API para Monitoramento de Desempenho em Sistemas Multicore Embarcados

Giovani Gracioli e Antônio Augusto Fröhlich

[1]Laboratório de Integração Software/Hardware (LISHA)
Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476 – 88.040-900 – Florianópolis – SC – Brasil
{giovani,guto}@lisha.ufsc.br

Abstract. *Hardware Performance Counters (HPCs) are special registers available in the most modern processors that can be used to monitor shared hardware resources in multicore processors. Specifically for embedded real-time applications running on a multicore processor, such shared resources can affect their performance and cause deadline misses. This paper presents a hardware performance counter interface designed to embedded systems. The use of the interface is demonstrated through a benchmark that shares data between two threads executing in different cores of a multicore processor. As a result, the operating system can obtain an accurate view of software's behavior.*

Resumo. *Contadores de desempenho de hardware (CDHs) são registradores disponíveis nos processadores atuais capazes de contar eventos microarquiteturais. Em especial para aplicações embarcadas de tempo real implementadas em processadores multicore, os CDHs podem ajudar o SO a obter uma visão precisa dos recursos físicos que estão sendo compartilhados pelas threads do sistema e que podem causar a perda de deadlines. Este artigo apresenta uma interface de CDHs especificamente projetada para sistemas embarcados. O uso da interface é demonstrado através de um benchmark que compartilha dados entre duas threads executando em diferentes cores. Como resultado, o SO é capaz de obter uma visão correta sobre o comportamento da aplicação.*

1. Introdução

Processadores *multicore* (com múltiplos núcleos) estão sendo cada dia mais usados no contexto de sistemas embarcados de tempo real (SETR) devido à evolução e integração de funcionalidades em tais sistemas. Por exemplo, em um automóvel moderno, novas funções de segurança como "parada automática de emergência" e "auxílio de visão noturna" devem ler dados dos sensores, processar o vídeo e exibir avisos preventivos quando um obstáculo é detectado na via em tempo real [Mohan et al. 2011]. Além disso, a adição de novas funcionalidades ao sistema custa em termos de consumo de energia, dissipação de calor e espaço (e.g., cabeamento) [Cullmann et al. 2010]. Assim, processadores *multicore* tornam-se uma boa alternativa para diminuir esses custos e integrar as diversas funções em uma única unidade de processamento, ao contrário de se ter diversas ECUs (unidades de controle eletrônicas) espalhadas no veículo.

Entretanto, é difícil garantir que os *deadlines* de tempo serão atendidos em uma arquitetura *multicore*, principalmente devido ao compartilhamento

de diferentes recursos físicos, tais como a memória cache, barramentos e periféricos [Wehmeyer and Marwedel 2005, Zhuravlev et al. 2010]. Neste contexto, é importante que o sistema operacional de tempo real (SOTR) seja capaz de corretamente monitorar quando a disputa por recursos compartilhados influencia o tempo de execução das threads e, consequentemente, a perda de *deadlines*.

Um exemplo de compartilhamento de recursos em um processador *multicore*, que pode ocasionar a perda de *deadlines*, é o compartilhamento de dados. Tradicionalmente, cada *core* possui uma cache privada e uma cache de nível 2 ou 3, que é compartilhada por todos os *cores*. Quando existe dado compartilhado, cada cópia do dado é armazenada na cache privada dos *cores* e um protocolo de coerência de cache garante a consistência entre as cópias, através de *snooping* no barramento. Quando um *core* escreve no dado compartilhado, o protocolo de coerência invalida todas as cópias, causando um atraso implícito na escrita do dado. A leitura e escrita frequentes a um dado compartilhado causa a serialização de acessos à mesma linha da cache e a saturação do barramento entre os *cores*, aumentando o tempo de execução da aplicação [Boyd-Wickizer et al. 2010].

Neste contexto, contadores de desempenho de hardware (HPCs - *Hardware Performance Counters*) são uma boa alternativa para monitorar quando eventos de hardware que podem causar disputa por recursos compartilhados acontecem. HPCs são registradores especiais presentes na maioria dos microprocessadores modernos através de uma unidade de monitoramento de desempenho (PMU - *Performance Monitoring Unit*). HPCs oferecem suporte para contar ou amostrar diversos eventos microarquiteturais. Em um processador *multicore*, por exemplo, é possível contar o número de *snoops* no barramento, número de ciclos em que dados são enviados pelo barramento, entre outros. Com base nas medições dos HPCs, o SOTR pode tomar uma decisão, como escalonar ou não uma thread em um determinado instante ou movê-la para um outro *core*.

Este artigo apresenta uma interface para uma família de PMUs especificamente projetada para sistemas embarcados. A interface utiliza o conceito de mediadores de hardware [Polpeta and Fröhlich 2004] para criar uma camada de comunicação entre o software e o hardware de maneira simples, eficiente, com baixo consumo de memória e portável. Para efeitos de demonstração de uso da interface proposta, uma aplicação que gera excessivas invalidações na mesma linha de cache é utilizada e seu código fonte é relacionado com os eventos mensurados pelos HPCs.

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta, brevemente, o conceito de mediadores de hardware. A seção 3 apresenta a interface para uma família de PMUs proposta neste artigo. A correlação entre o código fonte da aplicação e os eventos de hardware mensurados pelos HPCs é apresentada na Seção 4. A Seção 5 discute os trabalhos relacionados e a Seção 6 conclui o artigo.

2. Mediadores de Hardware

A metodologia de Projeto de Sistemas Embarcados Dirigido pela Aplicação (ADESD) [Fröhlich 2001], define o conceito de mediadores de hardware [Polpeta and Fröhlich 2004]. Mediadores de hardware são funcionalmente equivalentes aos *drivers* de dispositivos em SOs baseados em UNIX, mas não apresentam uma camada de abstração de hardware (HAL - *Hardware Abstraction Layer*) tradicional. Mediadores provêm uma interface entre os componentes do SO e o hardware através

de técnicas de metaprogramação estática e *inlining* de métodos que diluem o código do mediador nos componentes em tempo de execução. Consequentemente, o código gerado não possui chamadas a métodos, nem camadas ou mensagens, atingindo uma maior portabilidade e reúso em comparação com as HALs tradicionais [Marcondes et al. 2006].

A Figura 1 apresenta um exemplo de mediador de hardware para uma família de CPUs. Este mediador trata a maioria das dependências do gerenciamento de processos. A classe CPU::CONTEXT é definida por cada arquitetura e representa os dados necessários para um fluxo de execução, ou seja, o contexto de execução. O método *CPU::switch_context* é responsável pela troca de contexto, recebendo o contexto antigo e o novo. Os mediadores de CPU implementam também uma série de funcionalidades como habilitação e desabilitação de interrupções e operação de bloqueio (e.g., *test and set lock*). Cada arquitetura ainda define um conjunto de registradores e endereços específicos, porém a mesma interface permanece. Assim, é possível manter as mesmas operações para os componentes independentes de arquitetura que usam a CPU tais como thread, sincronizadores e temporizadores.

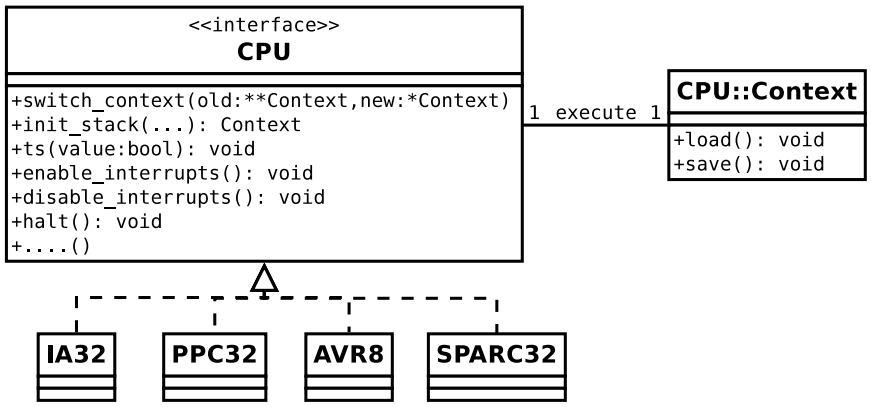

Figura 1. Mediador de hardware CPU.

3. Interface para PMU

Através do uso do conceito de mediadores de hardware apresentado, uma interface para a família de PMUs da Intel foi projetada. A Figura 2 mostra o diagrama de classes UML da interface proposta. Os processadores Intel, dependendo da microarquitetura (e.g., Nehalem, Core, Atom, etc), possuem diferentes versões da PMU. Cada versão provê diferentes funcionalidades e um número variável de contadores de hardware. Por exemplo, a PMU versão 2 possui dois contadores que podem ser configurados com eventos gerais e três contadores fixos que contam eventos específicos. Já a versão 3 estende a versão 2 e tem suporte para *multi-threading* (SMT) e até 8 contadores [Intel Corporation 2011]. Existem também eventos arquiteturais pré-definidos que são compartilhados pelas 3 versões, como número de *misses* no último nível de cache e número de instruções executadas.

A tarefa de configurar um evento envolve a programação de um registrador de seleção (IA32_PERFEVTSELx) correspondente a um contador físico (IA32_PMCx). Existem diversas máscaras associadas a cada evento e máscaras de controle que devem ser escritas no registrador de evento para que o contador comece a contar o evento selecionado. Por exemplo, para configurar o registrador PMC0 para contar o número de

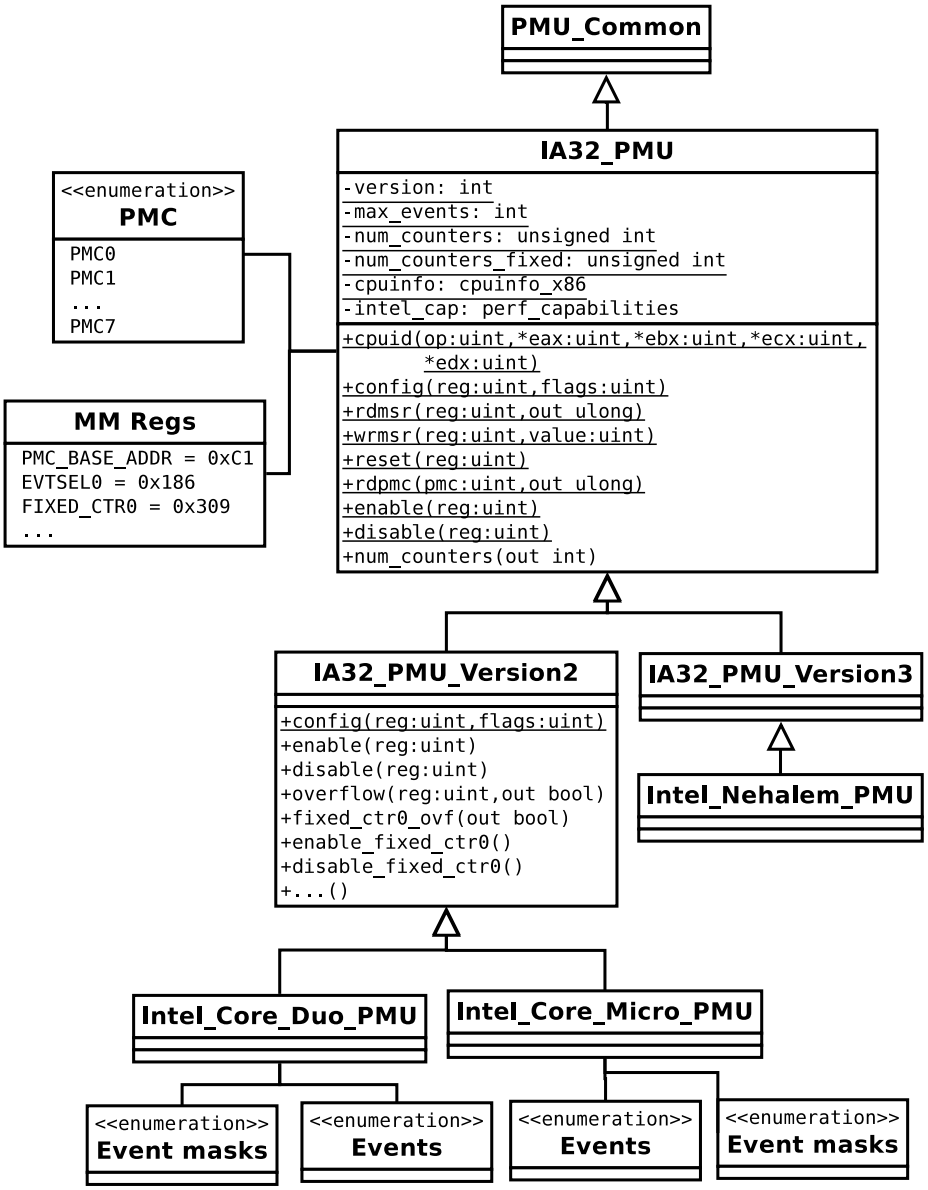

Figura 2. Diagrama de classes UML da interface de mediadores de hardware da família de PMUs da Intel.

respostas *snoop* de transações do barramento, o PERFEVTSEL0 deve ser escrito com o máscara do evento EXT_SNOOP (0x77) e duas máscaras que definem a condição de quando o evento deve ser contado.

A família de mediadores de hardware projetada reflete a organização das PMUs descritas. Uma classe base IA32_PMU implementa a PMU versão 1 e serviços comuns às versões 2 e 3, incluindo os eventos arquiteturais pré-definidos. Essa classe ainda declara os registradores mapeados em memória e os contadores de hardware (PMCs). A classe IA32_PMU_Version2 e IA32_PMU_Version3 estendem a classe base e implementam os serviços específicos somente disponíveis àquela versão. Finalmente, os eventos de hardware disponíveis para cada microarquitetura são listados por suas respectivas classes. Por exemplo, a classe Intel_Core_Micro_PMU lista todas máscaras de eventos relacionadas

com a microarquitetura Intel Core.

A interface de mediadores de hardware poderia ser usada por um componente independente de plataforma. Esse componente seria o responsável por agregar inteligência aos mediadores. Por exemplo, taxas como paralelização, dados compartilhados modificados e utilização do barramento combinam dois ou mais eventos de hardware com o intuito de prover uma melhor compreensão do comportamento e desempenho das aplicações [Malladi]. Além disso, este componente poderia multiplexar os contadores para superar a limitação do número de contadores físicos. Técnicas de multiplexação dividem o uso dos contadores no tempo, dando a visão de que existem mais contadores em hardware do que o processador possui. No entanto, o uso de multiplexação pode diminuir a precisão [Dongarra et al. 2003].

Para demonstrar o desempenho dos mediadores, o PMC0 foi usado para contar o evento EXT_SNOOP. O método para configurar o contador ocupou 32 bytes de código e 11 instruções. Já o método para a leitura do contador ocupou 100 bytes e 40 instruções. Outras APIS projetadas para sistemas computacionais de propósito geral, tal como a PAPI [Mucci et al. 1999], necessitam a inicialização e criação de listas ou conjunto de eventos que afetam o desempenho. Além disso, essas APIs disponibilizam diversas funcionalidades, como gerenciamento de objetos ativos e tratamento de erros, que podem não ser de interesse de uma aplicação embarcada e de tempo real. Assim, uma interface especificamente projetada para suprir as necessidades de uma aplicação pode obter um melhor desempenho. Não é o objetivo deste artigo, no entanto, comparar os mediadores propostos com as APIs de propósito geral pois elas são de domínios diferentes. É importante salientar também que o mesmo conceito de interface para PMUs pode ser aplicado em outras famílias de processadores, como PowerPC e ARM.

4. Exemplo de Uso

Com o intuito de demonstrar a utilidade da interface proposta em um processador *multicore*, um *benchmark* composto por duas versões de uma mesma aplicação e uma aplicação de melhor-caso para efeitos comparativos foram desenvolvidos (veja Figura 3):

- **Sequencial**: nesta versão, duas funções executam em ordem sequencial. Não existe qualquer tipo de compartilhamento de dados (Figura 3(a)).
- **Paralela**: duas threads executam ao mesmo tempo e compartilham dados (Figura 3(b)). O objetivo desta versão é avaliar o desempenho da versão anterior quando implementada em um processador *multicore*. As funções 1 e 2 de ambas versões têm as mesmas operações e acessos a memória (veja Figura 4). Esta versão deveria ser executada duas vezes mais rápida que a versão anterior. Não foram utilizados nenhum tipo de sincronizadores entre as threads pois o objetivo é demonstrar, através dos HPCs, o quanto a disputa por dados compartilhados afeta o desempenho da aplicação.
- **Melhor-caso**: duas threads executam em paralelo mas não há compartilhamento de dados (Figura 3(c)). O objetivo desta aplicação é ter um cenário onde há paralelismo mas não há compartilhamento de dados.

A interface de mediadores proposta e o *benchmark* foram implementados no EPOS[1] [Fröhlich 2001]. As 3 aplicações possuem dois arrays bi-dimensionais com ta-

[1]EPOS está disponível online no endereço `http://epos.lisha.ufsc.br`.

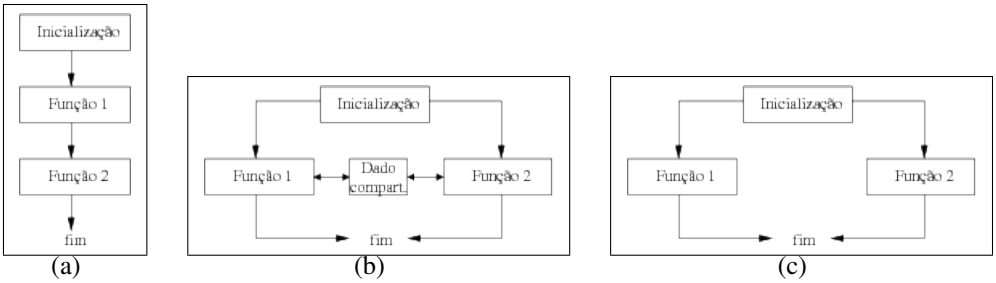

Figura 3. Aplicações do *benchmark*: (a) sequencial (b) paralela (c) melhor-caso.

```
1  register unsigned int sum0;
2  register unsigned int sum1;
3
4  for(unsigned int i = 0; i <= REP; i++) {
5     for(unsigned int j = 0; j < ROWS; j++) {
6      sum0 = 0;
7      sum1 = 0;
8        for(unsigned int k = 0; k < COLS; k++) {
9           sum0 += array0[j][k];
10          sum1 += array1[j][k];
11       }
12       for(unsigned int k = 0; k < COLS; k++) {
13          array0[j][0] = sum0;
14          array1[j][0] = sum1;
15       }
16    }
17 }
```

Figura 4. Parte do código fonte das aplicações sequencial e paralela.

manho variável (ROWS x COLS) e um laço de 10000 repetições (veja Figura 4). O escalonador utilizado nos experimentos foi o CPU_Affinity, no qual cada thread possui uma afinidade com um *core*. Todos os testes foram executados no processador Intel Core 2 Q9550 (Tabela 1).

Intel Core 2 Q9550	
Frequência	2.83 Ghz
Conjunto de instruções	64 bits
Número de cores	4 (2 dual-core)
Velocidade do barramento	Front-side bus 1.3 Ghz
Cache nível 1 privada (L1)	64 KB 8-way set associative
Tamanho da cache L2	12 MB (2x 6 MB) 24-way set associative
Tamanho da linha de cache	64 bytes
Arquitetura de memória	UMA
Protocolo de coerência	MESI
Versão da PMU	2
Microarquitetura	Intel Core Microarchitecture

Tabela 1. Configuração do processador Intel Core 2 Q9550.

Primeiramente, a média de 10 execuções de cada aplicação foi medida. A Figura 5 apresenta os valores obtidos através do componente `Chronometer` em escala logarítmica para cada versão. As dimensões dos dois arrays (ROWS x COLS) foram configuradas em 64x64 (32 KB), 128x128 (128 KB), 256x256 (512 KB), 512x512 (2 MB), 1024x1024 (8 MB), 1536x1024 (12 MB), e 1536x1536 (18 MB) que excede o tamanho da cache L2 (12 MB). O desvio padrão apresentou uma variação entre 0,01% e 1% do total do tempo de execução. É possível notar que o tempo de execução da versão paralela é

sempre superior ao da versão sequencial, até 1,55 vez mais lenta. Devido ao compartilhamento dos dados e consequentemente a invalidações da mesma linha de cache realizadas pelo protocolo de coerência, o tempo de execução da versão paralela é afetado. Como esperado, a versão de melhor-caso foi cerca de 2 vezes mais rápida que a sequencial. As linhas 9, 10, 13 e 14 da Figura 4 são as responsáveis por gerar as transações no barramento que aumentam o tempo de execução.

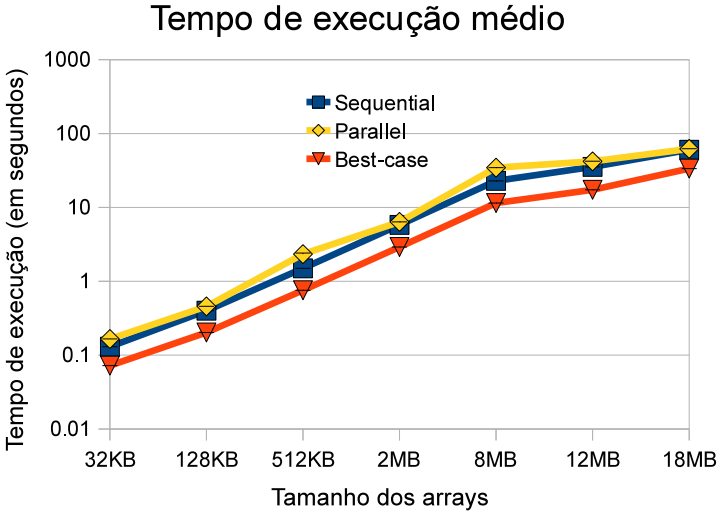

Figura 5. Tempo de execução do *benchmark* para cada tamanho dos arrays.

Com isso é possível constatar que o tempo de execução das aplicações pode ser afetado pela arquitetura dos processadores *multicore* atuais e, em caso de aplicações com restrições de tempo, *deadlines* podem ser perdidos. Assim, é extremamente desejável que o SO seja capaz de monitorar e detectar em tempo real quando atividades que geram disputa por recursos compartilhados aconteçam, a fim de tomar decisões para que o desempenho das aplicações não seja tão degradado. Portanto, a família de mediadores de hardware proposta neste artigo é adequada para este fim.

Neste sentido, o próximo experimento realizado exemplifica o uso da interface para medir o efeito de excessivas invalidações da mesma linha de cache no hardware. O evento medido foi o CMP_SNOOP. Esse evento conta o número de vezes que dados presentes na cache L1 de um *core* são requisitados (*snooped*) por outro. Um *snoop* é realizado através de uma porta de escrita na cache de dados L1. Consequentemente, frequentes *snoops* podem conflitar com escritas à cache de dados L1 e assim aumentar a latência e impactar o desempenho [Intel Corporation 2011]. A Figura 6 mostra parte do código fonte das aplicações utilizando o componente de monitoramento de desempenho (*Perf_Mon*) que faz uso do mediador de hardware implementado. As chamadas aos métodos de configuração (*l1_data_cache_snooped*) e leitura (*get_l1_data_cache_snooped*) do evento são diluídas em tempo de compilação no código das aplicações, assim nenhuma chamada de método é gerada na imagem final do sistema.

A Figura 7 mostra o número de eventos mensurados para cada versão do *benchmark* e para cada tamanho dos arrays. O evento foi configurado para contar os *snoops* que geram invalidações da mesma linha de cache. Uma invalidação em uma linha de cache

```
1  ...
2  Semaphore s;
3  ...
4  int func0(void)
5  {
6   #ifndef __SEQUENTIAL
7   Perf_Mon perf0;
8   perf0.l1_data_cache_snooped();
9   #endif
10  register unsigned int sum0;
11  ....
12
13  #ifndef __SEQUENTIAL
14  s.p();
15  cout << "\nL1 data cache snooped func0 = " <<
        perf0.get_l1_data_cache_snooped() << "\n";
16  s.v();
17  #endif
18 }
19 ...
```

Figura 6. Exemplo de como a API foi utilizada nos experimentos nas aplicações paralela e melhor-caso.

acontece quando o *core* não possui aquela linha ou quando a linha está disponível somente para leitura e o outro *core* deseja escrever nessa linha. Basicamente, as linhas 13 e 14 do código fonte da Figura 4 são responsáveis pelas invalidações mensuradas por este evento.

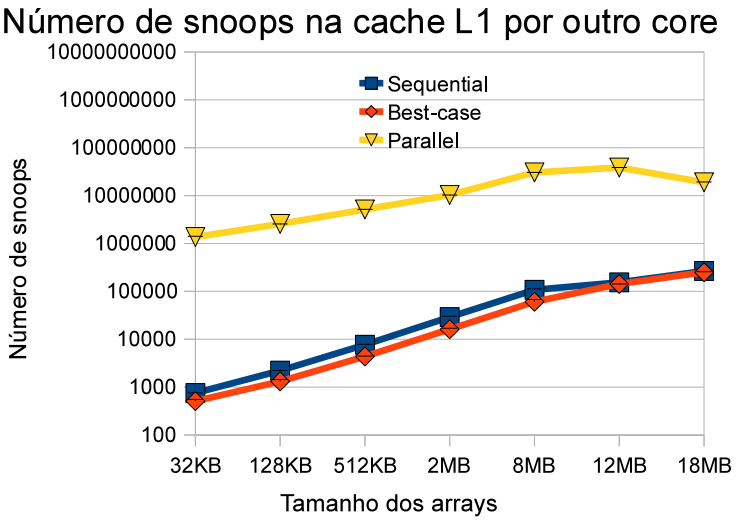

Figura 7. Número de *snoops* de uma linha de cache. O evento CMP_SNOOP foi configurado para contar os *snoops* de invalidação de todos *cores*.

A versão paralela apresentou de 2 a 3 ordens de magnitude mais *snoops* que as versões sequencial e melhor-caso. Quando o tamanho dos dados é maior que a cache L2 (de 12 MB a 18 MB), a versão paralela apresentou uma pequena queda no número de *snoops* devido ao fato de que existem mais dados sendo substituídos na cache (e.g., conflitos pela mesma linha de cache) e assim a probabilidade de um dado requisitado não estar na cache é menor. O desvio padrão do melhor-caso variou de 10% para 32 KB e 128 KB a 0,05% para 18 MB. O desvio padrão máximo para as versões sequencial e paralela foi 4,82% para tamanho de 2 MB e 3,49% para 62 KB respectivamente. O número total de

snoops representa de 0,04% a 1,67% do total de escritas realizadas pelo código em um *core* para a versão paralela e cerca de 0,0005% para as versões sequencial e melhor-caso. Os *snoops* mensurados nas versões sequencial e melhor-caso são resultado das variáveis de controle (*spinlocks*) necessárias em um kernel *multicore*.. As linhas 13 e 14 da Figura 4 são responsáveis pelas escritas que geram o evento medido. As leituras não são consideradas pois elas não geram transações de invalidações no barramento. É importante salientar que nem toda escrita ou leitura de um dado vai gerar um *snoop* no barramento. Dependendo do estado da linha da cache em um *core* (dado pelo protocolo de coerência), uma ação é tomada pelo controlador de memória. Por exemplo, considerando o protocolo de coerência MESI, se a linha de cache em um *core* está no estado E (exclusivo) e uma requisição para leitura é vista no barramento, este *core* deve mudar a linha para o estado S (compartilhado). Por outro lado, se um *core* escreve em uma linha de cache no estado E ou M, nenhuma transação no barramento é gerada pois esta linha só está presente na cache daquele *core*. Invalidações da linha de cache acontecem apenas em escritas nas linhas de cache compartilhadas (estado S) ou inválidas (estado I).

Os valores medidos na Figura 7 foram então normalizados em um período de 10 ms dado pela Equação 1:

$$N = \frac{AVG_s}{AVG_{ex}/10ms} \tag{1}$$

Onde AVG_s é a média de *snoops* e AVG_{ex} é a média do tempo de execução. A Figura 8 mostra os valores normalizados para cada aplicação e tamanho dos dados. Nota-se que praticamente não existe variação para as aplicações sequencial e melhor-caso, cerca de 60 a 80 *snoops* em um período de 10 ms. A versão paralela apresentou até 100.000 *snoops* em um período. Esse gráfico mostra a frequência de invalidações de uma linha da cache e como um evento de hardware pode ser utilizado pelo SO. Por exemplo, durante um quantum de escalonamento, esse evento poderia ser monitorado e se o valor mensurado fosse maior que um valor limite, o escalonador poderia tomar uma decisão de escalonar ou não uma thread, movê-la para o mesmo *core* de outra ou ainda diminuir o grau de paralelismo para diminuir a disputa pelos dados compartilhados.

A API proposta foi implementada em C++ em sistema operacional baseado em componentes (EPOS). Sendo assim, a mesma infra-estrutura de mediadores e componentes poderia ser facilmente utilizada por qualquer outro sistema operacional baseado em componentes e implementado em C++.

5. Trabalhos Relacionados

PAPI (*Performance API*) é a interface de código aberto mais utilizada para monitoramento de desempenho através de HPCs [Dongarra et al. 2003, Mucci et al. 1999]. PAPI consiste de uma camada independente de plataforma, uma interface de alto nível que provê comandos como leitura, inicialização e parada dos contadores e uma interface de baixo nível que disponibiliza mais funcionalidades aos desenvolvedores. Essa camada de baixo nível poderia ser comparada aos mediadores de hardware propostos neste artigo. Entretanto, não é objetivo nesse trabalho, fazer uma comparação entre as duas implementações. PAPI suporta uma grande variedade de arquiteturas, tendo sido projetada para sistemas computacionais de propósito geral e está em desenvolvimento por

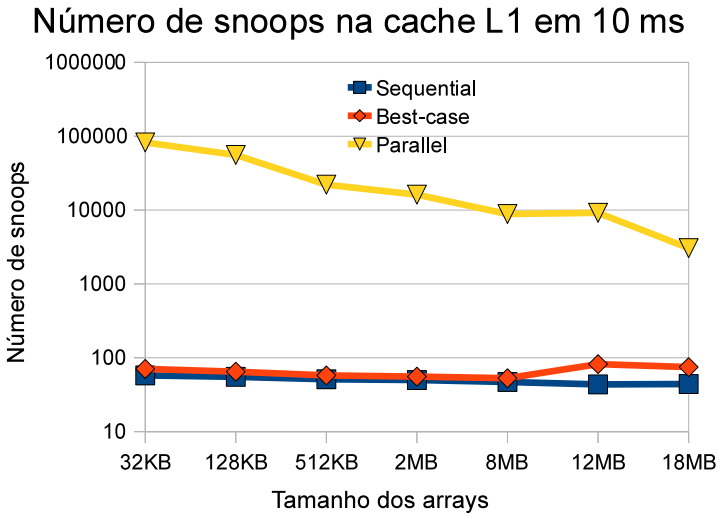

Figura 8. Número de *snoops* de uma linha de cache em um período de 10 ms.

mais de 10 anos. A interface proposta, ao contrário, foi projetada para aplicações embarcadas que possuem seus requisitos conhecidos em tempo de projeto. Assim, é possível gerar somente o código necessário para a aplicação, economizando recursos e deixando a comunicação entre software e hardware mais rápida.

O LINUX abstrai o uso de HPCs através de um subsistema de contadores de desempenho. A ferramenta *perf*, disponível juntamente com o código fonte do SO, usa esse subsistema e permite que desenvolvedores obtenham e analisem o desempenho de suas aplicações. Outras ferramentas como a Intel VTUNE [Malladi] e AMD CODEANALYST [Drongowski 2008] oferecem uma interface gráfica "amigável" para monitorar o desempenho de processadores e aplicação através de HPCs. Entretanto, SETR geralmente não apresentam uma interface entre o desenvolvedor/usuário e o sistema em si.

Existem trabalhos que utilizam HPCs juntamente com o escalonador do SO para dinamicamente tomar decisões como mover threads de um *core* para outro. Bellosa e Steckermeier foram os primeiros autores a usar HPCs com esse propósito [Bellosa and Steckermeier 1996]. Weissman usa HPCs para monitorar o número de cache *misses* em um quantum de escalonamento e, juntamente com anotações do desenvolvedor no código fonte, montar um grafo representando as dependências do estado da cache entre as threads [Weissman 1998]. O objetivo é avaliar os efeitos do tamanho dos dados das threads na cache de um processador. Com base no modelo, são propostas duas políticas de escalonamento, porém elas não consideram invalidações na mesma linha de cache devido ao compartilhamento de dados.

Tam et al. usaram HPCs para monitorar os endereços de linhas de cache que são invalidados devido à coerência de cache. Uma estrutura de dados que contém os endereços lidos é construída para cada thread do sistema [Tam et al. 2007]. Usando as informações desta estrutura, o escalonador monta um grupo composto por um conjunto de threads e aloca este grupo a um *core* específico. West et al. propuseram uma técnica baseada em um modelo estatístico para estimar a ocupação da cache por cada thread do sistema em tempo de execução [West et al. 2010]. No entanto, compartilhamento de dados entre as

threads também não é considerado pelos autores.

Zhuravlev identifica os principais problemas que causam contenção em processadores *multicore*: contenção pelo controlador de memória, barramento de memória, hardware *prefetching* e espaço da cache [Zhuravlev et al. 2010]. Os autores ainda propõem dois algoritmos de escalonamento que usam a taxa de *miss* da cache para distribuir as threads de forma equilibrada entre os *cores* e assim diminuir esta taxa e aumentar o desempenho das aplicações. Calandrino e Anderson também utilizam o número de *misses* da cache para estimar o tamanho dos dados de cada thread [Calandrino and Anderson 2009]. Baseando-se nesse número estimado, um escalonador de tempo real e consciente do tamanho da cache, escalona as threads nos *cores* de maneira que não causem *thrashing* na cache compartilhada.

6. Considerações Finais

Este artigo apresentou o projeto de uma família de mediadores de hardware para unidades de monitoramento de desempenho presentes nos processadores atuais, dentro do contexto de sistemas embarcados. O código gerado pelos mediadores é diluído em tempo de compilação nos componentes que os utilizam. Assim, é obtido um baixo consumo de memória e um bom desempenho, características desejáveis para um sistema embarcado. Com o intuito de demonstrar a utilização dos mediadores propostos, um *benchmark* que compartilha dados entre duas threads foi implementado e os mediadores propostos foram usados para prover ao sistema operacional uma visão da relação entre o software em execução e os eventos gerados no hardware por esse software.

Como trabalhos futuros, pretendemos utilizar o suporte a HPCs juntamente com o escalonador para garantir o atendimento aos *deadlines* em aplicações embarcadas de tempo real. Um componente independente de plataforma responsável por agregar funcionalidades à família de mediadores de PMU também será alvo de implementação futura.

Agradecimentos

Este trabalho foi financiado parcialmente pela CAPES, projeto RH-TVD 006/2008.

Referências

Bellosa, F. and Steckermeier, M. (1996). The performance implications of locality information usage in shared-memory multiprocessors. *Journal of Parallel Distributed Computing*, 37:113–121.

Boyd-Wickizer, S., Clements, A. T., Mao, Y., Pesterev, A., Kaashoek, M. F., Morris, R., and Zeldovich, N. (2010). An Analysis of Linux Scalability to Many Cores. In *OSDI 2010: Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*.

Calandrino, J. M. and Anderson, J. H. (2009). On the design and implementation of a cache-aware multicore real-time scheduler. In *Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems*, pages 194–204, Washington, DC, USA. IEEE Computer Society.

Cullmann, C., Ferdinand, C., Gebhard, G., Grund, D., Maiza, C., Reineke, J., Triquet, B., Wegener, S., and Wilhelm, R. (2010). Predictability considerations in the design of multi-core embedded systems. *Ingénieurs de l'Automobile*, 807:36–42.

Dongarra, J., London, K., Moore, S., Mucci, P., Terpstra, D., You, H., and Zhou, M. (2003). Experiences and lessons learned with a portable interface to hardware performance counters. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 289.2–, Washington, DC, USA. IEEE Computer Society.

Drongowski, P. J. (2008). *An introduction to analysis and optimization with AMD CodeAnalyst Performance Analyzer*.

Fröhlich, A. A. (2001). *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin.

Intel Corporation (2011). *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253668-037US.

Malladi, R. K. *Using Intel® VTuneTM Performance Analyzer Events/Ratios Optimizing Applications*. Number Intel® White Paper.

Marcondes, H., Hoeller, A. S., Wanner, L. F., and Fröhlich, A. A. (2006). Operating systems portability: 8 bits and beyond. In *ETFA '06: IEEE Conference on Emerging Technologies and Factory Automation*, pages 124 –130.

Mohan, S., Caccamo, M., Sha, L., Pellizzoni, R., Arundale, G., Kegley, R., and de Niz, D. (2011). Using multicore architectures in cyber-physical systems. In *Workshop on Developing Dependable and Secure Automotive Cyber-Physical Systems from Components*, Michigan, USA.

Mucci, P. J., Browne, S., Deane, C., and Ho, G. (1999). Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10.

Polpeta, F. V. and Fröhlich, A. A. (2004). Hardware mediators: a portability artifact for component-based systems. In *In Proceedings of the International Conference on Embedded and Ubiquitous Computing*, volume 3207 of *Lecture Notes in Computer Science*, pages 45–53, Aizu, Japan. Springer Berlin / Heidelberg.

Tam, D., Azimi, R., and Stumm, M. (2007). Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. *SIGOPS Operating System Review*, 41:47–58.

Wehmeyer, L. and Marwedel, P. (2005). Influence of memory hierarchies on predictability for time constrained embedded software. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 600–605, Washington, DC, USA. IEEE Computer Society.

Weissman, B. (1998). Performance counters and state sharing annotations: a unified approach to thread locality. *SIGOPS Operating System Review*, 32:127–138.

West, R., Zaroo, P., Waldspurger, C. A., and Zhang, X. (2010). Online cache modeling for commodity multicore processors. *SIGOPS Operating System Review*, 44:19–29.

Zhuravlev, S., Blagodurov, S., and Fedorova, A. (2010). Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 129–142.