 por acessos à *cache* na multiplicação convencional e na blocada.

gráficos de *eventosXciclo* ou *taxaXciclo*. A distribuição temporal das taxas dos eventos monitorados auxiliam na compreensão do comportamento da hierarquia de memória dos dois programas de multiplicação de matrizes, além de fornecer informações das taxas de faltas para cada etapa da execução dos programas, ao invés de apenas um histograma ou de uma média das taxas de faltas das execuções dos programas.

CAPÍTULO 8

ANÁLISE DO MERGESORT

Este capítulo descreve um conjunto de experimentos para determinar o comportamento da hierarquia de memória durante a execução de programas que implementam duas versões do algoritmo de ordenação *Mergesort*. A primeira é uma implementação simples do algoritmo Mergesort e a segunda é uma versão otimizada para tirar proveito da hierarquia de memória, chamada de *Tiled Mergesort*.

8.1 Descrição dos Programas

A primeira versão do *Mergesort* ordena um vetor dividindo-o ao meio e fazendo uma chamada recursiva da própria função de ordenação para uma metade do vetor. Essas chamadas se repetem até que o vetor tenha apenas um elemento. Neste ponto, a recursão volta um estágio e é realizada uma chamada para a outra metade do vetor. Na volta da segunda recursão é chamada uma função que agrega as duas metades do vetor de forma ordenada, sobre o vetor original, com o auxílio de um segundo vetor denominado *vetor de resultado*.

O segundo algoritmo, *Tiled Mergesort*, é uma versão do *Mergesort* convencional em que as chamadas recursivas sobre as metades do vetor param quando o vetor atinge o tamanho da *cache* da máquina. Neste ponto (da recursão) é utilizado outro algoritmo de ordenação, para que se tire proveito da localidade espacial dos dados em *cache*. O outro algoritmo utilizado nesta implementação do *Tiled Mergesort* é o *Quicksort*.

O *Quicksort* é um algoritmo que ordena um vetor selecionando um elemento, denominado pivô, e ordenando os demais componentes do vetor de forma que os elementos menores do que o pivô sejam posicionados antes dele e os maiores sejam posicionados após ele. O processo se repete recursivamente para os subvetores à esquerda do pivô, até que o subvetor possua apenas um elemento. Neste ponto, a recursão retorna um nível e

a metade direita do subvetor é ordenada através do mesmo processo. Este procedimento se repete até o retorno de todos os níveis da recursão.

8.2 Preparação do Experimento

Esta seção descreve os valores utilizados nos programas para cada teste, além dos parâmetros de configuração utilizados no *Oprofile Estendido*.

8.2.1 Parâmetros dos Programas

Os dois programas estudados ordenam o mesmo vetor, de 819.200 elementos do tipo *double*. Este vetor ocupa um espaço de 6 Mbytes na memória e portanto, é maior do que a capacidade da L1 (1.000 ×), da L2 (25 ×), da L1dTLB (50 ×) e da L2dTLB (6 ×). Este tamanho de vetor foi utilizado para que a cada execução da medida, os dados presentes nas *caches* e TLBs do sistema não fossem reaproveitados. Os resultados apresentados nos gráficos e tabelas dessa seção correspondem à média de 20 execuções de cada teste.

O vetor utilizado nos programas foi gerado com a função `random()` e salvo em um arquivo '.c'. Este vetor foi compilado à parte e o seu arquivo objeto foi ligado aos programas *Mergesort* e *Tiled Mergesort*. Não foi utilizado um arquivo separado contendo o vetor porque a leitura desses dados causaria interferências indesejadas dos acessos a disco.

Cada programa de ordenação estudado foi monitorado durante a execução com o mesmo tamanho da faixa de trabalho na qual o *Quicksort* era utilizado. A faixa utilizada tem o tamanho da *cache* L2 do processador. A função `readtsc()` foi utilizada nesse teste para demarcar o início e o final da execução de cada programa, assim como o ponto da ordenação em que o *Tiled Mergesort* inicia a execução da função `quicksort()`. O ponto equivalente no *Mergesort* convencional também é demarcado pela função `readtsc()`, além das chamadas à função `merge()`, que agrega dois vetores recém ordenados.

É importante frisar que, devido à forma com que os vetores são manipulados pelo *Mergesort*, o trecho delimitado pelas linhas verticais que destaca o momento em que a função `quicksort()` é executada, demarca duas execuções da função – `quicksort()`

Programa	Número de ciclos
Mergesort	$1{,}252 \cdot 10^9$
Tiled Mergesort	$1{,}111 \cdot 10^9$
Diferença Percentual	12%

Tabela 8.1: Contagem de ciclos para os programas de ordenação.

ou `mergesort()` – uma para cada vetor do tamanho da *cache* L2. Esta abordagem, de delimitar duas execuções de cada função, foi utilizada por ser mais simples de se implementar de forma correta do que a abordagem que delimita apenas uma execução das funções de ordenação.

8.2.2 Configurações do OProfile Estendido

Os eventos monitorados com o *Oprofile Estendido* durante a execução dos programas de ordenação *Mergesort* e *Tiled Mergesort* foram: (i) `Data_Cache_Accesses`; (ii) `Data_Cache_Misses`; e (iii) `Data_Cache_Refills_From_System`, ou faltas na L2.

Em princípio, o contador de ciclos (`Cpu_Clock_Unhalted`) seria ajustado de acordo com o tempo de execução de cada um dos testes, de forma que a quantidade de informação registrada pelo *Oprofile Estendido* fosse equivalente nos dois experimentos. Entretanto, uma vez que a diferença no tempo de execução dos programas é da ordem de 12%, como mostrado na Tabela 8.1, optou-se por utilizar a mesma configuração para todos os testes de ordenação.

O contador de ciclos de relógio `Cpu_Clock_Unhalted` foi configurado para interromper o processador a cada 75.000 ciclos e os demais contadores foram configurados para entrar em *overflow* a cada $2^{31} - 1$ eventos, o que equivale a *não* entrarem em *overflow*.

A diferença entre o valor de leitura da função `readtsc()` e os valores dos contadores registrados no ciclo de relógio mais próximo – do início ou do término da execução – do trecho do programa que se quer monitorar não apresenta problemas para a visualização dos dados nos gráficos. Dependendo da frequência da amostragem dos contadores, pode-se perder precisão na medida da contagem total dos eventos. Quanto mais baixa a amostragem maior será o tempo – e a quantidade de eventos ocorridos mas não registrados –

decorrido entre a última amostra coletada e a chamada da função `readtsc()`. Quanto mais curto o tempo de execução do experimento, maior a sensibilidade dos dados à baixa taxa de amostragem.

Para contornar este problema e obter as contagens dos eventos monitorados no ponto em que ocorreu a chamada à função `readtsc()`, é feita uma aproximação calculando-se a variação da contagem dos eventos monitorados em relação à variação no número de ciclos registrados. A taxa é então multiplicada pela diferença entre a leitura do TSC obtida da amostra do *Oprofile Estendido* imediatamente anterior à chamada da função `readtsc()` e o valor retornado pela função. Dessa forma, melhora-se precisão da medida, mesmo utilizando taxas de amostragem mais baixas.

8.3 Resultados das Medidas

Esta seção apresenta os resultados das medidas dos experimentos para os programas enquanto ordenam os vetores do tamanho da *cache* L2.

8.3.1 Comportamento dos Programas e Contagem dos Eventos

A Figura 8.1 mostra os gráficos da execução das duas versões do Mergesort. A linha *Merge* indica as chamadas da função `merge()`, que agrega os dois vetores. A linha *Quick* indica o ponto no qual o vetor utilizado naquele nível da recursão tem o tamanho igual ao dobro da capacidade da cache L2.

No Tiled Mergesort, quando o vetor atinge o dobro do tamanho da L2, o programa executa o *Quicksort* para as duas metades do vetor, cada uma delas com o tamanho da *cache* L2. No Mergesort a linha de nome *Quick* serve apenas para comparar o comportamento dos dois programas durante a ordenação dos vetores num mesmo estágio da execução.

As demais linhas do gráfico indicam a contagem dos eventos monitorados pelo *Oprofile Estendido*. `Data_Cache_Accesses` mostra a contagem dos acessos à *cache* na sua escala normal. `Data_Cache_Misses` mostra as faltas na L1, que estão numa escala equivalente

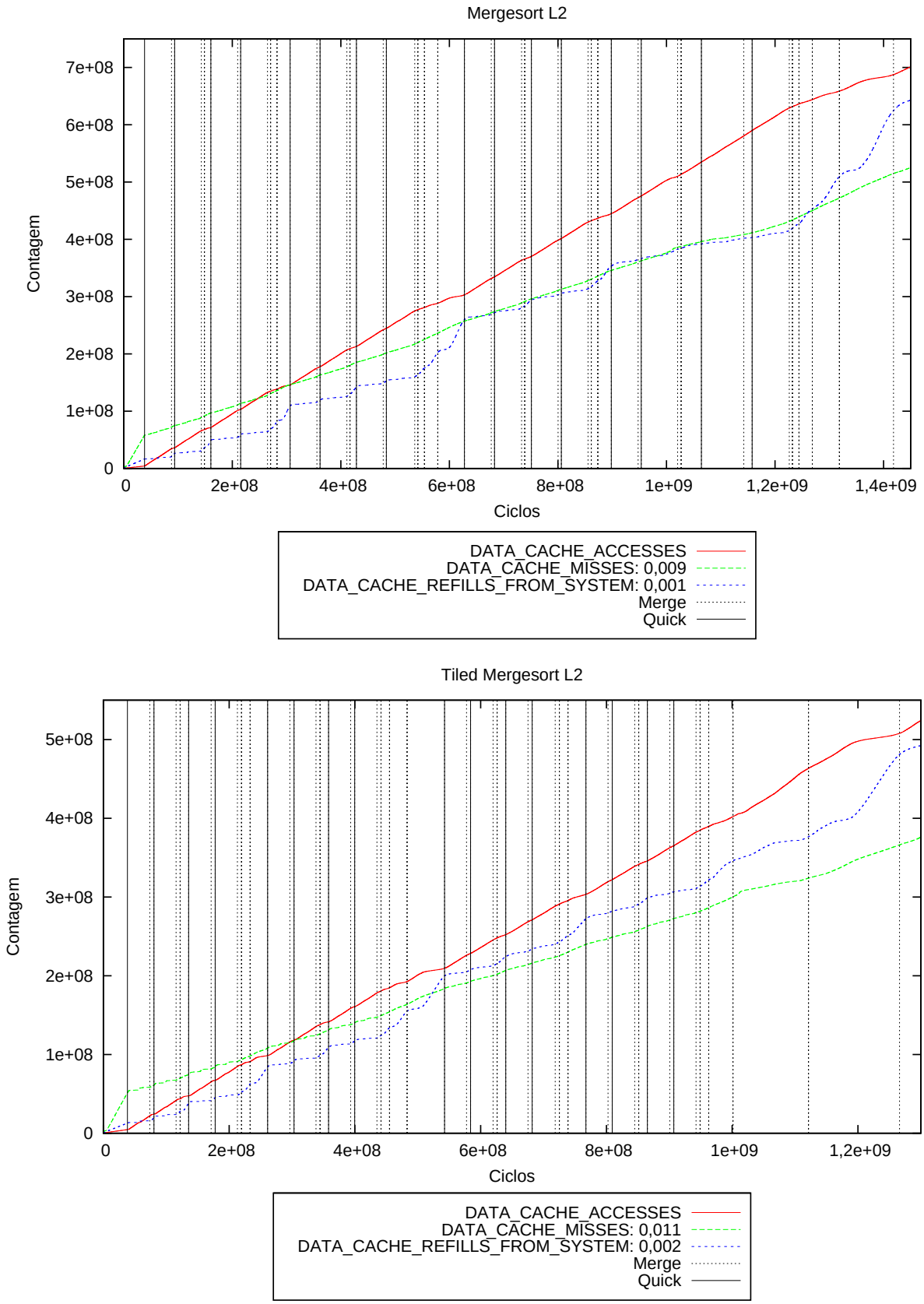

Figura 8.1: Acima, execução do *Mergesort*. Abaixo *Tiled Mergesort* no mesmo vetor.

a 0,009 dos valores obtidos do registro do *Oprofile Estendido* para o *Mergesort* e 0,011 para o *Tiled Mergesort*. Finalmente, a terceira linha mostra a contagem das faltas na L2, representadas pelo evento `Data_Cache_Refills_From_System`; Os valores reais destes eventos estão em 0,001 do valor do gráfico para o *Mergesort* e em 0,002 para o *Tiled Mergesort*.

Uma ampliação de trechos equivalentes da execução dos dois programas que ilustra a sequência das chamadas de função pode ser observada nos gráficos da Figura 8.2. A linha vertical sólida indica o ponto em que a faixa de trabalho do vetor está com o dobro da capacidade da *cache* L2. Na área demarcada entre as linhas verticais indicada por A_1, o *Mergesort* (topo) continua a sua execução normal enquanto o *Tiled Mergesort* executa o *Quicksort* para os dois vetores. A área indicada por B_1 demarca a execução da função `merge()`, juntando os dois vetores recém ordenados. Em seguida, a àrea indicada por A_2 delimita outro par de vetores do tamanho da *cache* sendo ordenado. Após essa ordenação, a área B_2 indica os dois vetores recém ordenados sendo agrupados com o auxílio do vetor de resultado.

Na chamada seguinte à função `merge()` (C_1) os dois vetores (ordenados em A_1 e A_2) são agregados nas duas últimas chamadas da `merge()` com o auxílio do vetor de resultados. As áreas seguintes do gráfico – A_3, B_3, A_4, B_4 e C_2 – indicam este mesmo processo se repetindo e, em D_1 ocorre o agrupamento de todos os vetores ordenados durante o trecho da execução dos programas exibido nos gráficos da Figura 8.2.

Durante a execução da função `merge()`, a taxa de faltas na L2 é elevada, uma vez que o tamanho do conjunto de dados utilizado é, no mínimo, quatro vezes maior do que a capacidade da L2: cada vetor recém ordenado tem o tamanho da L2 e o vetor de resultados trabalha copiando todos esses dados, dobrando a área de memória acessada durante a ordenação. Enquanto a taxa de faltas na L2 aumenta, as faltas na L1 se mantém constantes e os acessos à cache diminuem porque a busca dos blocos faltantes produz bloqueios no processador.

A análise mais detalhada deste trecho da execução dos programas – utilizando dados do registro do *Oprofile Estendido* – mostra que a quantidade de acessos à L1 *cache* do

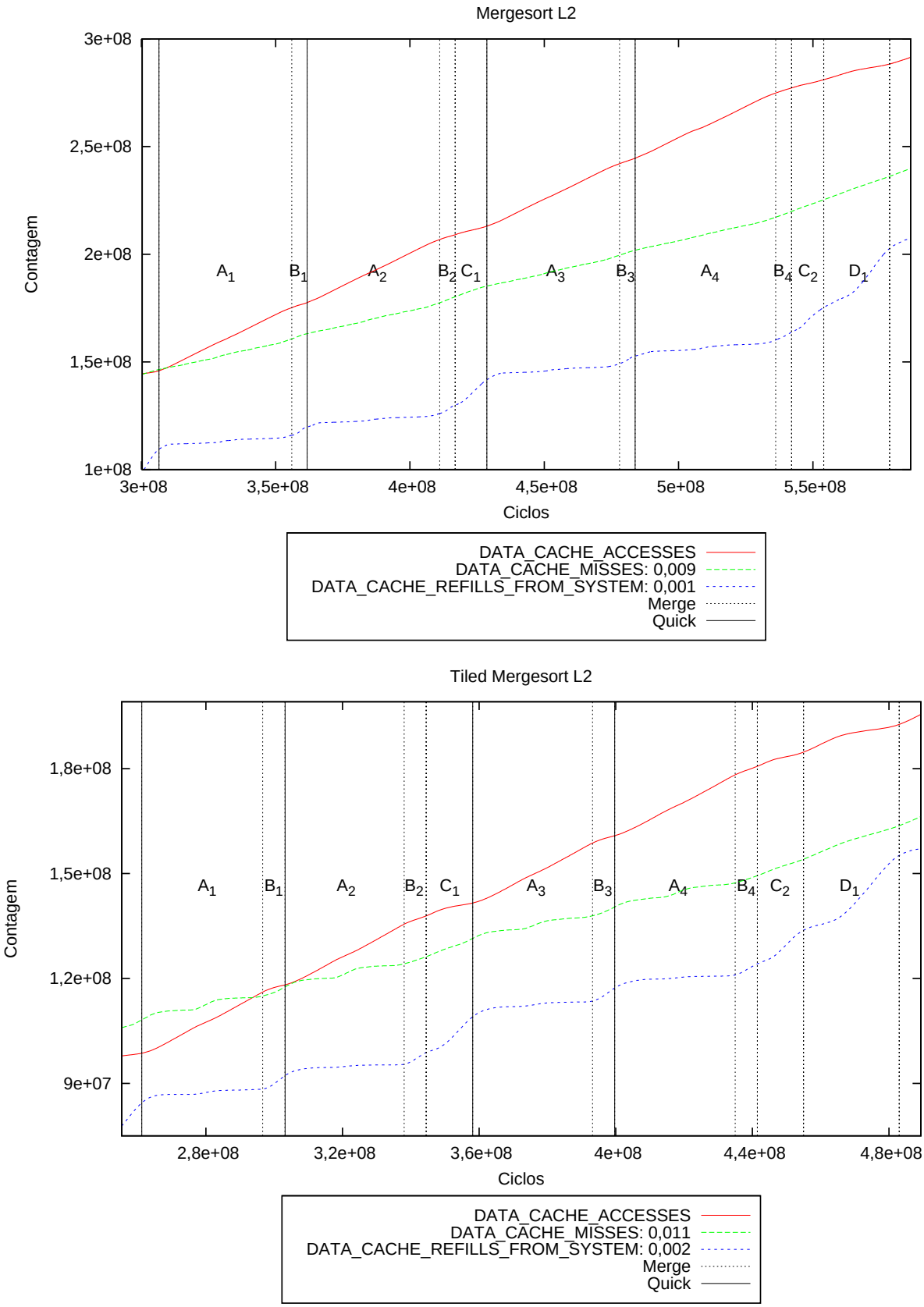

Figura 8.2: Ciclos da ordenação; topo *Mergesort* e acima *Tiled Mergesort*.

	Programa		
Evento	Mergesort L2	Tiled Mergesort L2	Diferença (%)
Ciclos	$2{,}72{\cdot}10^8$	$2{,}21{\cdot}10^8$	22
Cache Access	$1{,}53{\cdot}10^8$	$9{,}4{\cdot}10^8$	63
Cache Misses	854.347	601.180	42
Refills from System	168.521	165.887	1,5

Tabela 8.2: Contagem de eventos para o trecho da execução dos programas.

Mergesort é 63% superior à do *Tiled Mergesort*. Quanto aos demais eventos medidos, a proporção é de 42% a mais para as faltas na L1, 1,5% a mais para as faltas na L2 *cache* e 22% a mais na quantidade de ciclos. Os valores utilizados de base para esses cálculos estão dispostos na Tabela 8.2.

Para comparar o comportamento do *Mergesort* e do *Quicksort* durante a ordenação da mesma faixa do vetor, um trecho menor da execução dos dois programas foi ampliado. Para estes trechos apresentados, a média das 20 execuções descaracterizou o comportamento dos programas e por isso, os gráficos apresentados até o final desta seção também mostram os resultados da execução "mais parecida" com média das 20 execuções. Os valores apresentados no entanto, são das médias de 20 testes.

O comportamento dos acessos à *cache* é mostrado na Figura 8.3. O gráfico do *Mergesort* (topo) mostra o momento que a função `mergesort()` é chamada para o vetor com o dobro do tamanho da L2. O *Tiled Mergesort*, mostrado na porção inferior da Figura 8.3, indica o momento em que a função `quicksort()` é executada. A contagem dos acessos é de aproximadamente $3,17 \cdot 10^7$ para o primeiro programa e de $1,75 \cdot 10^7$ para o segundo. O total de ciclos utilizados é de $4,92 \cdot 10^7$ para *Mergesort* enquanto para o *Quicksort* é de $3,53 \cdot 10^7$ ciclos.

O comportamento dos acessos do *Mergesort* começa constante e aproximadamente na metade do gráfico, ponto onde ocorre a última chamada à `merge()`, há uma redução gradativa na contagem de acessos até o ponto em que ela se mantém constante, devido ao aumento das faltas e os consequentes *stalls* do processador. Quando o próximo vetor começa a ser ordenado, a contagem de acessos volta a subir na mesma taxa e só baixa no final da área delimitada, quando mais uma vez, a função `merge()` é chamada e as duas

metades do vetor são ordenadas com o auxílio do vetor de resultado.

Os acessos à *cache* executados pelo *Quicksort* apresentam uma redução no número de acessos tanto no início quanto no meio da área delimitada pelas linhas verticais do gráfico. Cada oscilação dessa representa o início da execução do *Quicksort*, quando todos os elementos do vetor são lidos e movimentados de acordo com o seu valor em relação ao pivô. Nessa fase, muitas faltas ocorrem na L2, o que ocasiona bloqueios aos demais acessos e consequentemente a redução na contagem.

O mesmo trecho delimitado pelas linhas verticais, ajustado para mostrar as faltas na L1 para os dois programas é exibido na Figura 8.4. A linha que representa as faltas causadas pelo *Mergesort* (topo) apresenta o comportamento do programa ao longo da ordenação de dois vetores do tamanho da L2. É possível observar nos degraus causados pela função `merge()` os oito estágios da execução da ordenação, marcados com flechas.

O primeiro estágio (A_1) representa o final da ordenação do primeiro quarto do vetor, quando a contagem das faltas aumenta, formando um pequeno degrau na linha. Em B_1, o segundo quarto do vetor começa a ser ordenado até o próximo degrau da linha, mais alto, que representa a ordenação da primeira metade do vetor. O terceiro quarto do vetor começa a ser ordenado e termina no terceiro degrau (C_1). O último quarto do vetor é ordenado e termina no degrau mais alto, indicado por D_1. Neste ponto, as duas metades de todo o vetor são agregadas, o que causa o aumento nas faltas. Da metade do gráfico adiante, o processo se repete para o segundo vetor, como demonstrado em A_2, B_2, C_2 e D_2.

Para o *Quicksort*, no início da ordenação (Figura 8.4, abaixo), ocorrem a maioria das faltas enquanto todos os elementos do vetor são lidos e reposicionados. Depois dessa primeira ordenação, apesar das faltas diminuirem elas continuam a ocorrer, uma vez que o tamanho do vetor é quatro vezes a capacidade da L1. Na ordenação do segundo vetor, o comportamento é praticamente o mesmo. O momento em que as faltas ocorrem depois que a contagem se estabiliza depende do reaproveitamento dos dados em *cache*. De acordo com a movimentação dos valores no vetor em relação ao pivô, o reaproveitamento dos elementos pode ser maior ou menor, o que interfere no momento em que as faltas

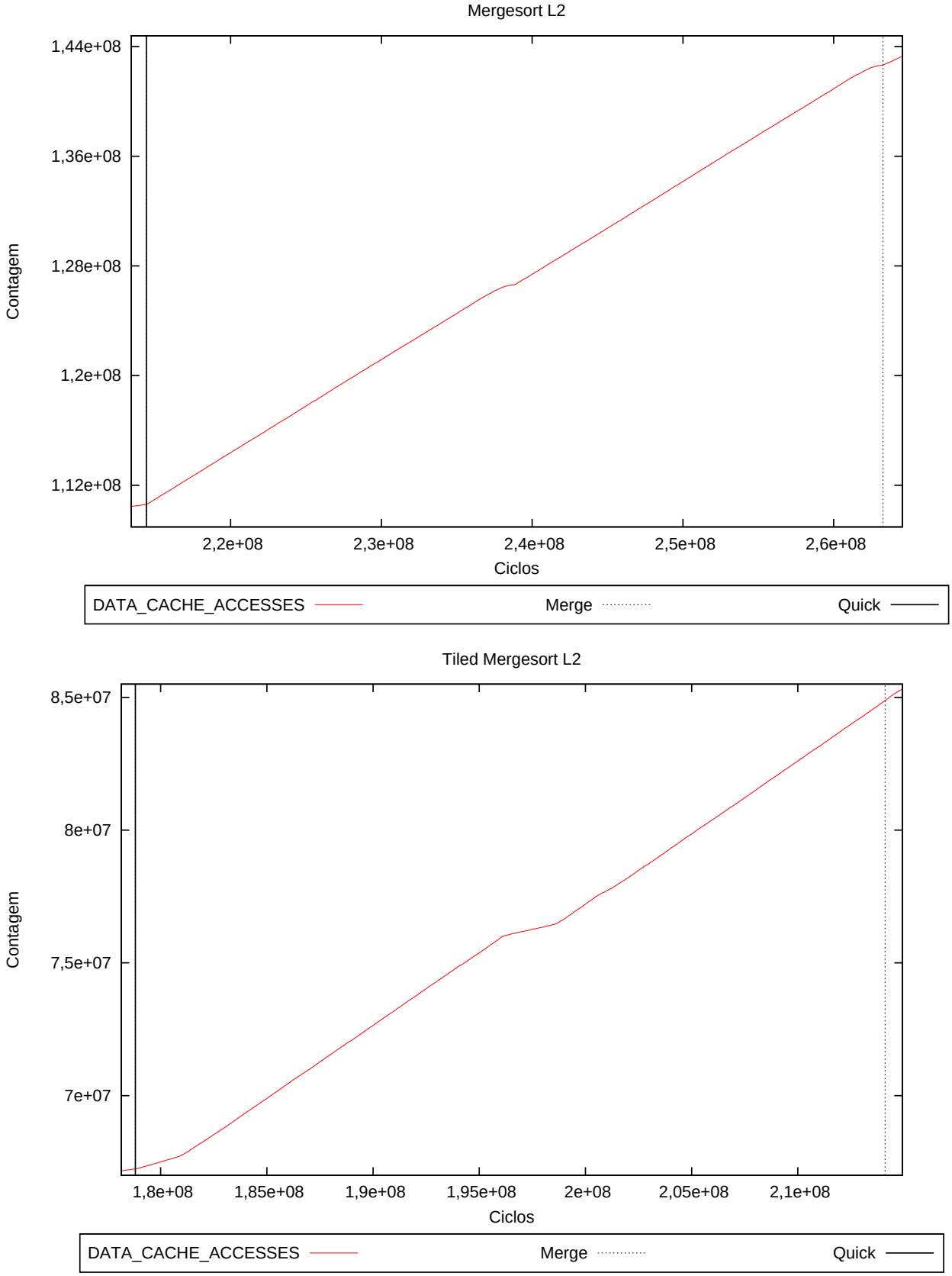

Figura 8.3: Acessos à *cache* do *Mergesort* e *Tiled Mergesort*, faixas do tamanho da L2.

ocorrem, mas não na quantidade de faltas.

A contagem total de faltas para cada programa, na área delimitada pelas linhas verticais, é de aproximadamente 134.371 para o *Mergesort*. O *Quicksort* causa 71.937 faltas na L1 durante a ordenação do mesmo trecho do vetor. A diferença entre as contagens, de 86%, é devida à baixa associatividade da L1 e também ao fato do *Mergesort* utilizar o vetor de resultados para agregar os vetores. Os trechos da ordenação do *Mergesort* utilizam o dobro de espaço necessáro ao *Quicksort* e por isso há uma quantidade maior de faltas e, consequentemente, perda de desempenho.

A contagem do evento `Data_Cache_Refills_from_System`, ou faltas na L2, apresenta um comportamento similar ao das faltas na L1 para os dois programas, como pode ser observado na Figura 8.5. Para o *Mergesort*, à medida que a função `merge()` é executada, as faltas ocorrem e os dados são armazenados em *cache*. As chamadas subsequentes da função `merge()` sobre conjuntos maiores do vetor tiram proveito dos dados presentes em *cache* e não causam faltas na L2 até o momento em que ocorre a última ordenação do vetor, quando a quantidade de faltas é baixa e aumenta bruscamente, uma vez que a L2 não acomoda todo o conjunto de dados.

O comportamento da contagem de faltas na L2 apresenta trechos planos indicados por flechas onde a contagem de faltas na L1 apresenta degraus, à exeção do final de cada ordenação, quando a contagem de faltas na L2 aumenta. Para este programa, o total de faltas na L2 contabilizado na área delimitada pelas linhas verticais é de 9.926.

O *Quicksort* apresenta na L2 o mesmo comportamento em relação às faltas na L1. No início da execução, todo o vetor é percorrido e o número de faltas aumenta rapidamente. Em seguida, o vetor está carregado na *cache* e as movimentações seguintes dos elementos causam faltas de acordo com o posicionamento do pivô. Aparentemente, a quantidade de faltas que ocorre na ordenação do segundo vetor é maior do que a que ocorre no primeiro vetor. Observando atentamente a linha do gráfico, há uma pequena falha na ascenção das faltas que significa a troca de vetores. O primeiro vetor apresentou um aumento na taxa de faltas no final da ordenação, provavelmente devido a alguma interferência do Sistema Operacional. A contagem neste trecho da ordenação foi de 7.773 faltas. A diferença de

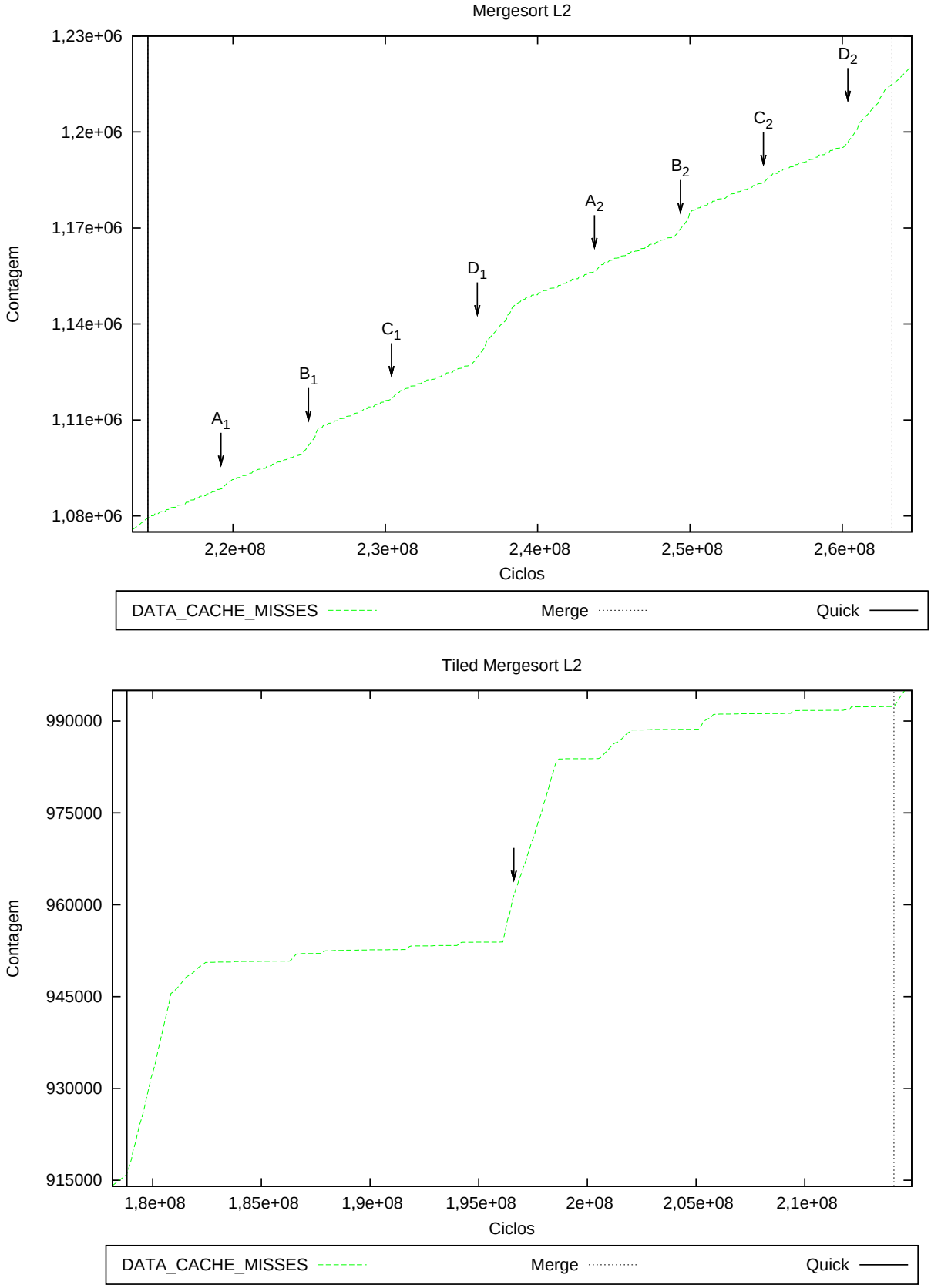

Figura 8.4: Faltas na L1 para o *Mergesort* no topo e para o *Quicksort* na base.

27% entre a quantidade de faltas do *Mergesort* e do *Quicksort* no trecho demarcado é devida à associatividade 4 da L2, que obtém um reaproveitamento de dados superior ao da L1, cuja associatividade é 2 [19].

Com a distribuição temporal dos eventos, é possível observar o comportamento das faltas e acessos à *cache* de forma mais clara do que com as demais ferramentas baseadas em CDHs disponíveis. Assim como na Seção 7.3.2, o *Oprofile Estendido* ajuda na compreensão da forma com que cada programa utiliza a hierarquia de memória, viabilizando a observação e a contagem das faltas e acessos ao longo da execução dos programas. Esta observação permite a identificação de cada etapa da execução dos programas e, consequentemente, um estudo e uma compreensão mais aprofundados da interação entre os programas e a hierarquia de memória do sistema.

8.3.2 Análise das Taxas das Medidas

Da mesma forma que apresentado na Seção 7.3.4, os dados obtidos nas medidas dos experimentos com os programas de ordenação foram utilizados para calcular taxas, dividindo as variações (Δ[1]) de um evento por outro, seguindo as mesmas métricas dos resultados já apresentados: a taxa dos acessos à cache é Δ`Data_Cache_Accesses`/Δ`Ciclos`, a de faltas na L1 é Δ`Data_Cache_Misses`/Δ`Data_Cache_Accesses`, enquanto a taxa das faltas na L2 é Δ`Data_Cache_Refills_from_System`/Δ`Data_Cache_Accesses`.

Em todos os gráficos apresentados nesta seção, o eixo Y apresenta a variação da taxa da medida, o eixo X representa a contagem dos ciclos de relógio ao longo do tempo, os pontos correspondem aos valores da métrica para cada intervalo amostrado e as linhas verticais pontilhadas indicam o início e o fim da ordenação do vetor.

Os dados da taxa dos acessos à *cache* para os dois programas são mostrados na Figura 8.6. O *Mergesort* (topo), apresenta uma variação entre 0,50 e 0,74 acessos por ciclo durante as ordenações de pequenos trechos do vetor. Quando ocorrem chamadas à função `merge()` em porções maiores do vetor – já carregadas em *cache* – a taxa de acessos por ciclo se estabiliza em aproximadamente 0,7. Os pontos em que a taxa apresenta estabili-

[1] $\Delta x = ...x_2 - x_1, x_3 - x_2...$, $\Delta y = ...y_2 - y_1, y_3 - y_2...$, sendo x e y eventos plotados nos eixos x e y.

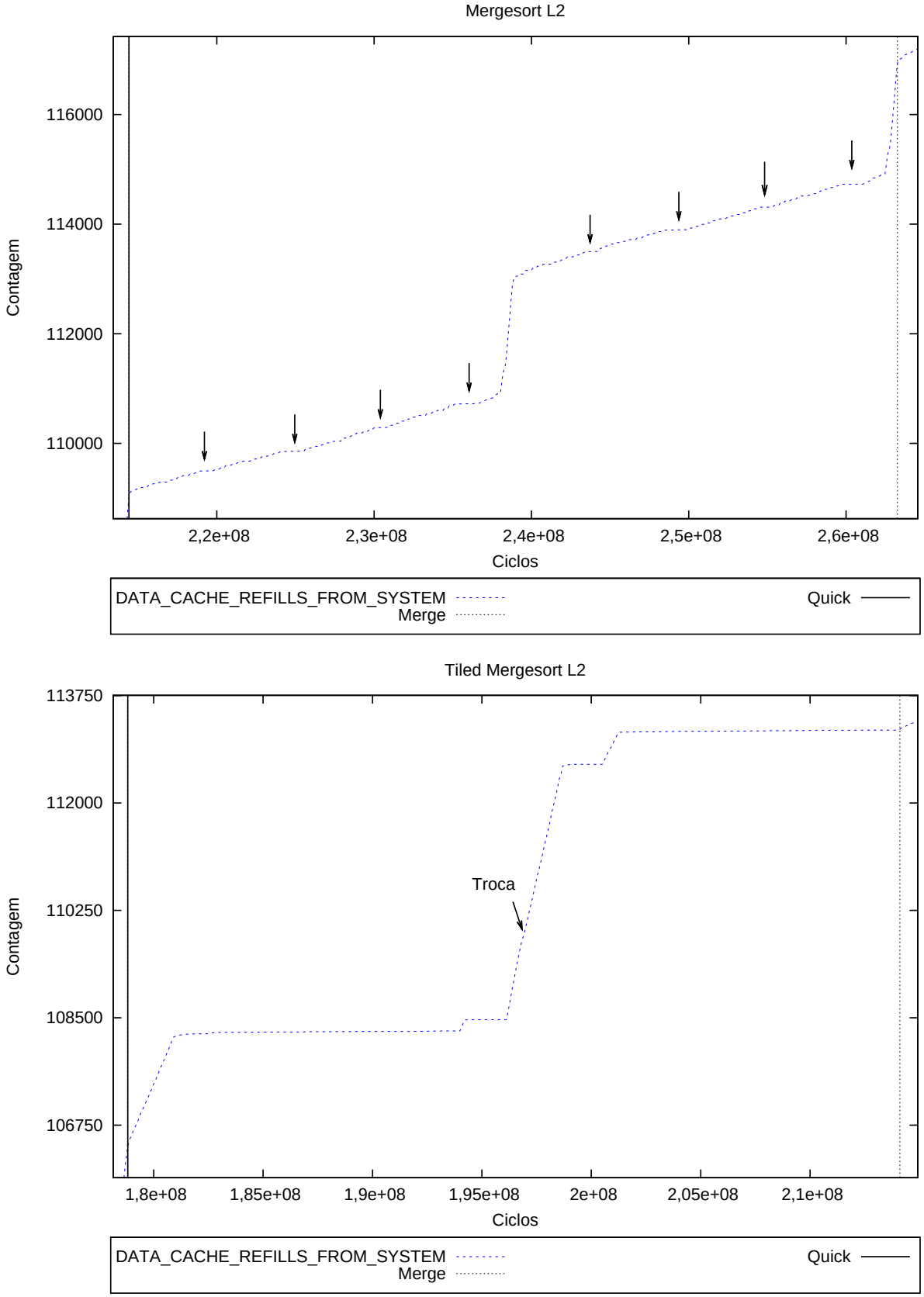

Figura 8.5: Faltas na L2 para o *Mergesort* no topo e para o *Quicksort* na base.

dade estão indicados com flechas verticais e são os mesmos em que o gráfico do *Mergesort* na Figura 8.4 apresenta degraus indicando a ocorrência de faltas na L1, e o da a Figura 8.5 apresenta redução na contagem de faltas na L2.

O motivo da estabilidade da taxa de acessos nestes pontos é a presença dos dados na L2 e a ausência dos mesmos na L1. Nos trechos em que a taxa de acessos é mais instável ocorre que, enquanto os dados são carregados da memória principal durante a ordenação dos trechos menores do vetor, a taxa de acessos diminui quando o dado não está disponivel e aumenta quando eles estão carregados na L1. Como a quantidade de valores ordenados nestes subvetores ainda é muito pequena, há um reaproveitamento desses dados ainda na L1, o que causa o aumento na taxa de acessos.

À medida que a quantidade de dados aumenta, os valores são expurgados da L1 mas permanecem ainda na L2. Quando estes dados são acessados novamente, a taxa de acessos por ciclo passa a ser limitada pelo tempo de acesso à L2 e por isso se mantém constante nos trechos demarcados. No centro do gráfico ocorre a última ordenação do vetor, que causa faltas na L2 além da L1, e causa uma redução na taxa de acessos.

No caso do *Quicksort*, percebe-se que a taxa de acessos diminui durante a primeira fase da execução do programa devido às faltas que estes acessos causam na L1 e na L2. Logo no início, a taxa de acessos cai, oscilando entre 0,27 e 0,17 acessos por ciclo, convergindo para 0,23. A seguir, ela se estabiliza em 0,55 acessos por ciclo. Em alguns pontos, a taxa apresenta uma ligeira baixa, atingindo valores próximos a 0,4. Como os indícios de faltas na L2, que seriam as possíveis causadoras de tamanha redução na taxa de acessos, são muito fracos, efetuou-se uma análise destes trechos diretamente no registro do *Oprofile Estendido*.

Em três dos pontos de baixa, foram encontradas amostras de execução de código do Sistema Operacional. É possível, portanto, que os demais pontos de baixa da taxa de acessos também representem a execução de código do SO, não detectada devido à baixa taxa de amostragem – para este tipo específico de monitoramento.

No registro foram encontradas mais amostras de execução de código de SO na região entre $1{,}96{\cdot}10^8$ e $1{,}98{\cdot}10^8$, entre a troca dos vetores indicada pela flecha vertical, o que

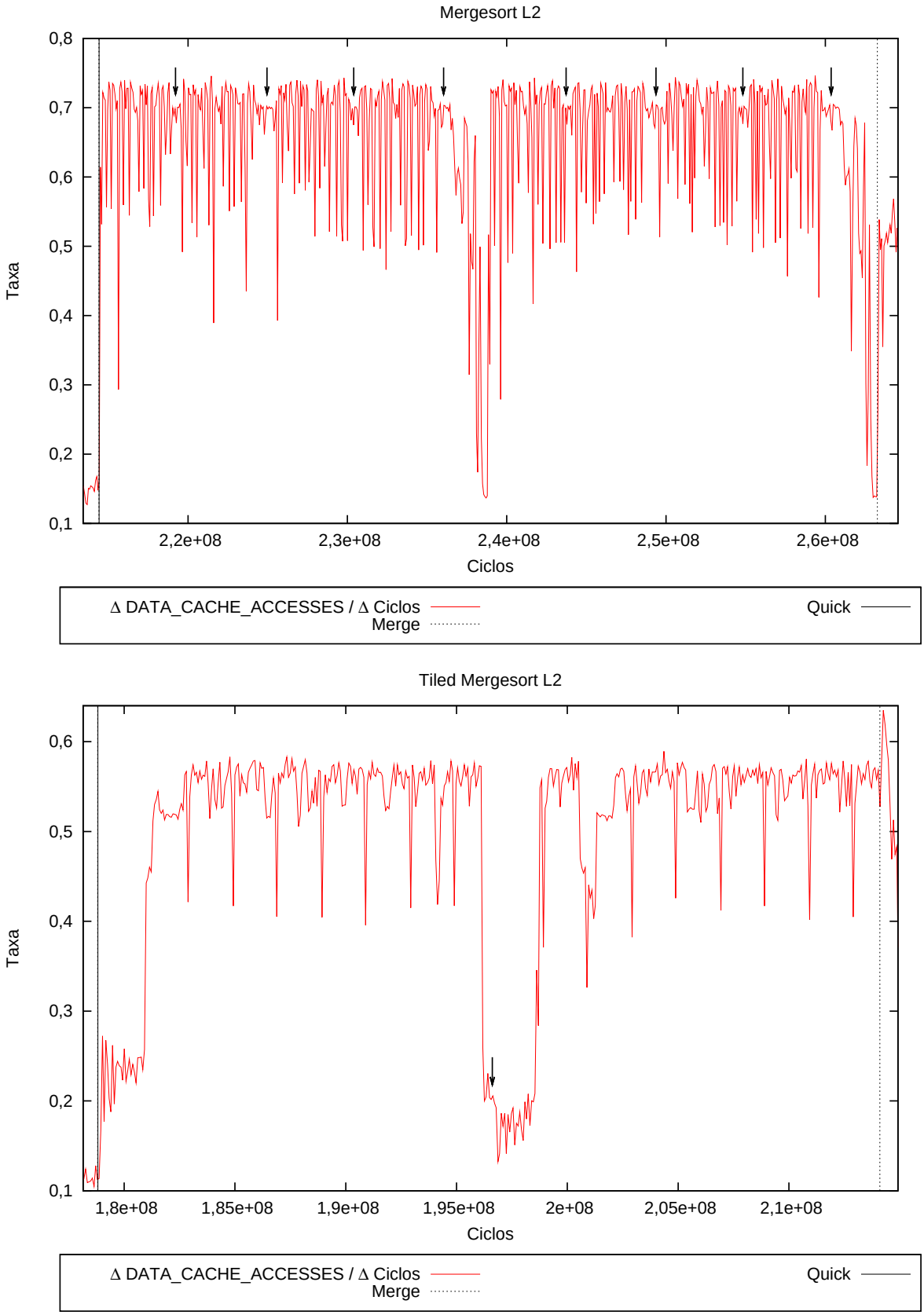

Figura 8.6: Taxas de acessos por ciclos para *Mergesort* (topo) e *Quicksort*.

confirma a hipótese de que a alta contagem de faltas no trecho próximo ao início da ordenação do segundo vetor com o *Quicksort* tenha sido causada por interferência do Sistema Operacional.

Em relação às faltas na L1, mostradas na Figura 8.7 o *Mergesort* apresenta uma oscilação entre $4 \cdot 10^{-4}$ e 0,008 com alguns picos em 0,025. Essa variação da taxa de faltas está relacionada ao comportamento do sistema de memória virtual: os dados são acessados pela primeira vez, causam faltas e aumentam a taxa. Depois de carregados na *cache* L1, enquanto os vetores ordenados são pequenos, ocorre o reaproveitamento dos dados, o que reduz a taxa de faltas.

Os trechos indicados com as flechas na Figura 8.7 são os mesmos indicados nos gráficos anteriores. Devido ao comportamento das faltas na L1 e na L2 observado nos gráficos das Figuras 8.4 e 8.5, a taxa de faltas por acesso apresenta uma pequena estabilidade nestes pontos, devido à execução da função `merge()`. O trecho no centro do gráfico em que a taxa de faltas aumenta temporariamente para 0,025 indica o momento em que ocorre uma chamada para a função `merge()` com uma quantidade de dados superior à capacidade da L1. No início da união dos vetores os dados acessados ainda estão em *cache* e a taxa se mantém em 0.008. A partir do ponto em que os dados ordenados não estão mais presentes em *cache*, a taxa de faltas sobe para 0,025.

No caso do *Quicksort*, o comportamento da taxa de faltas é o esperado. Na fase inicial da ordenação ocorre a maioria das faltas e a taxa oscila em aproximadamente 0,065 faltas por acesso, caindo para aproximadamente $1{,}4 \cdot 10^{-4}$, mantendo-se neste patamar até ocorrer a interferência do SO, pouco antes do término da ordenação do vetor. As pequenas oscilações que ocorrem depois da fase inicial da ordenação são decorrentes da pequena capacidade da L1 em relação ao tamanho do conjunto de dados. Cada oscilação mostrada no gráfico corresponde a um pequeno degrau do gráfico da Figura 8.4.

Analisando a relação entre a taxa de acessos e faltas na *cache* para os dois programas, apesar das faltas do *Mergesort* ocorrerem em maior quantidade, a taxa de acessos à *cache* deste programa é maior do que a do *Quicksort*. Isso ocorre porque, enquanto o *Quicksort* realiza todas as operações de ordenação sobre o mesmo vetor, o *Mergesort* acessa os valores

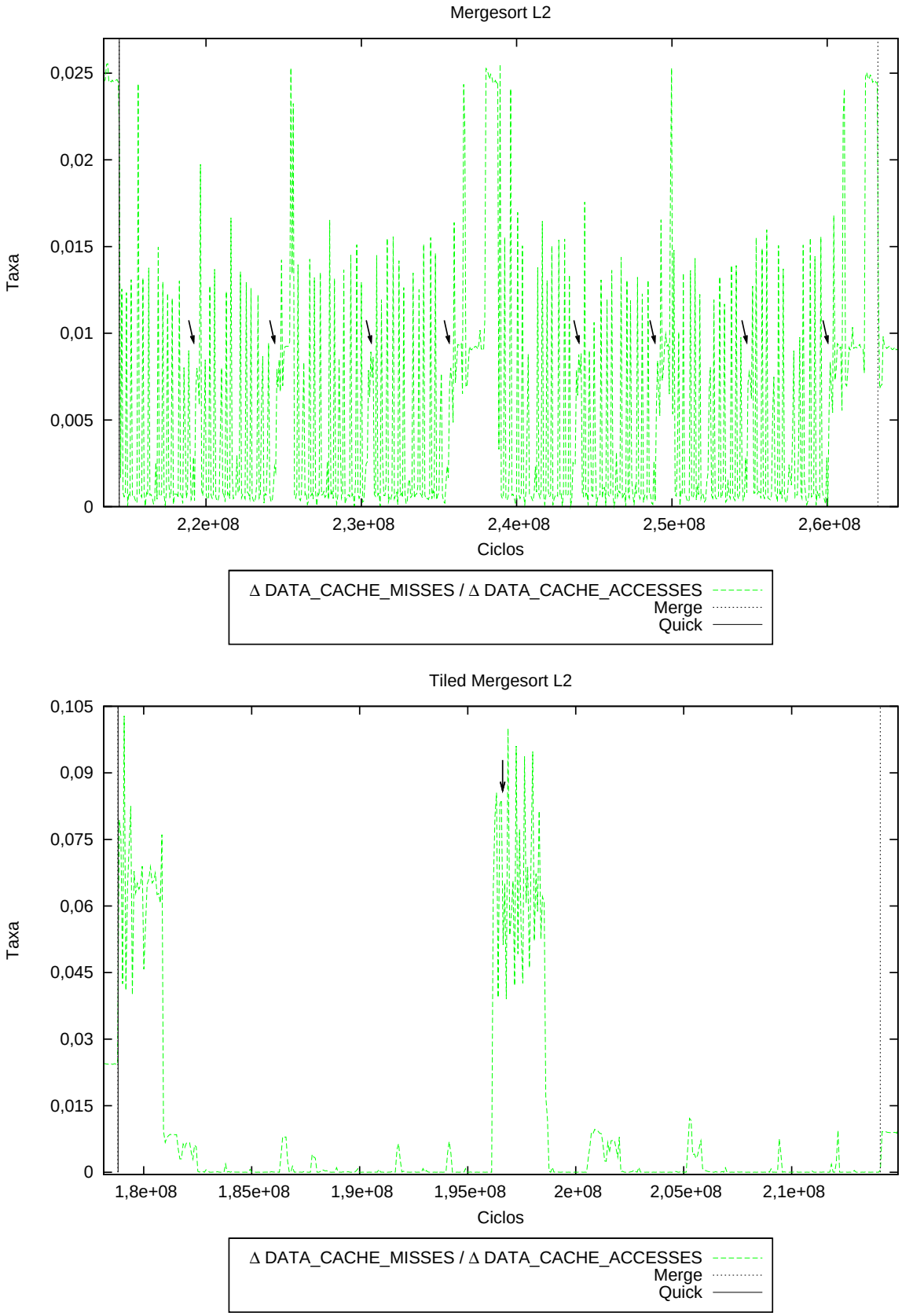

Figura 8.7: Taxas de faltas por acessos na L1 para o *Mergesort* (topo) e *Quicksort*.

no vetor original e agrega os valores ordenados no vetor de resultado a cada execução da função `merge()`.

A taxa de faltas na L2 por acessos à *cache* pode ser vista no topo da Figura 8.8 para o *Mergesort*. Os trechos indicados com as flechas mostram a taxa de faltas próximas a zero durante as execuções da função `merge()` com dados já presentes na L2. Os trechos que apresentam oscilações entre 0 e $8{\cdot}10^{-4}$ indicam as faltas que ocorrem na L2 e que interferem na taxa de acessos à cache, já mostrada na Figura 8.7.

A taxa de faltas na L2 aumenta para 0,025 no *Mergesort* apenas no estágio final da ordenação, quando o tamanho da área de dados acessada – vetor de entrada mais vetor de resultado, cada um ocupando o tamanho da L2 – supera a capacidade deste nível da hierarquia de memória virtual, ao final da ordenação de cada vetor.

No caso do *Quicksort* (Figura 8.8), no início da ordenação do primeiro vetor, a taxa de faltas na L2 começa oscilando entre 0,003 e 0,0058 e estabilizando em aproximadamente 0,0036. Depois que os elementos do vetor são carregados em *cache*, a taxa de faltas reduz para $6{,}5{\cdot}10^{-5}$ e permanece nessa faixa até que ocorre a interferência do SO e o início da ordenação do segundo vetor, indicado pela flecha. Neste ponto, a taxa de faltas oscila entre 0,007 e 0,009 e logo em seguida cai novamente para $6{,}5{\cdot}10^{-5}$.

A distribuição temporal das taxas dos eventos permitiu observar o comportamento das taxas de faltas e acessos à *cache* de forma mais clara do que com outras ferramentas baseadas em CDHs disponíveis. Assim como na Seção 7.3.2, os dados obtidos com o *Oprofile Estendido* mostram a forma com que cada programa utiliza a hierarquia de memória ao longo da execução, permitindo computar valores para a taxa de faltas ou acessos em cada trecho. Esta informação permitiu a identificação das etapas da execução dos programas de ordenação e, consequentemente, uma compreensão maior sobre a interação entre os programas e a hierarquia de memória do sistema. Este conhecimento adicional é útil para a depuração do desempenho destes programas, sugerindo, por exemplo, novos testes utilizando faixas de dados menores para o *Tiled Mergesort*, com o propósito de melhorar a utilização cache L2.

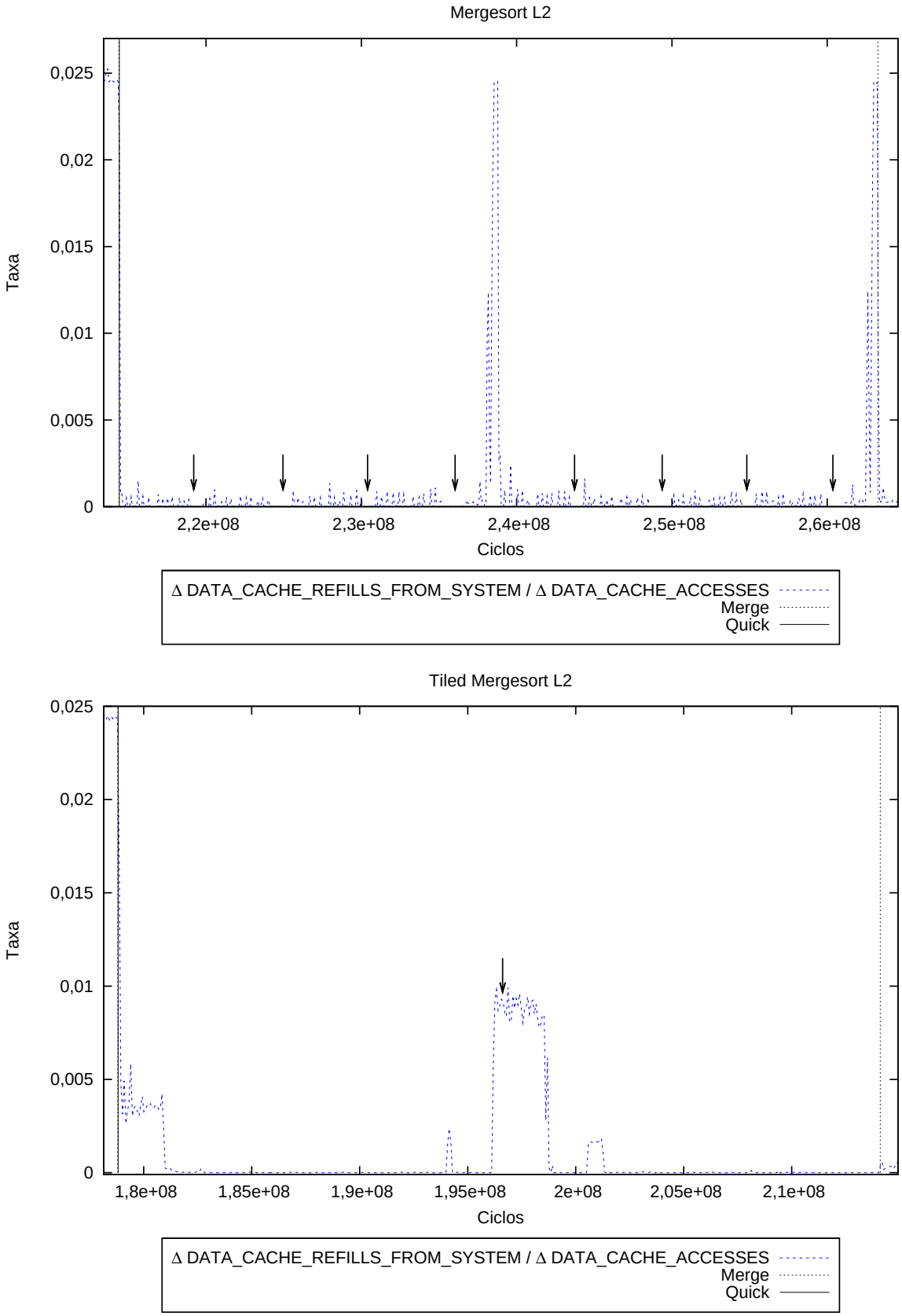

Figura 8.8: Taxas de faltas na L2 por acessos, *Mergesort* (topo) e *Quicksort*.

CAPÍTULO 9

INTERFERÊNCIA NAS MEDIDAS

Qualquer ferramenta de monitoramento interfere na execução do programa que esteja sendo monitorado. Por isso, o resultado de uma execução normal de um programa tende a ser diferente do resultado apresentado pela ferramenta de medida. Quanto maior a interferência da ferramenta, maior a distorção na medida.

No *Oprofile* original, a cada interrupção provocada por um contador de eventos, somente aquela ocorrência é registrada. Normalmente, cada contador provoca várias interrupções ao longo da execução de um programa e tipicamente há mais de um contador gerando interrupções. Assim, o tratamento de uma interrupção é relativamente rápido, mas elas podem ocorrer com alguma frequência.

Com o *Oprofile Estendido*, as interrupções ocorrem a uma taxa que depende da resolução desejada, mas o tratamento de cada interrupção não é rápido porque, além do TSC, todos os contadores são lidos e seus valores registrados. Em relação ao *Oprofile* original, esse aumento na quantidade de dados interfere também no *overhead* causado pela sincronização do sistema de arquivos e pela cópia do arquivo de registros da memória para o disco.

Para mensurar a interferência do *Oprofile Estendido* nas medidas realizadas, em relação à execução normal dos testes, todos os programas apresentados neste trabalho foram monitorados de três maneiras e com diferentes configurações em cada uma: (i) com o *Oprofile Estendido*; (ii) com o *Oprofile* original; e (iii) sem o *Oprofile*. A medida de tempo nos dois últimos é obtida com duas invocações da função `readtsc()`, no início e no final da execução.

Cada experimento também foi monitorado com uma combinação diferente de configurações entre o *Oprofile* e o *Oprofile Estendido* com a finalidade de mostrar as variações da interferência causada pelo *Oprofile Estendido* em cada uma das situações.

Nas tabelas apresentadas neste capítulo, a primeira coluna mostra o programa executado ou algum parâmetro da execução que será citado, a segunda coluna mostra a contagem total dos ciclos da execução do programa monitorado com o *Oprofile Estendido*, a terceira coluna mostra a contagem de ciclos obtidos com o *Oprofile* convencional e a terceira coluna apresenta a contagem obtida na execução normal do programa.

Além dos dados de contagens de ciclos, ao lado de cada medida está mostrada a interferência da ferramenta na quantidade de ciclos medidos. A percentagem de ciclos adicionais apresentada na tabela é contabilizada em relação à execução do programa sem qualquer tipo de monitoramento e é denominada *Overhead*.

9.1 Faltas nas TLBs

Com o intuito de mostrar a variação da interferência das medidas quando se altera o comportamento do programa, a Tabela 9.1 mostra as contagens de ciclos registradas pelo *Time Stamp Counter* (TSC) durante as três execuções do programa que causa faltas na TLB, apresentado no Capítulo 6.

Páginas percorridas	Oprofile Estendido		Oprofile		–
0 a 350	$2,96 \cdot 10^9$	5,6%	$2,83 \cdot 10^9$	0,9%	$2,81 \cdot 10^9$
22 a 42	$2,91 \cdot 10^7$	1,8%	$2,89 \cdot 10^7$	1,5%	$2,85 \cdot 10^7$
250 a 280	$3,71 \cdot 10^8$	1%	$3,70 \cdot 10^8$	0,8%	$3,66 \cdot 10^8$

Tabela 9.1: *Overhead* do OProfile e do OProfile Estendido para faltas nas TLBs.

A configuração utilizada nestes testes foi: (i) `Cpu_Clk_Unhalted`, 150.000 eventos; (ii) `Data_Cache_Accesses`, 500.000 eventos; (iii) `L1_Dtlb_Misses_L2_Dtld_Hits`, 500.000 eventos; e (iv) `L1_And_L2_Dtlb_Misses`, 500.000 eventos.

As variações apresentadas na Tabela 9.1 mostram que para estes experimentos e esta configuração, o *overhead* causado pelo *Oprofile Estendido* é de 0,2 a 0,3% em relação ao *Oprofile* original para os intervalos de páginas apresentados na segunda e terceira linhas da tabela. Na primeira linha, a interferência do *Oprofile* é de 0,9% enquanto a do *Oprofile Estendido* é de 5,6%, ou 5,2% a mais do que o *Oprofile*.

A possível causa da diferença entre as duas medidas é o tempo de execução de cada

experimento. É provável que a execução com o *Oprofile* original tenha passado pelo trecho de interesse da execução antes do *Oprofile* realizar a descarga dos *buffers*, como ocorre nas Figuras 6.3 e 6.5 (páginas 32 e 36). No caso da execução monitorada pelo o *Oprofile Estendido*, a descarga dos buffers ocorre dentro da área de interesse, como pode ser visto na Figura 6.2, página 30. Uma vez que os *dumps* do *Oprofile Estendido* ocorrem em um intervalo fixo de tempo, a descarga ocorre em todas as 20 execuções do programa, afetanto todas as medidas. Por este ser um programa de rápida execução, a interferência da descarga pode alterar significativamente a medida.

9.2 Multiplicação de Matrizes

Os resultados para programas de multiplicação de matrizes apresentados no Capítulo 6 estão na Tabela 9.2, que mostra as contagens de ciclos registradas pelo *Time Stamp Counter* (TSC) durante a execução de cada um dos programas monitorados pelo *Oprofile Estendido*, pelo *Oprofile* e sem nenhum monitoramento, na segunda, terceira e quarta colunas respectivamente. A primeira linha mostra os dados para a multiplicação de matrizes convencional e a segunda, para a multiplicação otimizada com *padding* e blocagem.

Programa	Oprofile Estendido		Oprofile		–
Normal	$4,32 \cdot 10^{11}$	1,8%	$4,27 \cdot 10^{11}$	0,7%	$4,24 \cdot 10^{11}$
Otimizado	$5,07 \cdot 10^{10}$	11,9%	$4,79 \cdot 10^{10}$	5,7%	$4,53 \cdot 10^{10}$

Tabela 9.2: *Overhead* do OProfile e do OProfile Estendido no produto de matrizes.

A configuração utilizada para o *Oprofile* é diferente daquela do *Oprofile Estendido* para estes programas. Neste caso, o intuito das configurações distintas é apresentar mais uma difenção do comportamento da interferência das duas ferramentas durante o monitoramento.

A configuração utilizada nos testes realizados com o *Oprofile* foi: (i) `Cpu_Clk_Unhalted`, 1.500.000 eventos para a multiplicação convencional e 250.000 para a otimizada; (ii) `Data_Cache_Accesses`, 100.000 eventos; (iii) `Data_Cache_Refills_From_L2`, 50.000 eventos; e (iv) `Data_Cache_Refills_From_System`, 50.000 eventos.

Para o *Oprofile Estendido* a configuração utilizada foi: (i) `Cpu_Clk_Unhalted`, 1.500.000 eventos para a multiplicação convencional e 250.000 para a otimizada; (ii) `Data_Cache_Accesses`, $2^{31}-1$ eventos; (iii) `Data_Cache_Refills_From_L2`, $2^{31}-1$ eventos; e (iv) `Data_Cache_Refills_From_System`, $2^{31}-1$ eventos.

As variações apresentadas na Tabela 9.2 mostram que para estes experimentos, nesta configuração, o *overhead* causado pela interferência do *Oprofile* é de 0,7% enquanto o do *Oprofile Estendido* é de 1,8% na multiplicação de matrizes convencional. No caso da multiplicação de matrizes com *padding* e blocagem, a interferência causada pelo *Oprofile Estendido* é de 11,9%, enquanto a interferência do *Oprofile* é de 5,7% em relação à execução da multiplicação sem nenhum tipo de monitoramento.

A diferença nas taxas de amostragem dos ciclos de relógio `Cpu_Clk_Unhalted` é fator determinante para o aumento do *overhead* do programa de multiplicação de matrizes normal para o otimizado. A taxa de amostragem deste evento para o programa otimizado é seis vezes superior à taxa de amostragem para o programa convencional.

No caso do *Oprofile Estendido*, a relação entre a medida do *overhead* para os dois programas é de 6,4 vezes, próximo da relação entre as taxas de amostragem configuradas para cada um dos programas. Para o *Oprofile* original este raciocínio não se aplica, uma vez que qualquer evento pode causar uma interrupção do processador e prejudicar a medição. A diferença de *Overhead* entre as medidas realizadas com o *Oprofile* é de aproximadamente 9 vezes, bastante superior à diferença da configuração.

De acordo com os dados apresentados nesta seção, o *Oprofile Estendido* apresentou um *overhead* superior ao do *Oprofile* para as configurações utilizadas nestes testes. Apesar disso, o pior *overhead* causado pelo *Oprofile Estendido* é de apenas 11,9%, que é aceitável quando se considera o benefício proporcionado pelas informações adicionadas a cada amostra.

9.3 Mergesort

Os programas de ordenação apresentados na Capítulo 8 foram novamente executados e a quantidade de ciclos foi medida de acordo com os três critérios propostos no início

deste capítulo. A Tabela 9.3 mostra as contagens de ciclos registradas pelo *Time Stamp Counter* (TSC) durante as três execuções de cada um dos programas.

A configuração utilizada para o *Oprofile* foi diferente da utilizada no *Oprofile Estendido* para estes programas. Neste experimento, a intenção é demonstrar duas configurações nas quais o comportamento da interferência das duas ferramentas durante o monitoramento é similar.

Programa	Oprofile Estendido		Oprofile		–
Merge L2	$1,380 \cdot 10^9$	10%	$1,377 \cdot 10^9$	10%	$1,252 \cdot 10^9$
T-Merge L2	$1,227 \cdot 10^9$	10%	$1,228 \cdot 10^9$	10%	$1,111 \cdot 10^9$

Tabela 9.3: *Overhead* do OProfile e do OProfile Estendido na ordenação de vetores.

A configuração utilizada nos testes realizados com o *Oprofile* foi: (i) `Cpu_Clk_Unhalted`, 75.000 eventos; (ii) `Data_Cache_Accesses`, 10.000 eventos; (iii) `Data_Cache_Refills_From_L2`, 10.000 eventos; e (iv) `Data_Cache_Refills_From_System`, 10.000 eventos.

Para o *Oprofile Estendido* a configuração utilizada foi: (i) `Cpu_Clk_Unhalted`, 75.000 eventos; (ii) `Data_Cache_Accesses`, $2^{31} - 1$ eventos; (iii) `Data_Cache_Refills_From_L2`, $2^{31} - 1$ eventos; e (iv) `Data_Cache_Refills_From_System`, $2^{31} - 1$ eventos.

As variações apresentadas na tabela mostram que, para estes experimentos e esta configuração, o *overhead* causado pelo *Oprofile Estendido* é o mesmo causado pelo *Oprofile* convencional. Os valores de interferência medidos para as duas ferramentas são de 10%.

Os valores de *overhead* causados pelo *Oprofile Estendido* e o *Oprofile* apresentados neste capítulo mostram diferentes resultados para diferentes combinações de amostragem em cada ferramenta. Estes resultados sugerem um estudo futuro que relacione a taxa de amostragem das duas ferramentas e o *overhead* causado por cada uma delas. A motivação do estudo é encontrar o ponto em que o *overhead* do *Oprofile Estendido* seja tão baixo quanto o do *Oprofile*, sem comprometer a precisão das medidas obtidas.

CAPÍTULO 10

DIVERGÊNCIAS NAS MEDIDAS

O *Oprofile Estendido* utiliza um método de coleta de dados diferente do *Oprofile* original. Enquanto o *Oprofile* realiza uma amostragem de cada contador ao longo da execução de um programa, o *Oprofile Estendido* atualiza a contagem de todos os eventos a cada intervalo de tempo definido pelo usuário.

Dessa forma, o *Oprofile* original não conta 'eventos' mas sim "o número de *overflows* ocorridos" no contador de eventos. Estes contadores são configurados com valores adequados para as métricas, inicializados em zero e incrementados a cada evento. Quando a contagem chega ao limite, o processador é interrompido. Portanto, o número (aproximado) de eventos é o produto entre o número de interrupções e o valor de configuração do contador.

No *Oprofile Estendido*, o processador é interrompido pelo contador de ciclos de execução (`Clock_Unhalted`) de acordo com o intervalo definido pelo usuário, e então os valores em todos os contadores são registrados. Assim, as contagens dos eventos monitorados são atualizadas a cada intervalo de medida, ao invés de acumuladas a cada *overflow* de cada contador configurado.

As diferenças entre a contagem baseada nas amostragens do *Oprofile* e as contagens apresentadas pelo *Oprofile Estendido* variam bastante, principalmente em função da configuração do *Oprofile* e do tempo de execução do experimento. Os dados apresentados neste Capítulo são baseados nas mesmas configurações utilizadas nos experimentos do Capítulo 9.

Os resultados obtidos com o *Oprofile* foram calculados a partir dos relatórios gerados pelo programa `opreport`, incluido na distribuição do *Oprofile*. Embora todos os 20 experimentos de cada programa analisado tenham sido realizados com a mesma configuração, em alguns casos, o `opreport` não gerou os relatórios com a apresentação dos dados no

padrão ideal para a composição da média[1], e por isso estes relatórios não foram incluidos no cálculo. Os resultados do *Oprofile Estendido* correspondem à média de 20 execuções de cada experimento.

Em todas as tabelas apresentadas nesta Capítulo, a primeira coluna apresenta o evento monitorado, a segunda coluna a média da contagem do evento obtida através do *Oprofile Estendido*, a terceira coluna apresenta a média da contagem calculada a partir dos resultados do *Oprofile*.

10.1 Faltas nas TLBs

A Tabela 10.1 apresenta os resultados das contagens de faltas nas TLBs obtidas com o programa da Capítulo 6. Dentre os resultados dos testes realizados para este programa, o que apresenta a maior divergência entre as medidas é o teste que percorre de 22 a 42 páginas. Neste teste, que dura apenas $3{\cdot}10^7$ ciclos, o número de faltas na L1 não atinge o valor de *overflow* do contador e portanto nenhuma falta foi registrada pelo *Oprofile*.

Evento	Oprofile Estendido	Oprofile	Variação
Páginas entre 0 e 350			
`L1_Dtlb_Misses_L2_Dtld_Hits`	$2{,}66{\cdot}10^7$	$2{,}65{\cdot}10^7$	0,37%
`L1_And_L2_Dtlb_Misses`	$4{,}88{\cdot}10^6$	$4{,}64{\cdot}10^6$	5,17%
Páginas entre 22 e 42			
`L1_Dtlb_Misses_L2_Dtld_Hits`	$1{,}88{\cdot}10^5$	0	–
Páginas entre 250 e 280			
`L1_And_L2_Dtlb_Misses`	$3{,}8{\cdot}10^5$	$3{,}5{\cdot}10^6$	8,57%

Tabela 10.1: Divergência na contagem de faltas de páginas.

A segunda maior divergência ocorre no teste que percorre o intervalo de 250 a 280 páginas. Neste teste, a diferença é de uma uma ordem de grandeza a mais para o *Oprofile*. É possível que, em virtude deste também ser um teste rápido – da ordem de 10^8 ciclos, uma a duas ordens de grandeza a menos do que os demais testes – a interferência da inicialização do programa seja a causadora da distorção apresentada na medida do *Oprofile*. Este tipo de interferência não altera o resultado do *Oprofile Estendido* uma vez que a contagem

[1]Os dados não estavam devidamente separados por binário, o que inviabiliza o processamento.

de eventos ocorrida antes da primeira chamada à função `readtsc()` é eliminada no pós processamento dos dados.

As diferenças podem ser parcialmente corrigidas repetindo-se os experimentos com os parâmetros alterados. Uma das possibilidades de correção consiste em aumentar o tempo de execução do programa, aumentando-se o número de vezes que o *buffer* é percorrido até que se aloque mais uma página. Outra possibilidade é reduzir o valor de *overflow* do contador de faltas na L1 TLB do *Oprofile*.

10.2 Multiplicação de Matrizes

Os resultados obtidos nos programas de multiplicação de matrizes podem ser observados na Tabela 10.2. Quanto maior o tempo de execução do programa, menor é a divergência entre os valores medidos pelas duas ferramentas.

Evento	Oprofile Estendido	Oprofile	Variação
Multiplicação Convencional			
`Data_Cache_Accesses`	$1{,}66{\cdot}10^{10}$	$1{,}53{\cdot}10^{10}$	8,49%
`Data_Cache_Refills_From_L2`	$1{,}08{\cdot}10^{9}$	$1{,}08{\cdot}10^{9}$	0%
`Data_Cache_Refills_From_System`	$1{,}07{\cdot}10^{9}$	$1{,}07{\cdot}10^{9}$	0%
Multiplicação Otimizada			
`Data_Cache_Accesses`	$2{,}97{\cdot}10^{10}$	$3{,}06{\cdot}10^{10}$	3,03%
`Data_Cache_Refills_From_L2`	$1{,}03{\cdot}10^{9}$	$1{,}09{\cdot}10^{9}$	5,82%
`Data_Cache_Refills_From_System`	$3{,}4{\cdot}10^{7}$	$2{,}04{\cdot}10^{7}$	66,6%

Tabela 10.2: Divergência na contagem de eventos no produto de matrizes.

No caso da multiplicação de matrizes, o programa convencional, com maior tempo de execução, apresenta uma pequena divergência entre os acessos à *cache*, e aproximadamente os mesmos valores para as demais medidas. No caso da multiplicação otimizada com *padding* e *blocagem*, que executa em aproximadamente um décimo do tempo da multiplicação convencional, as diferenças existem em todos os eventos. Estudos anteriores relacionados à acurácia dos CDHs relacionam o tempo de execução dos testes à precisão dos valores obtidos dos contadores [33, 26].

10.3 Mergesort

O comportamento das medidas dos programas de ordenação de vetores pode ser visto na Tabela 10.3. Nesta tabela, a diferença entre as medidas do *OProfile* e do *Oprofile Estendido* é de, em média, 54% com um desvio padrão de 13 pontos percentuais. Estes experimentos utilizam uma taxa de amostragem 5 vezes maior do que a recomendada pelo fabricante do processador [12], mas não se obtém uma convergência entre os valores obtidos pelas ferramentas.

Evento	Oprofile Estendido	Oprofile Original	Diferença
Mergesort L2			
`Data_Cache_Accesses`	$7{,}35{\cdot}10^8$	$4{,}26{\cdot}10^8$	72,53%
`Data_Cache_Misses`	$4{,}35{\cdot}10^6$	$2{,}5{\cdot}10^6$	74%
`Data_Cache_Refills_From_System`	$1{,}09{\cdot}10^6$	$6{,}91{\cdot}10^5$	57,74%
Tiled Mergesort L2			
`Data_Cache_Accesses`	$5{,}03{\cdot}10^8$	$3{,}47{\cdot}10^8$	44,95%
`Data_Cache_Misses`	$3{,}41{\cdot}10^6$	$2{,}31{\cdot}10^6$	47,61%
`Data_Cache_Refills_From_System`	$1{,}09{\cdot}10^6$	$8{,}39{\cdot}10^5$	29,91%

Tabela 10.3: Divergência na contagem de eventos na ordenação de vetores.

Analisando de uma forma geral a proximidade entre as medidas coletadas das ferramentas e o tempo de duração dos experimentos, é possível estabelecer uma relação entre o tempo de execução de um teste e a diferença entre as medidas coletadas com as duas ferramentas. Esta relação sugere, como estudo futuro, um aprofundamento no estudo da relação entre a precisão da amostragem e o tempo de execução do experimento, complementando estudos como [26].

Ainda em relação à amostragem, diferentemente do *Oprofile*, – em que cada evento mensurável possui um intervalo de amostragem recomendado – a amostragem ideal para o *Oprofile Estendido* pode variar de acordo com o nível de detalhe requerido pelo tipo de análise desejada e pelo tempo de execução do experimento.

Considerando que a contagem de eventos do *Oprofile Estendido* elimina as interferências anteriores e posteriores ao intervalo de interesse, não é necessário executar o teste por tempo longo o suficiente para que o comportamento do programa domine os valores dos contadores. Outro aspecto a ser considerado é que, como mostrado em [31],

os programas podem apresentar um mesmo comportamento que se repete ao longo da execução – como demonstrado aqui, na multiplicação de matrizes e na ordenação de vetores. Nestes casos, a análise de um pequeno trecho do programa pode ser suficiente para compreender toda a sua execução, o que torna desnecessários testes de longa duração para programas que apresentem essas características.

CAPÍTULO 11

CONCLUSÃO

Apresentamos uma extensão do *Oprofile* que permite gerar perfis de execução que combinam várias métricas e a informação de tempo absoluto. Com o *Oprofile Estendido* é possível tomar amostras temporais de todos os contadores de eventos observados a uma taxa de amostragem tão alta quanto permitida pelo projeto da CPU, sem causar perturbação significativa na execução do programa monitorado.

Relatamos três experimentos que demonstram a utilização do *Oprofile Estendido*. O primeiro utiliza um programa de teste que provoca faltas controladas nos dois níveis da TLB, de forma que seja possível observar a evolução das faltas nessas estruturas na medida que o conjunto de trabalho aumenta. As faltas são monitoradas até que seja atingido o estado de *thrashing* nas TLBs. Deste material resultou o artigo "OProfile Estendido para Depuração de Desempenho" [1], apresentado no VI Workshop de Sistemas Operacionais (WSO'2009).

O segundo experimento estuda o comportamento da hierarquia de memória durante a execução de dois programas de multiplicação de matrizes, um convencional e um otimizado. Nos resultados é possível observar a evolução das faltas nos dois níveis da *cache* de dados, além de fazer correlações entre a perturbação que cada evento monitorado causa nos demais.

O terceiro experimento compara o desempenho, com relação à hierarquia de caches, da ordenação de um vetor de `doubles` com duas versões do algoritmo de ordenação Mergesort. Uma é a implementação simples e a outra divide o vetor em faixas do tamanho das caches e emprega o Quicksort nestas faixas. Nos resultados é possível observar a evolução das referências e das faltas nas caches L1 e L2 à medida que as chamadas recursivas ocorrem e que os vetores ordenados são agregados.

Além da análise dos resultados obtidos com as contagens normais dos eventos, reali-

zamos, tanto para a multiplicação de matrizes quanto para a ordenação de vetores, uma análise das variações nas taxas dos eventos monitorados, associando os eventos às fases da execução dos programas.

Analisamos a interferência que o *Oprofile Estendido* causa na execução de cada programa em relação à interferência causada pelo *Oprofile* original em diferentes configurações, e em relação à execução normal dos testes. Encontramos variações de 0,6 a 10% para o *Oprofile*, e de 1 a 10% para o *Oprofile Estendido* em relação à execução normal, de acordo com a configuração utilizada em cada uma das ferramentas, o que serve de base para definir, em experimentos futuros, qual deve ser a melhor relação entre taxa de amostragem e interferência.

Por fim, comparamos as medidas obtidas com o *Oprofile* e com o *Oprofile Estendido* em todos os programas de teste investigados e relacionamos fatores e características – como tempo de execução e configurações das ferramentas – de cada um dos testes às divergências observadas nas medidas. Os parâmetros de configuração do *Oprofile Estendido* devem variar de acordo com o experimento testado, ao contrário do *Oprofile* – e possivelmente demais ferramentas baseadas em amostragem de contadores, que utilizam intervalos de valores pré-definidos para cada evento monitorado.

O *Oprofile Estendido* é uma ferramenta capaz de agregar informações úteis às amostras do *Oprofile* com baixo *overhead*. As informações complementares auxiliam à compreensão da interação entre o programa examinado e a hierarquia de memória, o que é fundamental para depuração de desempenho em sistemas computacionais. Outros eventos monitoráveis podem enriquecer as informações mostradas neste trabalho. Estudos visando ao monitoramento dos demais eventos serão realizados em trabalhos futuros.

Os estudos futuros incluem: (i) relacionar as taxas de amostragem do *Oprofile Estendido* ao *overhead* causado; (ii) relacionar as taxas de amostragem e o tempo de convergência entre as medidas do *Oprofile* e do *Oprofile Estendido*; e (iii) utilizar a visualização da *contagem discriminada* dos dados do *Oprofile Estendido* (Figura 4.3, página 24) para estudar o desempenho da carga de bibliotecas dinâmicas no sistema.

Estamos considerando a integração do *Oprofile Estendido* ao *Oprofile* mediante con-

tato, e desenvolvimento conjunto com o autor do *Oprofile*, ampliando as funcionalidades da ferramenta, o *hardware* suportado e, consequentemente, as possibilidades de utilização.

APÊNDICE A

EVENTOS SUPORTADOS PELO OPROFILE ESTENDIDO

Este apêndice apresenta a saída do comando `opcontrol --list-events`, que lista os eventos monitoráveis no Athlon.

```
info02:/var/lib/oprofile# opcontrol --list-events
oprofile: available events for CPU type "Athlon"

See AMD document x86 optimisation guide (22007.pdf), Appendix D

CPU_CLK_UNHALTED: (counter: all)
        Cycles outside of halt state (min count: 3000)
RETIRED_INSNS: (counter: all)
        Retired instructions (includes exceptions, interrupts, resyncs)
(min count: 3000)
RETIRED_OPS: (counter: all)
        Retired Ops (min count: 500)
ICACHE_FETCHES: (counter: all)
        Instruction cache fetches (min count: 500)
ICACHE_MISSES: (counter: all)
        Instruction cache misses (min count: 500)
DATA_CACHE_ACCESSES: (counter: all)
        Data cache accesses (min count: 500)
DATA_CACHE_MISSES: (counter: all)
        Data cache misses (min count: 500)
DATA_CACHE_REFILLS_FROM_L2: (counter: all)
        Data cache refills from L2 (min count: 500)
        Unit masks (default 0x1f)
        ----------
        0x10: (M)odified cache state
        0x08: (O)wner cache state
        0x04: (E)xclusive cache state
        0x02: (S)hared cache state
        0x01: (I)nvalid cache state
        0x1f: All cache states
DATA_CACHE_REFILLS_FROM_SYSTEM: (counter: all)
        Data cache refills from system (min count: 500)
        Unit masks (default 0x1f)
        ----------
        0x10: (M)odified cache state
        0x08: (O)wner cache state
        0x04: (E)xclusive cache state
        0x02: (S)hared cache state
        0x01: (I)nvalid cache state
        0x1f: All cache states
DATA_CACHE_WRITEBACKS: (counter: all)
        Data cache write backs (min count: 500)
        Unit masks (default 0x1f)
        ----------
        0x10: (M)odified cache state
        0x08: (O)wner cache state
        0x04: (E)xclusive cache state
        0x02: (S)hared cache state
```

```
        0x01: (I)nvalid cache state
        0x1f: All cache states
RETIRED_BRANCHES: (counter: all)
        Retired branches (conditional, unconditional, exceptions, interrupts)
(min count: 500)
RETIRED_BRANCHES_MISPREDICTED: (counter: all)
        Retired branches mispredicted (min count: 500)
RETIRED_TAKEN_BRANCHES: (counter: all)
        Retired taken branches (min count: 500)
RETIRED_TAKEN_BRANCHES_MISPREDICTED: (counter: all)
        Retired taken branches mispredicted (min count: 500)
L1_DTLB_MISSES_L2_DTLD_HITS: (counter: all)
        L1 DTLB misses and L2 DTLB hits (min count: 500)
L1_AND_L2_DTLB_MISSES: (counter: all)
        L1 and L2 DTLB misses (min count: 500)
MISALIGNED_DATA_REFS: (counter: all)
        Misaligned data references (min count: 500)
L1_ITLB_MISSES_L2_ITLB_HITS: (counter: all)
        L1 ITLB misses (and L2 ITLB hits) (min count: 500)
L1_AND_L2_ITLB_MISSES: (counter: all)
        L1 and L2 ITLB misses (min count: 500)
RETIRED_FAR_CONTROL_TRANSFERS: (counter: all)
        Retired far control transfers (min count: 500)
RETIRED_RESYNC_BRANCHES: (counter: all)
        Retired resync branches (only non-control transfer branches counted)
(min count: 500)
INTERRUPTS_MASKED: (counter: all)
        Interrupts masked cycles (IF=0) (min count: 500)
INTERRUPTS_MASKED_PENDING: (counter: all)
        Interrupts masked while pending cycles (INTR while IF=0) (min count: 500)
HARDWARE_INTERRUPTS: (counter: all)
        Number of taken hardware interrupts (min count: 10)
```

APÊNDICE B
MULTIPLICAÇÃO DE MATRIZES BLOCADA

Este apêndice apresenta o código em linguagem 'C' do programa de multiplicação de matrizes blocado, implementado a partir do código disponível em [17].

```
#include <stdio.h>
#include <stdlib.h>
#include <tsc.h>

#define Mtype double
#define MSIZE 1024
#define CLSIZE 16 // # de inteiros pra encher uma linha da cache

int min (int v1, int v2) {

    if (v1 <= v2) return (v1);
    else return (v2);
}

Mtype a[MSIZE][MSIZE];
int pad1 [CLSIZE];
Mtype b[MSIZE][MSIZE];
int pad2 [CLSIZE];
Mtype c[MSIZE][MSIZE];

int main (int argc, char * argv[])
{
    Mtype sum;
    int B, i, j, jj, k, kk;

    if (argc !=2) {
        fprintf(stderr,"Modo de usar: %s <tam. bloco>", argv[0]);
        exit (1);
    }

    B = atoi(argv[1]);

    printf ("%llu 0\n",native_read_tsc());
    for (i = 0; i < MSIZE; i++)
        for (j = 0; j < MSIZE; j++)
            a[i][j] = b[i][j] = c[i][j] = 0.1;

    printf ("%llu 1\n",native_read_tsc());
    for (jj = 0; jj < MSIZE; jj = jj + B)
    for (kk = 0; kk < MSIZE; kk = kk + B)
    for (i = 0; i < MSIZE; i++)
        for (j = jj; j < min(jj + B, MSIZE); j++)
            for (sum = 0, k = kk; k < min(kk + B, MSIZE); k++) {
                sum += a[i][k] * b[k][j];
                c[i][j] = sum;
            }
    printf ("%llu 1\n",native_read_tsc());
exit (0);
}
```

APÊNDICE C
FUNÇÃO READTSC

Este apêndice apresenta o código em linguagem 'C' da função `readtsc()`, escrita com a ajuda de Aristeu Sérgio Rozanski Filho.

```
#include <stdint.h>
static inline unsigned long long native_read_tsc(void)
{
    unsigned long long val;
    asm volatile("rdtsc" : "=A" (val));
    return val;
}
```

APÊNDICE D

EXEMPLO DE CONFIGURAÇÃO DO OPROFILE ESTENDIDO

Este apêndice apresenta um exemplo do arquivo de configuração comentado do *Oprofile Estendido*, normalmente encontrado no diretório `/root/.oprofile/daemonrc`.

```
# Monitorar evento em modo SO -------------+
# Monitorar evento em modo usuário -------+ |
# Máscara (varia entre cada evento) ---+ | |
# Overflow do contador -------------+   | | |
# Evento Monitorado --+             |   | | |
# Contador # -+       |             |   | | |
#             v       v             v   v v v
CHOSEN_EVENTS_0=CPU_CLK_UNHALTED:40000:0:1:1
CHOSEN_EVENTS_1=DATA_CACHE_MISSES:1573741824:0:1:1
CHOSEN_EVENTS_2=L1_DTLB_MISSES_L2_DTLD_HITS:1573741824:0:1:1
CHOSEN_EVENTS_3=L1_AND_L2_ITLB_MISSES:1573741824:0:1:1

# Número total de eventos selecionados
NR_CHOSEN=4

# Separar amostras dos programas por bibliotecas (0 = não, 1 = sim)
SEPARATE_LIB=0

# Separar amostras dos programas quando eles executam código do kernel
SEPARATE_KERNEL=0

# Separar amostras dos programas por thread
SEPARATE_THREAD=1

# Separar as amostras dos programas por CPU
SEPARATE_CPU=0

# Localização da imagem do kernel
VMLINUX=/usr/src/_kernel-source-2.6.8/vmlinux

# Filtrar apenas as amostras geradas por um binário
IMAGE_FILTER=

# Habilita callgraph
CALLGRAPH=0

# Valores dos buffers utilizados. Podem (e devem) ser aumentados de
# acordo com o aumento da taxa de amostragem. Caso não sejam definidos,
# o OProfile utiliza seus valores padrão
CPU_BUF_SIZE=24576
BUF_SIZE=397200
BUF_WATERSHED=98304

# Faixa de memória ocupada pelo kernel (gerado sozinho)
KERNEL_RANGE=c0100000,c0308ba9

# Caso se faça o profile de uma máquina virtual
XENIMAGE=none
# Fim do arquivo
```

BIBLIOGRAFIA

[1] J C Albuquerque e R A Hexsel. Oprofile estendido para depuração de desempenho. *WSO'09 VI Workshop de Sistemas Operacionais*, páginas 1–6, 2009.

[2] J M Anderson et al. Continuous profiling: where have all the cycles gone. *ACM Trans on Computer Systems*, 15(4):357–390, 1997.

[3] T Austin, E Larson, e D Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, Fevereiro de 2002.

[4] R Azimi, M Stumm, e R W Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. *ICS'05: Proc 19th Intl Conf on Supercomputing*, páginas 101–110, 2005.

[5] T Baer. lperfex: A hardware performance monitor for Linux/IA32 Systems. `http://www.osc.edu/~troy/lperfex/`, Janeiro de 2002. Acesso em 30 jul 2008.

[6] R Berrendorf e H Zeigler. PCL, the Performance Counter Library, version 2.3. `http://www.fz-juelich.de/jsc/PCL/`, Julho de 2003. Acesso em 28 jul 2008.

[7] S Browne, J Dongarra, N Garner, G Ho, e P Mucci. A portable programming interface for performance evaluation on modern processors. *The Intl Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[8] B R Buck e J K Hollingsworth. Using hardware performance monitors to isolate memory bottlenecks. *SC Conference*, 0:40, 2000.

[9] J Cavazos, G Fursin, F Agakov, E Bonilla, M F.P. O'Boyle, e O Temam. Rapidly selecting good compiler optimizations using performance counters. *IEEE/ACM International Symposium on Code Generation and Optimization*, 0:185–197, 2007.

[10] J Dean et al. ProfileMe: hardware support for instruction-level profiling on out-of-order processors. *30th IEEE/ACM Intl Symp on Microarchitecture*, páginas 292–302, 1997.

[11] L DeRose e D A Reed. SvPablo: A multi-language performance analisys system. *ICPP'99: Proc 1999 Intl Conf on Parallel Processing*, páginas 311–318, Setembro de 1999.

[12] P J Drongowski. Basic performance measurements for AMD Athlon^{TM}64 and AMD Opteron^{TM} processors. `http://developer.amd.com/Pages/1212200690.aspx`, Dezembro de 2006. Acesso em 30 ago 2007.

[13] T C Ferreto, C A F DeRose, e L DeRose. RVision: An Open and Highly Configurable Tool for Cluster Monitoring. *Proc 2nd IEEE/ACM Intl Symp on Cluster Computing and the Grid*, páginas 75–82, 2002.

[14] T C Ferreto, L DeRose, e C A F DeRose. A hardware counters based tool for system monitoring. *Euro-Par Conf*, páginas 7–16, 2003.

[15] S L Graham, P B Kessler, e M K McKusick. GProf: A Call Graph Execution Profiler. *SIGPLAN Notices*, 39(4):49–57, 2004.

[16] D Heller. Rabbit: A performance counters library for Intel/AMD processors and Linux. `http://www.scl.ameslab.gov/Projects/Rabbit`, Outubro de 2001. Acesso em 30 jul 2008.

[17] J L Hennessy e David A Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 4th edition, 2007. ISBN 0-12-370490-1.

[18] K Hoste e L Eeckhout. The pitfall in comparing benchmarks using hardware performance counters. *ACES Symposium*, páginas 64–67, Outubro de 2006.

[19] Advanced Micro Devices Inc. AMD Athlon processor x86 code optimization guide, Fevereiro de 2004.

[20] C L Janssen. The visual profiler version 0.4. `http://aros.ca.sandia.gov/~cljanss/perf/vprof/`, Outubro de 1999. Acesso em 11 ago 2008.

[21] D Jones. x86info, a CPU identification utility, v1.24. `http://www.codemonkey.org.uk/projects/x86info/`, 2001-2009. Acesso em 3 de mai 2009.

[22] Martin Alain Kretscheck. Panalyser, uma ferramenta de baixo impacto para medição de utilização de recursos do sistema operacional Linux. Dissertação de mestrado, Departamento de Informática, UFPR, Maio de 2002. `http://www.inf.ufpr.br/roberto/dissMartin.pdf`.

[23] M A Kretschek, R A Hexsel, e A L dos Santos. Panalyser, uma ferramenta de baixo impacto para medição de utilização de recursos do sistema operacional linux. *XXX Semin Integrado de Software e Hardware*, páginas 345–358, agosto de 2003.

[24] J Levon. OProfile - A System Profiler for Linux. `http://oprofile.sourceforge.net/news/`, 2003. Acesso em 11 ago 2008.

[25] J Levon. OProfile Manual. `http://oprofile.sourceforge.net/docs/index.php3/`, 2003. Acesso em 11 ago 2008.

[26] P J Teller e L Salayandia M E Maxwell. Accuracy of performance monitoring hardware. *Proc Los Alamos Computer Science Institute Symposium*, Outubro de 2002.

[27] S V Moore. A comparison of counting and sampling modes of using performance monitoring hardware. *Intl Conf on Computational Science*, páginas 904–912, 2002.

[28] T Mytkowicz, P F Sweeney, M Hauswirth, e A Diwan. Time interpolation: so many metrics, so few registers. *40th IEEE/ACM Intl Symp on Microarchitecture*, páginas 286–298, 2007.

[29] M Petterson. Linux x86 performance-monitoring counter's driver, version 2.6.39. `http://user.it.uu.se/~mikpe/linux/perfctr`, Junho de 2008. Acesso em 30 jul 2008.

[30] J Renau. SESC: cycle accurate architectural simulator. `http://sesc.sourceforge.net/index.html`, 2002. Acesso em 25 de mai 2009.

[31] T Sherwood, E Perelman, G Hamerly, e B Calder. Automatically characterizing large scale program behavior. *SIGOPS Operating System Review*, 36(5):45–57, 2002.

[32] The Standard Performance Evaluation Corporation (SPEC). SPEC CPU2006. `http://www.spec.org/`, 2006. Acesso em 11 ago 2008.

[33] W Korn W, P J Teller, e G Castillo. Just how accurate are performance counters? *IEEE Performance, Computing, and Communications*, páginas 303–310, 2001.

[34] V M Weaver e S A McKee. Are cycle accurate simulations a waste of time? *7th Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, páginas 40–53, Junho de 2008.

[35] V M Weaver e S A McKee. Can hardware performance counters be trusted? *IEEE International Symposium on Workload Characterization*, páginas 141–150, Setembro de 2008.

JOÃO CLAUDIO MUSSI DE ALBUQUERQUE

ANÁLISE DO COMPORTAMENTO DA HIERARQUIA DE MEMÓRIA COM OPROFILE ESTENDIDO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Roberto André Hexsel

CURITIBA

2009