Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators

Yunsup Lee*, Rimas Avizienis*, Alex Bishara*, Richard Xia*, Derek Lockhart†, Christopher Batten†, and Krste Asanovi ´c* - ISCA 2011

Leandro Negri Zanotto RA: 001963 - Arquitetura de Computadores MO401 - Paulo Centoducatte

Este artigo apresenta o desenvolvimento de uma nova microarquitetura baseada em vector thread (vector-SIMD) chamada Maven. Também é mostrado o comportamento de outras arquiteturas como MIMD, vector-SIMD, subword-SIMD, SIMT e vector-thread que são padrões de arquitetura em processamento paralelo. Muitas áreas de pesquisa como exatas e biológicas produzem uma grande quantidade de dados a serem processados. Para acelerar o processamento utiliza-se aceleradores especializados em computação paralela, com isso tem-se uma redução no consumo de energia mesmo em processadores com um grande número de transistores. Os aceleradores são geralmente anexados a CPU, podendo ou não estar no mesmo die (pequeno bloco de material semicondutor). Os processadores fazem o processamento sequencial enquanto o paralelo é executado pelos aceleradores através de kernels. Com isso é obtido uma melhor eficiência de energia e um tempo de resposta do processamento menor.

Nas aplicações que utilizam processamento paralelo, os kernels encontrados são divididos em DLP (Data Level Parallelism) regular e irregular. DLP regular possui controle de acesso aos dados bem estruturado com fluxo de endereços regulares. DLP irregular tem acesso menos estruturado aos dados sendo difícil de predizer seus fluxos. Seu controle de fluxo é menos estruturado e dependente dos dados. Fazendo uma combinação entres os dois tipos de DLPs, a programação é simplificada e a eficiência é melhorada, mesmo ele sendo um DLP de propósito geral.

Sobre os padrões o MIMD possui um grande número de núcleos de threads sendo uma ou mais tarefas mapeadas em cada um. Em vector-SIMD o controle de thread executa instruções de memória vetorial movendo os dados entre ela e os registadores. Já o subword-SIMD usa um amplo caminho de dados e registradores escalares, tanto com precisão simples quanto com dupla, para fornecer uma unidade vetorial. O SIMT é uma combinação da visualização lógica com o padrão vector-SIMD. O maior benefício do SIMT, é que ele fornece uma maneira simples para mapear um controle de fluxo complexo de dados dependentes com salto escalar. Um exemplo de processador multithreaded SIMT é o FERMI da nVidia. O acelerador Maven se baseia no padrão VT que é hibrido podendo controlar os overheads e executar instruções de memória vetorial com maior eficiência.

Foi desenvolvida uma biblioteca com parâmetros que podem ser combinados para construir o MIMD, vector-SIMD e VT tiles. Ela possui um conjunto de unidades funcionais com suporte a multiplicação de inteiros e divisão, com o padrão IEEE para adição de ponto flutuante, multiplicação, divisão e raiz quadrada de precisão simples. O núcleo escalar inteiro desenvolvido implementa um conjunto de instruções RISC, com instruções executadas em um pipeline de cinco estágios em ordem, mas com dois conjuntos de filas de requisição e resposta anexando o núcleo a memória do sistema com unidades funcionais de grande latência. O vector-SIMD ISA desenvolvido suporta controle do fluxo de dados dependentes usando uma máscara vetorial convencional. A biblioteca também inclui bloqueio e desbloqueio dos componentes de cache com um conjunto de parâmetros. Foi construído dois tipos de tiles: tiles multi núcleo que consiste em quatro MIMD ou núcleos vetoriais com via única, enquanto que tiles multi vias tem um único C conectado a unidade vetorial de quatro vias. Todos os tiles têm o mesmo número de unidades funcionais de latência longa. Uma série de otimizações microarquiteturais para melhorar o desempenho, como área e eficiência de energia do Maven foi explorada. Desenvolveram como mais uma melhoria um fusor de memória dinâmica para um VT de multi vias de unidades vetoriais. A fusor da memória procura através das vias por oportunidades para satisfazer múltiplos uT acessos da memória de um único acesso de 128-bit. Aceleradores de dados paralelos efetivos devem manipular os dois tipos de DLPs de maneira eficiente e manter a facilidade de programação.

De acordo com o exposto acima, é visto que microarquiteturas baseadas em vetores são mais eficientes, tanto em área quanto em energia se comparados as microarquiteturas escalares. Nota-se também que o Maven, sendo uma nova microarquitetura baseada no tradicional vector-SIMD fornece maior eficiência de energia e facilita a sua programação. Sua eficiência é melhorada com novas otimizações arquiteturais e ULAs distribuídas próximos dos bancos com arquivos de registro, podendo ser implementado em futuros processadores que vão acelerar o processamento de grande quantidade de dados.

Aluno: Anselmo Castelo Branco Ferreira RA: 023169
Artigo: A Novel Tag Access Scheme for Low Power L2 Cache
Autores: Hyunsun Park et al
Conferência: DATE 2011

No artigo é tratado o problema do consumo de energia em *caches* L2 em comparações de *tag*. É proposto um esquema de redução de energia composto de 3 passos: primeiramente, na chegada de um acesso à cada entrada da *cache*, um **filtro de bloom melhorado de *tags* parciais irá melhorar a previsão de um *cache miss***, reduzindo desta forma as comparações de *tag*. Esse filtro verifica quantos endereços há na entrada da *cache* onde o endereço seria mapeado por *hash*. Caso não haja endereço algum há um *per-way cache miss*. Caso haja apenas um endereço na entrada da *cache* (segundo os autores isso ocorre em 60% dos casos) a *tag* parcial do mesmo é comparada com a de entrada. Se as *tags* parciais não combinarem, há novamente um *per-way cache miss*.

Caso o filtro de *bloom* no passo anterior não previna o *cache miss*, continua-se a análise nas **linhas de *cache* quentes e frias**. Linhas de *cache* quentes ocorrem no período em que há mais acessos à *cache*, sendo o contrário para as linhas frias. A maioria dos *cache hits* ocorrem em linhas de *cache* quentes. Portanto, no segundo passo **busca-se a informação nas linhas de *cache* quentes**. Caso a informação não esteja lá, vamos ao terceiro passo que é **buscar a informação nas linhas de cache frias**. Em caso de *cache hits* em linhas de *cache* quentes, economiza-se comparações de *tag* com as linhas de *cache* frias. Caso a informação não esteja nem nas linhas de *cache* quentes nem nas frias temos um *cache miss*.

Para definir se uma linha de *cache* é quente ou fria equipam-se elas com um contador de vivacidade. Cada vez que uma linha de *cache* é referenciada, seu contador de vivacidade é inicializado em um valor TH. Esse contador é decrementado a cada N ($N = 16$ nos experimentos) referências à *cache*. Se o contador zera, sua correspondente linha de *cache* é considerada fria. O número de comparações de *tag* é em função de TH. Se ele puder ser setado ao valor que resulte em um número mínimo de comparações de *tag*, pode-se reduzir o consumo de energia. O relacionamento entre TH e o número de comparações de *tag* muda em diferentes programas e diferentes fases em um programa. Para resolver esse problema os autores determinam o TH ideal ao rastrear a dinâmica da mudança de comportamento de programas.

O consumo de energia no segundo e terceiro passo pode ser ainda mais reduzido ao **dividir a comparação de *tags* em 2 micro-passos**. Primeiro compara-se uma *tag* parcial do endereço de entrada com as *tags* parciais da *cache*. No caso de erro nas linhas de *cache* quentes procedemos as comparações parciais nas linhas de cache frias. No caso de *hit* parcial nas linhas de *cache* quentes ou frias, a parte remanescente da tag é comparada para dar o resultado completo da comparação. No caso de erro nas linhas de *cache* quentes procedemos as comparações parciais nas linhas de *cache* frias. No caso de erro nas linhas de *cache* frias, temos um *cache miss*. Esses 2 micro-passos de comparação de *tag* parcial e completa pode ser realizada em um ou 2 ciclos de clock, dependendo da frequência do *clock* considerada.

Nos experimentos foram executados 10 programas do SPEC2000 e SPEC2006 no simulador comercial da Tensilica LX2. As técnicas escolhidas para serem comparadas com a proposta foram os acessos seriais (nativo da *cache* L2) e outros métodos existentes de redução de comparações de *tag* (método de filtro de *bloom*, predição de caminho e decadência de cache). As três técnicas anteriores reduziram o consumo médio de energia na faixa de 11.98% a 16.55% se comparados ao acesso de *tag* seriais. Já a técnica proposta reduziu o consumo médio de energia em 22.45% se comparada ao mesmo acesso serial.

Título: A First Step Towards Automatic Application of Power Analysis Countermeasures. **Bibliografia**: A first step towards automatic application of power analysis countermeasures; Bayrak, Ali Galip and Regazzoni, Francesco and Brisk, Philip and Standaert, François-Xavier and Ienne, Paolo; DAC '11 **Autor**: Erick Nogueira do Nascimento. **RA**: 032483

O trabalho propõe uma técnica para aplicação automática de uma contra medidas em software contra ataques de análise de potência em implementações em software de algoritmos criptográficos. A técnica proposta tem aplicação geral, mas os autores tomaram como exemplo uma implementação em software do algoritmo de encriptação simétrico AES executando em um microcontrolador AVR 8-bit e utilizando *peephole transformation* como proteção.

A técnica proposta consiste em 3 etapas:

1. Análise de vazamento de informação. As entradas desta etapa são o código assembly do programa e a plataforma de hardware. A saída consiste no código assembly original com uma anotação, onde cada instrução tem associado um valor de sensibilidade. Esta etapa é executada tomando-se o código assembly, executando-o e coletando os traços de potência (*power traces*) em alta frequência, resumindo um valor único de potência para cada ciclo (adotou-se o maior valor medido dentro de um ciclo). Este passo inicial é repetido para diversos pares (texto claro, chave) escolhidos aleatoriamente, e os valores de potência são gravados. A seguir, utilizando as informações gravadas no passo anterior, é computado para cada ciclo o valor de sua sensibilidade através da métrica de teoria da informação adotada: informação mútua entre a informação vazada (potência) e a chave simétrica empregada pelo algoritmo. Por fim, é associado a cada ciclo a instrução assembly executada naquele ciclo. Obs: os autores não deixam claro se este último passo pode ser feito em um processador pipeline também, já que várias instruções estarão executando simultaneamente em estágios diferentes.
2. Identificação das instruções que serão transformadas. Esta etapa recebe como entrada a saída da etapa anterior. É escolhido um limiar para o valor de sensibilidade, de modo que as instruções acima deste limiar são consideradas sensíveis, e portanto deverão ser protegidas.
3. Transformação do código. O código é transformado segundo a técnica *peephole transformation*, a qual consiste em realizar modificações locais no código, introduzindo assim instruções adicionais no datapath, com o intuito de modificar o padrão dos traços de potência. Segundo os autores, para o microcontrolador AVR 8-bits utilizado, a inserção de instruções com operandos aleatórios antes e depois de cada instrução sensível (*random precharging*) é suficiente (e é a transformação adotada) para a proteção do código, pois o consumo de potência em tal plataforma depende do valor do dado que está fluindo pelos fios, portas ou unidades funcionais do datapath, logo, este consumo também será aleatorizado.

Experimento. As medidas foram realizadas usando um PC, uma placa com o microcontrolador, um osciloscópio e uma sonda diferencial. Todos os componentes foram ajustados para reduzir ao máximo o ruído eletrônico. Todas as medidas foram realizadas 25 vezes e a média aritmética foi utilizada.

Os resultados do experimento mostram, conforme previsto, que o código protegido de fato fez com que todas as instruções (as do código original e as novas) ficassem com o valor de sensibilidade abaixo do limiar definido, diminuindo assim a informação mútua trazida pelo vazamento de potência.

Entretanto, como os autores argumentam, a *peephole transformation* com *random precharging* não garante segurança perfeita, mas eleva o esforço (em número de traços de potência) requerido para um ataque bem sucedido. Para uma avaliação mais precisa do esforço que o atacante teria para conseguir realizar o ataque contra o código protegido, os autores realizaram ataques DPA (*differential power analysis*) baseados em correlação e concluíram que o número de traços de potência requeridos para um ataque bem sucedido é pelo menos 76 vezes maior do que a versão desprotegida.

No requisito desempenho, o código protegido apresentou um acréscimo de 126% no número de ciclos de execução em relação ao código desprotegido. Ainda assim, o código protegido levou apenas 64% do número de ciclos de clock de uma implementação que protege *todas* as instruções originais com o mesmo método de proteção.

Resumo do trabalho

Dataflow Execution of Sequential Imperative Programs on Multicore Architectures

de Gupta e Sohi, University of Wisconsin-Madison, MICRO-44 (2011)
Resumo escrito por Rafael Auler, RA 045840, 31/03/12

Este trabalho propõe um método para execução paralela de programas escritos em linguagens imperativas que não foram criados com paralelismo explícito (chamado de estaticamente paralelo). Portanto, trata-se de uma técnica de extração de paralelismo para um programa sequencial utilizando técnicas baseadas em análise de fluxo de dados. As técnicas tradicionais para extração de paralelismo com análise do fluxo de dados são restritas a programas escritos em linguagens funcionais, mas esta restrição é removida neste trabalho. O autor compara a sua técnica com a extração de paralelismo a nível de instruções: nesta, duas abordagens existem, aquela usada em arquiteturas VLIW, as estáticas, e a análise dinâmica do fluxo de dados usada, por exemplo, no método de Tomasulo. Ele argumenta que a análise dinâmica é mais proveitosa, pois requer compiladores menos elaborados e menos esforço do sistema operacional em resolver problemas que só aparecem em tempo de execução.

A granularidade do paralelismo extraído nesta proposta, entretanto, não é no nível de instruções, em que cada instrução independente executa em uma unidade funcional diferente a fim de aumentar o paralelismo. A ambição é usar informações de dependência de dados para executar funções que usam conjuntos de dados diferentes em diferentes núcleos de processamento (*cores*).

O modelo de execução proposto sugere uma infraestrutura de *hardware* para auxiliar a execução de programas escritos especificamente para este sistema. O usuário deve especificar quais funções são candidatas para serem executadas potencialmente em paralelo e, para tais funções, definir explicitamente seus conjuntos de objetos lidos e escritos. O sistema então implementa aquisição de *tokens* pra cada objeto lido e escrito de forma a serializar a execução em dependências RAW, WAR e WAW nos objetos e executar em paralelo funções independentes nestes quesitos.

Para testar a ideia, um protótipo completamente em software foi construído com o uso de *templates* em C++. O *template* "df_execute", por exemplo, sinaliza que uma função deve ser executada usando a infraestrutura que irá analisar suas dependências e executar outras funções em paralelo. O usuário também especifica barreiras com uma chamada a "df_end". Segundo os autores, impor esta descrição aos usuários é um ônus muito pequeno se comparado com a especificação de paralelização explícita.

Seis programas foram avaliados nesta infraestrutura: barneshut, blackscholes, bzip2, dedup, histogram e reverse_index, comparados contra a implementação usando pthreads (travas tradicionais). A média harmônica do *speedup* alcançado com a paralelização usando a infraestrutura proposta mostra que ela é apenas 13% pior do que o speedup alcançado com a programação paralela explícita utilizando travas e pthreads em um sistema Core i7, 21.7% pior em um AMD 8350 e 18.2% pior em um AMD 8356. Finalmente, os autores concluem argumentando que o futuro da paralelização não está na paralelização estática e explícita, mas em técnicas menos onerosas para os usuários, como a proposta neste trabalho.

Virtualizing Performance Asymmetric Multi-core Systems

***Referência**: Youngjin Kwon, Changdae Kim, Seungryoul Maeng, Jaehyuk Huh, Virtualizing Performance Asymmetric Multi-core Systems , Procedings of ISCA 2011, p. 45-56, June 4-8, 2011, San Jose, CA, USA. ACM 2011, ISBN 978-1-4503-0472-6*

Nome: Franz Pietz **RA: 076673** **MO401 – T1**

Resumo: Sistemas de multicore assimétricos consistem em núcleos heterogêneos, que funcionam em uma mesma ISA, mas têm diferentes características de processamento (desempenho, área e consumo de energia). Para o sistema assimétrico ser eficiente, deve-se analisar e encontrar características nos processos, para planejar como distribui-los nos diferentes tipos de núcleos, rápidos ou lentos, com o objetivo de maximizar o throughput. Esse sistema se mostra mais econômico, no geral, do que os sistemas homogêneos convencionais. Nos últimos anos, é comum a utilização de virtualização em aplicações como computação nas nuvens e centros de processamento. A virtualização apresenta um problema de uso eficiente de núcleos heterogêneos, por que esconde a assimetria física e os sistemas operacionais organizam os processos como se estivem em sistemas homogêneos. Uma solução para o problema é um gerenciador de máquinas virtual (hypervisor) modificado para exportar a assimetria. Com um hypervisor capaz de fazer agendamento de utilização (scheduling), a máquina virtual solicita uma combinação de núcleos rápidos e lentos para executar uma tarefa. Para utilizar todo o throughput potencial de um sistema assimétrico, o sistema operacional de uma máquina virtual não deve ter consciência do sistema assimétrico, cabendo somente ao hypervisor tomar as decisões de scheduling, ou seja, as máquinas virtuais devem utilizar os recursos sem consciência de outras máquinas e recursos disponíveis no sistema físico.

Para alcançar os benefícios da virtualização com núcleos assimétricos, o hypervisor deve cumprir as seguintes condições: i) maximizar o throughput do sistema, sem necessidade de auxílio do sistema operacional e aplicações, encontrando o melhor mapeamento entre CPUs virtuais e físicos; ii) deve cumprir os requerimentos de imparcialidade (fairness) de distribuição de núcleos rápidos entre as máquinas virtuais; iii) deve tratar da escalabilidade do sistema, caso sejam adicionados mais núcleos rápidos ao sistema. Um fator a se considerar para medir o desempenho é quanto efetivamente um CPU virtual usa cada tipo de núcleo. Se o número de instruções por clico (IPC) de uma aplicação é alta, ela tende a exibir maiores speedups ao usar núcleos rápidos em vez de lentos. Outro fator é o quanto a capacidade computacional é importante para determinada aplicação. Por exemplo, em testes de carga, o desempenho de núcleos rápidos é prejudicado quando há muitas operações de entrada e saída (I/O), pois o processador devem aguardar o termino das operações para continuar a processamento. No que diz respeito à imparcialidade, um scheduler totalmente imparcial garante que cada CPU virtual receba a mesma parcela de ciclos em núcleos rápidos e lentos, o que não traz melhoria na performance do sistema. Uma alternativa é utilizar um scheduler parcialmente imparcial, que divide uma porção dos ciclos de núcleos rápidos igualmente entre CPUs virtuais, deixando o restante reservado para aplicações conhecidamente eficientes em núcleos rápidos. O hypervisor pode medir a relação entre threads e utilização de ciclos de cada CPU virtual de duas maneiras: considerando a utilização de núcleos rápidos pelas máquinas virtuais, sendo que cada máquina recebe uma pontuação de utilização; ou considerando todos os núcleos rápidos, independente da máquina virtual que o está utilizando. No geral, o segundo método apresenta ganho de throughput, mas existem algumas restrições, com o comportamento heterogêneo ser estável para determinados intervalos de tempo. Baseado nas medidas obtidas, as regras de scheduling deve ser atualizadas em intervalos de tempo, para eliminar problemas de sobrecarga/ociosidade.

Para a os testes de implementação, foi feita uma modificação do scheduler do Xen, solução opensource para a virtualização que oferece suporte à scheduling para multicores homogêneos, para suportar scheduling de sistemas assimétricos. O sistema utilizado para os testes tinha 12 núcleos AMD Opteron 6168, sendo 4 configurados como núcleos rápidos e 8 como lentos. Para simular a performance de sistemas assimétricos em um sistema homogêneo, foi usado voltagem dinâmica e dimensionamento de frequências (dynamic voltage and frequency scaling – DVFS). Foram testadas aplicações com várias características, tais como carga intensiva de I/O(sysbench), servidores (apache), threads únicos (SPECCPU2006) e paralelas(pmake e PARSEC).

O artigo apresenta dados e figuras comparando os desempenhos entre cada uma das alternativas. Por exemplo, o scheduler nativo do Xen, que funciona com “crédito para CPUs”, teve resultados piores que os modelos propostos pelos autores. Outro resultado é que schedulers totalmente imparciais tem desempenho menor que os parcialmente imparciais, como esperado. No geral, foi mostrado que sem o devido suporte a assimetria, não é possível atingir o desempenho e escalabilidade potencial de núcleos rápidos quando ambos os tipos de núcleos são utilizados. Uma maneira de contornar é o hypervisor prevenir que núcleos rápidos fiquem ociosos antes de núcleos lentos, além de permitir que CPUs virtuais de uma mesma máquina virtual utilizem núcleos rápidos e lentos.

Título: A Shared-Variable-Based Synchronization Approach to Efficient Cache Coherence Simulation for Multi-Core Systems
Citação: Cheng-Yang Fu, Meng-Huan Wu, Ren-Song Tsay, "A shared-variable-based synchronization approach to efficient cache coherence simulation for multi-core systems", Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011, pp.1-6, 14-18 March 2011
Autor: Andrei Braga **RA:** 079713

Para manter a consistência de memória em uma arquitetura multi-core, é necessário empregar um sistema adequado para a manutenção de coerência das memórias cache. Parâmetros das memórias cache como tamanho e política de substituição são fundamentais para o desenvolvimento de um hardware em uma arquitetura multi-core. Os efeitos da manutenção de coerência das memórias cache é igualmente importante para o desempenho de um software de execução paralela. Assim, a simulação de um sistema de manutenção de coerência das memórias cache é crucial para ambas as frentes, hardware e software, em uma arquitetura multi-core.

Uma simulação de manutenção de coerência das memórias cache envolve vários simuladores, um para cada core. Para manter o tempo de simulação consistente para cada core, é necessária uma sincronização de tempo. As abordagens convencionais e intuitivas, de sincronizar a cada ciclo ou acesso de memória, propriciam uma simulação de baixo desempenho. À luz do conhecimento dos autores deste artigo, as abordagens existentes geram simulações com apenas uma de duas boas características: rapidez ou precisão. A principal contribuição deste artigo é, então, uma abordagem eficiente para a sincronização de tempo.

Para avaliar o desempenho da abordagem proposta, os autores realizam experimentos que a comparam com as quatro outras abordagens seguintes. A primeira (CB, do inglês cycle-based) é a abordagem que sincroniza cada simulador de um core em cada tempo de um ciclo. A simulação gerada por esta abordagem é garatidamente precisa. No entanto, a quantidade de sincronizações realizadas prejudicam a rapidez da simulação.

A segunda e a terceira abordagens são abordagens dirigidas a eventos. Abordagens dirigidas a eventos sincronizam os simuladores de cada core somente quando ocorrem determinados eventos. A segunda abordagem (MC, do inglês memory accesses with coherence actions) sincroniza, em vez de em cada tempo de um ciclo, no momento da execução do próximo evento de passagem de mensagem. A terceira abordagem (MA, do inglês memory accesses) faz a sincronização em cada ponto de acesso de memória. Na prática, estas abordagens podem gerar muitos eventos e as simulações obtidas também podem ter um desempenho ruim.

A quarta abordagem (SMC, do inglês shared variable access with coherence actions) faz uso da observação a seguir. Programas de execução paralela utilizam variáveis compartilhadas para se comunicar ou integarir um com o outro. Somente variáveis compartilhadas podem residir em mais de uma memória cache (em mais de um core) e apenas acessos a variáveis compartilhadas são importantes para a manutenção de coerência das memórias cache. Com esta observação, é possível sincronizar os simuladores de cada core apenas no acesso de variáveis compartilhadas e, assim, obter uma simulação com bom desempenho e precisão.

A abordagem proposta neste artigo (SMA, do inglês shared variable access) ainda se utiliza da seguinte análise. Para melhorar a eficiência da simulação produzida, os autores observam que o tratamento das *ações de coerência* em um simulador de um core pode ser adiado até se encontrar um evento de acesso de uma memória compartilhada. No entanto, é preciso processar as ações de coerência na ordem cronológica correta.

Os autores, então, realizam experimentos que permitem comparar as cinco abordagens citadas quanto a três quesitos: rapidez (milhões de instruções processadas por segundo), ônus causado pelas sincronizações e tempo total de simulação. Nos três quesitos, a abordagem SMA supera as outras. Quanto à rapidez, em especial, a abordagem SMA é de 18 a 44 vezes mais rápida que a abordagem CB, de 8 a 17 vezes mais rápida que abordagem MC e de 6 a 8 vezes mais rápida que a abordagem MA.

Em suma, os autores apresentam, comprovando com experimentos, uma abordagem eficiente para a sincronização de tempo em simulações de coerência das memórias cache em uma arquitetura multi-core. A abordagem faz uso das propriedades operacionais de manutenção de coerência das memórias cache e mantém eficientemente a sequência correta de simulação. Como um trabalho futuro, os autores propõem estender o esquema de sincronização proposto para a simulação de sistemas multitarefas.

Register Allocation for Simultaneous Reduction of Energy and Peak Temperature on Registers

Geraldo Magela Silva - RA 079740

1. Resumo

Em [Liu et al. 2011] os autores apresentam um *framework* para redução do consumo de energia e do acúmulo de calor baseado em técnicas de alocação de registradores. O *framework* ajuda a reduzir a transição de *bits* no barramento e impede o acúmulo de calor no acesso aos registradores. Os autores introduzem um pequeno *hardware*, **Rotator**, com uma simples regra de rotação para balancear objetivos de energia e minimizar acúmulos de calor; e um algoritmo de alocação de registradores voltado à redução do consumo de energia e acúmulo de calor.

A entrada para o rotator (veja Figura 1) é um bloco de instruções com cada instrução incluindo três variáveis: *Destination*, *Source1* e *Source2*. Quando decodificando, o rotator irá rotacionar cada indice de registro lógico de uma instrução $(i-1)\%k$ *bits* para a direita, onde i é $1 \leq i \leq n$ e n é o numero de instruções, que fará uso de k registradores. Internamente, o algoritmo de alocação de registradores utiliza o ***Live Range Successive Access Graph*** (LRSAG), um grafo direcionado em que cada nó é uma "faixa de vida" (*live range*) e cada aresta denota um conflito de relacionamento entre nós. O conflito implica que duas faixas de vida não podem ser atribuídas a um mesmo registrador físico. No passo seguinte, o algoritmo possui dois objetivos quanto à alocação de registradores para as variáveis consideradas:

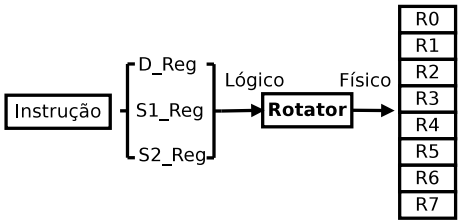


Figura 1. Decodificação de instrução com o Rotator

1. Redução do número de transições de *bits* (*bits transitions*), implicando na diminuição no consumo de energia. O autor utiliza o Hamming Distance para realizar o cálculo transições de *bits* entre dois registradores lógicos;
2. Densidade de acesso sobre os registradores é distribuído, evitando acúmulo de calor. O autor utiliza $Rcount(R_x)$ para indicar o número total de acessos para um registrador físico R_x.

Para avaliação o autor compara os resultados com duas outras propostas. A técnica de alocação chega a reduzir entre 7%-15% as transições de *bits* e até 6.8 graus *Celsius* em picos de teperatura nos registradores.

Referências

Liu, T., Orailoglu, A., Xue, C., and Li, M. (2011). Register allocation for simultaneous reduction of energy and peak temperature on registers. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6.

Power management of online data-intensive services [1]

Aluno: Marcelo Matsumoto
RA: 085937

Muito do acesso à internet depende de serviços *online* de intenso uso de dados (*Online Data-Intensive,* ***OLDI***). Serviços como grandes buscas de produtos e anúncios *online* são bons exemplos de OLDI. O grande problema é que, apesar da carga de uso de serviços OLDI variarem muito durante o dia, seu consumo de energia quase não se altera. Em servidores atuais, é comum consumirem de 50% à 60% de sua capacidade energética máxima quando estão sob 10% de sua capacidade de processamento. O objetivo do artigo é investigar o que, se existir, deve ser feito para tornar os serviços OLDI mais energicamente eficientes.

Estudo anteriores mostram que operações mais energicamente eficientes podem ser obtidos por servidores com pouca carga de uso através de modos de baixa energia (*idle low-power modes*). Essa técnica é comum em servidores que, em média, possuem baixa carga e um *dynamic range* estreito, características presente na maioria dos servidores. Para uma requisição OLDI ser executada com uma latência aceitável, o dado deve ser particionado em milhares de de *nodes* que trabalham em paralelo, dessa forma, a granularidade na qual o sistema deve ser desligado é no level de *cluster* não de *node*.

Em um *cluster* de busca, os nós folha são os mais numerosos, devido à alta distribuição de dados, e são responsáveis pela maioria do processamento e consumo de energia desse tipo de servidor, dessa forma, o estudo do artigo se focou nesses servidores. A latência nesses serviços (de busca) são de máxima importância, além disso, vale ressaltar que no nível de nó folha, a latência do serviço pode ser determinada pela latência de um único nó folha lento.

OS modos de baixo consumo podem ser divididos em três classes:

- *Idle Low-Power Mode*: Esse modo economiza energia em períodos de completa inatividade, nenhum processo pode ser executado enquanto o modo estiver ativo. O maior desafio desse modo é encontrar períodos de inatividade do servidor suficientemente longos. Este modo foi sub-dividido em dois modos:
  - ***Processor Idle Low-Power Mode*** Somente a CPU entra em modo de baixa energia, enquanto os outros componentes permanecem ativos. A vantagem sobre o *Full-system Idle Low-Power Mode* é a velocidade no tempo de transição de modo (na ordem de 100μs)
  - ***Full-system Idle Low-Power Mode*** Este modo tem o objetivo de obter economia de energia de todos os componentes do servidor, chega a alcançar reduções de energia de até 95%.
- ***Active Low-Power Mode***: Faz uma troca de desempenho por economia de energia, aumentando o tempo necessário par completar uma tarefa, mas ainda permite que o processamento continue. Esse modo obtém um ótimo resultado energético (energicamente proporcional à intensidade de uso) quando o servidor opera com mais de 50% de sua capacidade máxima. Para intensidades de uso baixas, o consumo de energia acaba sendo dominado por outros componentes (como discos, reguladores, *chipsets*, etc).

De uma maneira geral, concluiu-se que o modo *Full-system Idle Low-Power Mode* é o mais promissor dos três em relação à proporcionalidade energética com uma latência aceitável.

References

[1] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 319–330, New York, NY, USA, 2011. ACM.

RA:090056 Willian Roberto Montrezol

Design of High Speed Digital Circuits with E-TSPC Cell Library

João Navarro S. Jr.[1], Gustavo C. Martins[2]

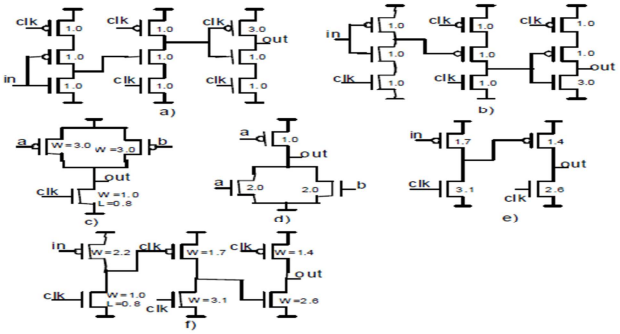
Este artigo se trata de uma aplicação com células extendidas únicas em uma única fase de tempo(E-TSPC) , em blocos de alta qualidade de circuitos digitais. O E-TSPC é uma técnica eficiente para aplicações em circuitos digitais. O resultado dessas simulações revelaram um aumento significativo de velocidade e aumento de desempenho cerca de 21% e 31% respectivamente, comparando a velocidade da implementação e redução de velocidade destes circuitos digitais, obteve-se 168% e 51%, respectivamente nas células implementadas.

A técnica E-TSPC

Esta técnica representa uma forma simples de representar blocos em seções separadas em circuitos digitais, complementando blocos estáticos e dinâmicos.

Implementação em células

Um bloco E-TSPC ordenado, implementa circuitos complexos. O design do circuito é similar ao desenvolvimento analógico dos próprios circuitos.



Observe que a implementação dos blocos de circuitos é facilmente entendido através das funções implementadas nas portas negativas (Inversores) nos circuitos. Esta aplicação E-TSPC trouxe uma alta velocidade nos blocos de transmissão dentro dos circuitos. Pode-se comparar a implementações pré-escalares de 32/33 em aplicações de aproximação lógica total. As implementações E-TSPC revelaram uma velocidade de performance de incremento e força de 21% e 31% respectivamente. Os resultados facilitaram a adição em blocos convencionais em circuitos lógicos digitais.

Referencias:

Yuan, J.-R and Svensson. C. 1989. High speed CMOS circuit technique. *IEEE J. Solid-State Circuits*, 24, 1 (Feb 1989), 62-70.

Navarro, J. and Van Noije, W. A. 1999. 1.6-GHz dual modulus prescaler using the Extended True-Single-Phase-Clock CMOS circuit tecnique (E-TSPC). *IEEE J. Solid-State Circuits*, 34, 1 (Jan. 1999), 97-102.

Chang, B., Park, J., and Kim, W. 1996. A 1.2 GHz CMOS dual-modulus prescaler using new dynamic D-type flip-flops. *IEEE J. Solid-State Circuits*, 31, 5 (May 1996), 749-752.

Miranda, F.P.H. et al. 2007. A 4.1 GHz Prescaler Using Double Data Throughput E-TSPC Structures. *In Proceeding of the SBCCI* (Rio de Janeiro, Brazil, Sept. 03- 06, 2007), 123-127.

Artigo: Architectural Support for Secure Virtualization under a Vulnerable Hypervisor [1]
Autor do Resumo: Thiago Augusto Lopes Genez (RA 100616)

Resumo

A computação em nuvem emergiu como um novo paradigma, onde hardware e software são entregues como serviços de utilidade geral e disponibilizados para os usuários através da Internet. Graças à camada de virtualização, construída sobre os recursos físicos, é possível otimizar estes recursos com o uso de máquinas virtuais – *Virtual Machine* (VM). Em outras palavras, a virtualização é a tecnologia que abstrai as características físicas do hardware e fornece recursos virtualizados para as aplicações de alto nível. Assim, uma máquina física é, de modo geral, logicamente dividida entre um monitor de máquina virtual – *Virtual Machine Monitor* (VMM) – e várias máquinas virtuais. Entretanto, cada máquina virtual é independente (e isoladas) uma das outras, podendo ter seu próprio sistema operacional, aplicativos e serviços. O VMM, também conhecido como *hypervisor*, é responsável pelo controle e virtualização dos recursos físicos compartilhados entre as VMs, como por exemplo, processadores, memória RAM, disco rígido e dispositivos de entrada e saída. É na vulnerabilidade do compartilhamento da memória RAM que os autores Jin *et al.* descreveram em [1].

Segundo Jin *et al.*, o *hypervisor* geralmente possui um controle total sobre os recursos de hardware, podendo acessar, sem nenhuma restrição, a memória RAM virtualizada de qualquer VM. Ao comprometer o *hypervisor*, um usuário malicioso pode acessar o conteúdo da memória das VMs usadas pelos usuários da nuvem. Isto é, a propriedade de *isolamento* das VMs é quebrada. Então, Jin *et al.* propõem um mecanismo baseado em hardware para proteger a memória RAM das VMs de acessos não autorizados. Denominado *hardware-assisted secure virtual machine* (H-SVM), este mecanismo independe da confiabilidade do *hypervisor*. Em outras palavras, os autores retiraram a responsabilidade de a alocação e gerenciamento da memória RAM física do *hypervisor* (software), e a transferiram para o hardware. No entanto, o mecanismo proposto pode ser vulnerável a ataques de hardware (não remotos), tais como leitura da memória RAM após desligá-la. Os autores, porém, assume que o provedor de nuvem é confiável e não tem a intenção de comprometer os hardwares.

Um dos principais requisitos importantes para o H-SVM é minimizar as modificações necessárias na arquitetura do hardware atual e no *hypervisor*, pois estas mudanças seriam muito complexas. Ou seja, o mecanismo H-SVM deve ser integrado nos projetos de processadores atuais sem aumentar significamente a complexidade do hardware, ou seja, o custo extra de hardware deve ser pequeno. Além disso, as interfaces com o *hypervisor* também deve ser simples. Resumidamente, o mecanismo H-SVM funciona da seguinte maneira: para quaisquer alterações na alocação de memória para uma VM, o *hypervisor* solicita um pedido ao H-SVM para atualizar a memória física. Assim, com intuito de impedir visualizações de memória RAM de qualquer VM pelo *hyperviosr* ou usuário malicioso, o H-SVM verifica se a solicitação pode violar o isolamento de memória entre VMs. Jin *et al.* realizaram experimentos do mecanismo H-SVM no *hypervisor Xen*, e os resultados mostraram que o *Xen* tornou-se seguro contra ataques na memória RAM das VMs e, além disso, o desempenho do *hypervisor* não foi prejudicado, ou seja, o *overhead* (custo extra) embutido no hardware foi mínimo e satisfatório.

Portanto, o mecanismo H-SVM melhora o isolamento de memória entre VMs através do bloqueio das alterações diretas na memória física pelo *hypervisor*, sendo gerenciado apenas pelo hardware. Assim, o provedor de nuvem pode oferecer um ambiente de execução seguro às VMs dos seus clientes.

Referências

[1] S. Jin, J. Ahn, S. Cha, and J. Huh, "Architectural support for secure virtualization under a vulnerable hypervisor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44 '11. New York, NY, USA: ACM, 2011, pp. 272–283. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155652

Identifying and Predicting Timing-Critical Instructions to Boost Timing Speculation

Xin, Jing and Joseph, Russ. *Identifying and predicting timing-critical instructions to boost timing speculation.* Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44 '11. pages 128–139. ACM, Porto Alegre, Brazil, 2011.

Walisson Ferreira Pereira
RA 115168

I. Objetivos

Os autores propuseram realizar simulações ao nível de porta lógica para explorar a erro temporal de localidade em instruções estáticas nos quais os padrões de uso de dados são sensíveis as falhas de atraso no trajeto. Outro objetivo foi a avaliação de mecanismos simples que predizem instruções prováveis de produzirem erros temporais no pipeline e fazer preparativos para a redução do custo de recuperação.

II. Procedimentos

Os experimentos realizados focaram no impacto que a predição de erro temporal e o preenchimento de erros têm num processador ambarcado de alto desempenho e baixo consumo energêtico. Um processador de 8 estágios foi simulado junto a um modelo detalhado a nível de porta lógica de unidades inteirais funcionais com erros temporais na execução do pipeline.

Os modelos implementados são suficientemente completos para virtualmente suportar todas as intruções inteiras em modo usuário sem as instruções MVI SIMD. Baseado-se em ciclos, os autores conduziram o simulador a nível de porta lógica com valores de programa que permitissem modelar o acoplamento entre o padrão controle com o fluxo de dados e os erros temporais na execução das unidades.

III. Resultados

Com o objetivo de medir a acurácia da medição, os autores exploraram as estratégias de predição de erros temporais *Last Value*, MDC (*Miss Distance Counter*) com 2, 3 e 4 bits, e Sat (*Saturating Counter*) com tamanho de 32 a 2048 entradas. A estratégia MRE (*Most Recent Entry*), foi avaliada de 2 até 128 entradas.

Os resultados da análise de sensibilidade das estratégicas em relação ao seu custo, o MDC com 4 bits obteve as melhores acurácias. A estratégia MRE obteve melhores números de acertos de predição (PPV, de *Positive Predictive Value*). Em contra partida, o MRE teve a pior sensibilidade e o MDC com 4 bits o pior PPV. Outra análise feita foi a penalidade sofrida pelas estratégias, e a partir desses resultados, os autores escolheram realizar uma análise mais detalhadas a estratégia MDC com 2 bits e 2048 entradas e o MRE de 32 entradas.

Os autores avaliaram a eficiência da predição de erros em aplicações independentes do ajuste de tensão. Nesse teste, a tensão (VDD) de operação era fixa em 70%, 80% e 90% da tensão nominal, obtendo redução média de 80% das penalidades causadas por erros temporais.

Os autores também testaram com a técnica de ajuste de tensão *Hill Climbing, Sampling* e *Fuzzy Controller* e comparado seu atraso energético com um base livre de erro, onde o sistema opera em máxima tensão.

Geralmente a previsão de erro e o preenchimento podem economizar mais energia trabalhando acima da tensão crítica e mascarando os erros de predição de tempo. Dentre outros *benchmarks* realizados pelos autores, ambos os preditores (MRE e MDC de 2 bits) reduziram o atraso energético em aproximadamente 21% no compilador gcc.

IV. Discursões e conclusões

A fronteira da tensão crítica é uma limitação que a especulação temporal enfrenta, pois impede sobrescalar o processador além de ponto específico. O trabalho dos autores podem ser aplicados no projeto com técnicas de otimização para reduzir o impacto da fronteira crítica de tensão e impulsionar a especulação temporal.

Os modelos analíticos limitam a análise a pequenas porções do processador. Por isso, os autores trabalharam com simulação, e mesmo assim, isso ainda era o gargalo em sua pesquisa. Eles esperam que outros pesquisadores devenvolvam novas metodologias e modelos para acelerar a simulação.

A variação de parâmetros pode alcançar alguns ganhos ao se explorar diferentes sensibilidades de entrada, porém mesmo com a variação de parâmetros, os erros temporais de localidade ainda são proiminentes, pois em geral um pequeno número de instruções é responsáveis pela maioria dos erros temporais e há uma tendência de instruções estáticas executarem na mesma unidade funcional.

Nas conclusões finais, os autores concluem que as instruções estáticas são mais previsíveis e possuem taxas de erros estáveis (erro de temporização por localidade); e que a maioria dos erros temporais ocorrem devido a um pequeno número de instruções estáticas que tendem a terem taxas de erros estáticas.

The Impact of Memory Subsystem Resource Sharing on Datacenter Applications

TANG, L.; MARS, J.; VACHHARAJANI, N.; HUNDT, R.; SOFFA, M. L. The impact of memory subsystem resource sharing on datacenter applications. In: International Symposium on Computer Architecture, 38, San Jose, California, USA, 2011.

Eliana Alves Moreira – RA 120437

O artigo apresenta estudos realizados para analisar os impactos do compartilhamento de recursos do subsistema de memória entre diferentes aplicações (três sensíveis à latência e duas *batch*) do *datacenter* da Google, envolvendo vários tipos de cenário de mapeamento *thread-to-core* (TCC) em uma plataforma primária *dual-socket Intel Xeon* E5345 (Clovertown). Em plataformas *multisocket multicore*, os núcleos de processamento podem ou não compartilhar recursos de memória, incluindo o último nível de cache (LLC) e largura de banda de memória. No entanto, os autores afirmam que, conforme já citado em trabalhos anteriores, é necessário reduzir a interferência causada no desempenho por aplicações de menor prioridade sobre aplicações de alta prioridade. Segundo os autores, *datacenters* modernos tem atribuído *threads* a núcleos de uma forma *ad-hoc*. O aumento de aplicações de *datacenter* traz à tona a importância de se compreender como é realizada a interação destas aplicações com a arquitetura de compartilhamento de recursos de memória da máquina utilizada. A caracterização das interações pode fornecer informações para novos projetos de arquiteturas para este tipo de cargas de trabalho, pois, na escala do *datacenter*, uma melhoria no desempenho de 1% em aplicações-chave pode resultar em milhões de dólares economizados.

Assim, os autores realizaram experimentos nas aplicações sensíveis à latência para analisar a variabilidade do desempenho quando a aplicação é executada sozinha ou conjuntamente com uma das aplicações *batch*. Observou-se que, se, ao executar sozinha, uma aplicação tem melhor desempenho usando LLC´s separadas e FSB´s separados, depois da transição para compartilhamento LLC existe um aumento de faltas LLC. Isto indica que a disputa de LLC ocorre entre as *threads*. O consumo de largura de banda do barramento é consistente com as faltas de LLC e tendências de desempenho, pois o desempenho destas aplicações é pior no cenário de mapeamento em que o FSB é compartilhado, devido à sua disputa. Quando executada simultaneamente com outras aplicações, a aplicação tem suas preferências alteradas dependendo da aplicação co-executante. Na aplicação sendo executada sozinha em que o desempenho é melhor ao compartilhar 2 LLC´s e um FSB, depois da transição para compartilhamento LLC existe uma diminuição nas faltas LLC. Isto indica que o compartilhamento de cache é construtivo e que *threads* estão compartilhando dados que cabem na LLC. Esta aplicação, ao executar simultaneamente com outras aplicações, mantém sua preferência pelo compartilhamento, apesar de existir oscilação de desempenho. Os experimentos mostraram também que as aplicações sensíveis à latência com média/alta demanda do barramento preferem compartilhar FSB com aplicações *batch* que tem menor demanda de barramento, para não haver degradação no desempenho. Observou-se, ainda, que aplicações com maiores taxas de falta LLC sofrem mais com a disputa de cache.

Os autores propuseram um algoritmo heurístico para prever um mapeamento *thread-to-core*. A ideia básica é caracterizar as aplicações co-alocadas, priorizando-se o desempenho de aplicações sensíveis à latência, e ainda evitar o compartilhamento de seus potenciais gargalos, maximizando o benefício. Para avaliar o algoritmo comparou-se o melhor mapeamento previsto com o melhor mapeamento baseado na verdade. A abordagem prediz corretamente o melhor mapeamento em 4 dos 6 pares de co-execução, e nos outros 2 casos a diferença é inferior a 2% . Porém, a abordagem tem limitações: características devem ser coletadas para cada aplicação e um novo algoritmo precisa ser gerado, porque cada arquitetura tem diferentes topologias e pontos de compartilhamentos.

Os autores construíram um mapeador *Thread-to-core* adaptativo (AToM), que usa uma heurística de competição para adaptativamente pesquisar pela ideal indicação *thread-to-core* para um dado conjunto de *threads*. Durante a fase de aprendizagem, AToM empiricamente coloca vários mapeamentos *thread-to-core* uns contra os outros para saber qual o mapeamento executa melhor. Durante a fase de execução, o vencedor do mapeamento *thread-to-core* é executado por um período fixo ou adaptado de tempo antes que outra competição seja realizada. AToM mostrou-se eficaz, alcançando um desempenho perto do ótimo. Em cada tipo de mapeamento, AToM supera o caso médio em até 22%, e tem significativamente um melhor desempenho do que as atribuições de pior caso.

Os resultados mostraram que o impacto no desempenho do compartilhamento de recursos de memória para as aplicações sensíveis à latência é significativo e que cada aplicação prefere diferentes configurações de compartilhamento. Ainda, oscilações do desempenho das aplicações entre o seu melhor e pior mapeamento TTC pode ser significante quando executam simultaneamente com outras aplicações. Estas características podem ser utilizadas para construir algoritmos heurísticos para realizar mapeamentos *thread-to-core*, bem como uma abordagem adaptativa. Os autores concluíram que esta traz melhores resultados do que a primeira em relação à melhor utilização do compartilhamento dos recursos disponíveis.

Scalable Hybrid Verification for Embedded Software

Behrend, J.; Lettnin, D.; Heckeler, P.; Ruf, J.; Kropf, T.; Rosenstiel, W. "*Scalable Hybrid Verification for Embedded Software*". Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011 , vol., no., pp.1-6, 14-18 March 2011

Aluno: Junior John Fabian Arteaga
RA: 123542

Atualmente, o software embarcado está presente em nossa vida diária. Este tipo de software é usado para controlar as funções do carro, produtos de telecomunicação, robôs, aparelhos elétricos, dispositivos médicos, etc. O software embarcado desempenha um papel fundamental com funcionalidades como a redução das emissões de combustíveis, a melhoria da segurança entre outras.

O poder de processamento cada vez maior, a diminuição do consumo de energia, a diminuição dos preços e a minimizacão do tamanho dos microcontroladores e microprocessadores permitem desenvolver mais funcionalidades do hardware no software embarcado; o qual termina com milhões de linhas de código e um rápido aumento da complexidade.

A maioria dos erros fatais ocorre nos softwares dependentes do hardware e não nos software de aplicação. Alguns exemplos de diferentes erros em software dependente de hardware (e.g., drivers de baixo nível) são os erros de temporização, estouro da memória, erros de inconsistência. É por isso que, a verificação de software embarcado tornou-se um tema importante nos últimos anos.

Este trabalho apresenta uma nova abordagem hibrida e escalável para a verificação de propriedades temporais em softwares embarcados com dependências de hardware. O método de verificação proposto consiste de três principais fases: Preprocessamento, “Bottom-up” e “Top-down”. No processamento, o código em C é processado mudando o código para 3-AC (three-address code) que é normalmente usado por compiladores. Seguidamente, geram um modelo “SystemC” do software embarcado. Depois na etapa de Bottom-up, verificam as principais funções do software e marcam as funções segundo um modelo verificador presente no estado da arte. Y finalmente na etapa Top-down, analisam as funções marcadas na fase anterior. Os autores apresentam dois tipos de experimentos diferentes nos quais mostram que a abordagem de verificação proposta tem bons resultados e é facilmente utilizado para softwares embarcados complexos dependentes do hardware. Além disso, a abordagem é mais escalável em comparação com métodos propostos no estado da arte.

RAID 6 Hardware Acceleration.

Gilroy, M.; Irvine, J.; Atkinson, R. RAID 6 Hardware Acceleration. ACM TECS, vol.10, no. 4, artigo 43. Novembro, 2011.

Aluno: 123789 – Gerson Nunho Carriel

O artigo apresenta um trabalho sobre RAID 6 e uso do algoritmo de Reed-Salomon. Em seu início apresenta a definição de Patterson de que o uso de RAID garante mais confiabilidade ao armazenamento de dados, devido ao uso combinado de unidades de disco, oferecendo melhor performance e tolerância a falhas. Outro ponto importante para justificar o trabalho é a probabilidade de mais um disco apresentar defeito cresce à medida que existem mais discos.

Os conceitos sobre o RAID 5 são apresentados para indicar que o RAID 6 trabalha sobre os mesmos conceitos, em que grava blocos de dados em faixa transversalmente nos discos e grava também os blocos de paridade em todo o array.

O RAID 6, objeto de estudo do artigo, oferece suporte para a recuperação de problemas em dois discos. Para isso, é necessário dois discos de checksum, que são aplicados ao conceito de ter as faixas usadas no RAID 5, que também é utilizado para garantir performance. Os autores deixam claro que o RAID 6 não está padronizado, embora exista uso Comercial.

Para compreensão do trabalho, é importante um entendimento mínimo do algoritmo utilizado: o Reed-Salomon(RS).

O RS é uma classe de codificação de correção de erros que permite a recuperação de qualquer **n** erros de um sistema com **m**+**n** entradas de dados. Este algoritmo possui uma capacidade de correção ótima à medida que precisa de **n** locais de armazenamento para recuperar **n** falhas. O controle da paridade feito pela codificação de RS e que são necessários dois discos redundantes e duas gravações por faixa, nesse algoritmo, é utilizada a Teoria de Campo de Galois, em que um campo de Galois **q** é denotado por GF(**q**).

No artigo, são mostrados os trabalhos de desenvolvimento para RAID 6. A primeira descrita é a implementação do Controlador RAID 6 de quatros discos que foi projetado para utilizar um mínimo de recursos de hardware. O codec do hardware foi implementado com RS codec, operando em GF(4) e considerando as seguintes regras: os discos de dados e de checksum estão em locais fixos, ambos os dados de entrada passam pelo codec simultaneamente e 2 bits por entrada são codificados ou decodificados simultaneamente. A primeira codificação é feita por um XOR de dois dados de um bit e o segundo valor é feito por meio de uma multiplicação por um valor fixo do GF. Caso um ou os dois discos de checagem sejam apagados ou tenham perda de dados, eles podem ser regenerados, passando novamente pelo codificador Reed-Salomon.

Nesse primeiro momento, foi colocar uma interface PCI ao projeto, além disso, uma memória foi adicionada como buffer antes e depois da codificação e decodificação.

A arquitetura com PCI foi analisada quanto aos benefícios de performance. O PCI foi implementado com um Kernel linux 6.2 modificado para atender a demanda. Todas as comparações de performance foram feitas em relação a outro kernel 6.2 não alterado. Para o teste de throughput foi utilizado o Bonnie++, para evitar o cache foi usado um arquivo de 4G.

Foi realizado o teste de Confiabilidade para geração e Recuperação do Array. Para este teste, foi utilizado o mdadm do linux. A verificação da integridade dos dados foi feita comparando um arquivo escrito com o original e checando diferenças. Testes exaustivos foram realizados e os dados do array foram reconstruídos com sucesso.

A performance de Leitura /Gravação, tanto do hardware quanto do software de RAID 6, foram testados com Bonnie++. E ocorreu uma melhora clara quando usado o acelerador de Hardware em comparação ao uso apenas da solução de software no teste com falha dupla de discos.

O Acelerador de Hardware RAID 6 produzia taxa de throughput, próximo ao encontrado a solução de software sob operação normal, oferecendo um ligeiro ganho na falha de um disco e grande benefício durante problemas com dois disco. O uso de CPU ficou entre 3% e 4% pelo Driver do Hardware, enquanto o software utilizava acima de 33% durante operação normal e acima de 50% durante falha em dois discos.

Os autores trabalharam também com uma implementação de 15 discos em GF(16), utilizando entradas de 4 bits de largura e para arrays de até 255 drives foi utilizado GF(256) com 8 bits de comprimento.

Os aceleradores baseados em GF16 e GF(256) foram feitos para serem mais flexíveis. O Acelerador RAID 6 Baseado em GF(16) conseguiu um consumo de 20% de CPU em relação a outro projeto já publicado. Os testes foram realizados com cinco discos, que foram particionados para atender o projeto.

Para conseguir verificações rápidas dos codec GF(16) e GF(256), um barramento Avalon foi utilizado. Essa arquitetura permitiu gerar métricas de performance.

Ao final os autores acreditam que o desenvolvimento dos três controladores RAID 6 baseados em Reed-Salomon resultaram em solução ótima na primeira implementação, e que a segunda, para 15 discos mostrou que conseguiu realizar a tarefa, em menor taxa de clock, usando apenas 20% dos recursos fornecidos em relação a uma melhor implementação de outro projeto. O terceiro projeto mostrou suporte para os controladores Reed-Salomon RAID, na qual a escalabilidade do codec de GF(256) tem se mostrado ser opção ideal dos fabricantes. Além disso, apresentam que da investigação feita foi demonstrado que reduções nos recursos de hardware podem ser feitas com adoção adequada de algoritmos direcionados a aplicações específicas e os testes mostraram que o Throughput foi mantido para os três projetos.

A Reconfigurable Cache Memory with Heterogeneous Banks

Benitez, D. ; Moure, J.C. ; Rexachs, D. ; Luque, E.; DATE 2010

Aluno: John Edward Esquiagola Aranda RA: 124208

O tamanho ótimo da memória cache num SoC *(System on chip)* pode ser diferente dependendo dos programas a serem executados nele. Num mesmo ponto, a redução dos erros de cache conseguidos quando aumentarmos o tamanho da memoria, tem como conseqüência um aumento no tempo de latência de cache, afetando o desempenho da mesma. A melhor configuração de cache depende das características da aplicação e das restrições do projeto (desempenho, potencia e área). Devido ao fato de que não existem caches que satisfaçam os requerimentos de todas as aplicações, uma abordagem promissora é adicionar capacidade de reconfiguração na memoria cache.

Este artigo em questão apresenta o *Amorphous Cache (AC)*, uma memoria cache L2 reconfigurável desenvolvida para melhorar o desempenho assim como reduzir o consumo de potencia. AC é composta de sub-caches heterogéneas que podem ser desligadas seletivamente, reduzido o tamanho total da cache e/ou sua associatividade, o qual reduz o tempo de acesso e o consumo de potencia. Uma técnica de reconfiguração de cache chamada *BBV&NDP* também é proposta, essa técnica usa *Basic Block Vectors (BBVs)* para identificar as fases de execução de um programa dinâmicamente. *Amorphous Cache*, é um circuito reconfigurável especializado que pode ser implementado na tecnologia CMOS e pode ser integrado nos processadores convencionais. O termo *Amorphous* indica que o range de configurações da cache, caminhos críticos e consumo de potencia não é homogéneo, eles dependem da carga de trabalho.

A arquitetura consiste em 18 configurações de *AC*, indo desde 64KB ate 2MB, e com associatividade em grupos de 4, 8 ou 16 vias. O circuito *AC* é organizado num conjunto de blocos heterogéneos chamados *Sub-Caches*, os quais são conectados nas entradas e saídas da cache através de uma inovadora rede de interconexão programável dentro do chip. As sub-caches que nao estiverem conectadas são desligadas. Existem registradores de configuração que permitem modificar o tamanho e a associatividade do *AC*, Quando uma configuração pequena é requerida, só a sub-cache com o tamanho requerido é ativado e usado, e o restante é desligado usando 1-bit num registrador de configuração de *Power-Supply*.

Quando um código de programa é executado, ele vai por diferentes estagios. O processo de reconfiguração começa limpando a *sub-cache* activa ou todas as *sub-caches*. Só depois, a sub-cache requerida ou todas as *sub-caches* são activadas. Tendo como alvo detectar dinâmicamente as mudanças nos estagios do programa e executar cada estagio com a máxima eficiencia se propõe um novo algoritmo de hardware called *BBV&NDP*. Este algoritmo resume a execução do programa coletando informação sobre quais e quantas vezes os blocos básicos de instrução *(BB)* são executados durante um intervalo de instrução, a partir deste ponto um novo vector chamado *´BBV reduzido´* é criado a partir dos 300 *BBs* mais executados.
Esse vectores BBV reduzidos são armazenados na *Pattern CAM Memory*, eles sao chamados de *Pattern BBVs* e representam os diferentes estagios do programa.Usa-se o algoritmo NDP para determinar qual Pattern BBV será designado para a configuração da cache.

De acordo com os autores, a heterogeneidade do *Amorphous Cache* faz dela uma proposta diferente se compararmos com o estado da arte atual dos caches reconfiguráveis. O novo predictor baseado em BBV proposto selecciona a melhor configuração de cache se comparado com outras técnicas adaptativas. Os resultados demostram que o *AC* reduz a latência de acesso, potência dinâmica e potencia estática na

media de 55,8%, 46,5% e 49,3% quando comparado com uma memória cache não adaptativa de 2MB.

Fractal Coherence: Scalably Verifiable Cache Coherence

Meng Zhang, Alvin R. Lebeck e Daniel J. Sorin

(43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), dec. 2010)

Resumo por Carla Négri Lintzmayer – RA 134042

Um protocolo de coerência de cache deve ser bem projetado e inspecionado para que não haja falhas no sistema que for utilizá-lo. Protocolos atuais permitem requisições pendentes e operações concorrentes, levando a vários estados de transição e um comportamento não-determinístico, tornando difícil a verificação do projeto para garantir sua confiabilidade. O uso de simulações não verifica todos os estados possíveis; a verificação formal automatizada é possível e pode superar o problema da simulação, mas possui ainda alguns problemas como o "problema da explosão de estado", que é causado pelo fato do método ser exaustivo e "explodir" o espaço de estados exponencialmente com o aumento do número de *cores*.

O artigo propõe uma metodologia de projeto que permite que os protocolos sejam verificados com o uso de ferramentas formais automatizadas existentes. A base está na teoria dos fractais, pelo fato de um fractal ser uma forma que pode ser dividida em partes onde cada parte é uma cópia reduzida do todo. Os autores criaram uma classe chamada *Coerência Fractal* de protocolos de referência que podem ser verificados por indução: se um sistema grande B tem comportamento fractal e um sistema menor A (que pode ter sua coerência comprovada) é cópia reduzida de B, então pode-se provar a coerência de B com base na de A.

A coerência de cache de um sistema fractal é verificada formalmente garantindo dois passos[1]. Os autores apresentam, além da teoria, a implementação de um protocolo de coerência fractal específico denominado *TreeFractal* para mostrar que a metodologia de projeto fractal é viável e explicam como utilizar duas ferramentas automatizadas de verificação para realizar esses dois passos:

1. **O sistema mínimo deve ser coerente em cache.** Um sistema mínimo inclui todos os tipos diferentes de componentes usados em sistemas maiores. Sua modelagem deve capturar o comportamento do protocolo de coerência de cache e garantir que ele fique dentro da capacidade de verificação das ferramentas atuais para evitar o problema da explosão de estado.

   O sistema mínimo implementado tem formato de árvore binária e possui dois nós básicos e uma Tag de Topo. Cada nó básico tem um *core*, duas caches privadas L1 e L2, uma porção de memória compartilhada e um controlador de coerência. A Tag de Topo é o "pai" dos dois nós básicos. Os autores utilizaram o verificador chamado *Murphi* para verificar a coerência de cache desse sistema mínimo. *Murphi* levou três horas para verificar o sistema e foram explorados 12.031.400 estados.

2. **O sistema inteiro deve possuir comportamento fractal.** Isso garante que a coerência será mantida quando o sistema for escalado. Como o sistema é construído por iteração, basta verificar a equivalência entre os níveis 1 e 2, mostrando que o "mundo exterior" não vê diferenças entre eles.

   O sistema mínimo implementado pode ser escalado para qualquer sistema com N nós adicionando o que os autores chamaram de Tags Internas entre a Tag do Topo e os nós básicos. Essa estrutura é uma árvore binária. As Tags são interfaces que dão suporte ao comportamento fractal mantendo cópias das tags das caches e dos estados de coerência dos seus filhos. No sistema como um todo, os nós básicos são as folhas da árvore, os nós internos são as Tags Internas e a raiz é a Tag do Topo.

   O comportamento fractal do sistema implementado foi verificado utilizando um verificador chamado *Bisimulator*, que retornou `true` indicando que os sistemas comparados eram de fato equivalentes.

Por fim, foram realizados experimentos para mostrar que *TreeFractal* não apresenta uma degradação de desempenho. Ele foi comparado com os protocolos *Snooping* e *Directory*. Os três utilizaram os mesmos parâmetros arquiteturais e vários *benchmarks* foram testados. De modo geral, *TreeFractal* tem um desempenho comparável ao *Snooping* e ao *Directory*, superando-os em alguns *benchmarks* em determinadas situações (por exemplo, com 2 ou 4 *cores*, *TreeFractal* se saiu melhor em praticamente todos os *benchmarks*). O mais importante é que o desempenho de *TreeFractal* é comparável com os protocolos utilizados e ainda garante a corretude do sistema por meio de métodos formais de verificação.

[1] É apresentada uma prova por indução do porque esses dois passos são suficientes.

A New IP Lookup Cache for High Performance IP Routers

Resumido por Diego Rodrigo Hachmann – RA134047

Guangdeng Liao, Heeyeol Yu, and Laxmi Bhuyan. 2010. A new IP lookup cache for high performance IP routers. In *Proceedings of the 47th Design Automation Conference* (DAC '10). ACM, New York, NY, USA, 338-343.

A demanda por alto desempenho nos roteadores da internet continua a aumentar. Uma tarefa crítica executada nestes roteadores é o *IP Lookup,* que consiste em procurar em uma tabela de dados, geralmente *off-chip*, alguma informação baseada em um endereço IP. Um exemplo bem conhecido é o encaminhamento de pacotes (*packet forwarding*), onde o roteador busca em uma tabela, baseado no IP de destino, a linha de saída para o próximo *hop*.

Para melhorar o desempenho, uma *cache on-chip* tem sido acrescentada nos já diversos esquemas de *IP Lookup* existentes (*TCAM*, *trie-based*, *hash-based*, etc.). Contudo, as *caches IP* atuais são baseadas nas *caches L1/L2* das *CPUs* e não empregam características do tráfico de informações da internet que possibilitam obter um melhor desempenho.

Baseado em várias observações, os autores deste trabalho propuseram uma nova arquitetura de *cache IP on-chip* baseada em dois eixos principais: indexação e política de substituição.

Quanto ao modo de indexação da cache, os autores estudaram o comportamento de várias funções *hash* bem conhecidas e perceberam que uma *2-Hash* universal (onde um dado pode estar em dois conjuntos de cache mapeados através de duas funções *hash* distintas) apresentam um melhor desempenho quando comparado a outros modelos. Para as funções *hash* foram escolhidas duas funções pertencentes as classe H3, que são fáceis de implementar tanto em hardware quanto em software. Para diminuir a complexidade do hardware, apenas 16 bits dos 32 do endereço de IP, entre o 8º e 24°, foram escolhidos como entrada das funções *hash*. Estes bits foram selecionados por apresentarem maior variação nos traces estudados, dentre os 32 bits, e, portanto, serem mais relevantes. Testes indicaram que o *hash* com os 16 bits escolhidos obtém o mesmo desempenho que o *hash* com 32 bits, mas com uma complexidade menor do *hardware*.

A nova política de substituição de *cache* foi motivada principalmente por outro estudo onde foi observado que o fluxo da internet segue uma distribuição *Zipf* (o site mais acessado é duas vezes mais acessado que o segundo, três vezes mais que o terceiro e assim por diante). Além disso, de dois *IP traces* estudados, 14% de um e 21% de outro, apresentaram fluxos não populares, onde um fluxo (IP) não será reusado depois de ser lido. Pensando nisto, os autores propuseram uma política progressiva de substituição de cache que descarta os fluxos não populares (que são acessados esporadicamente) em favor dos fluxos populares.

Para isto, cada linha da *cache* contém um bit que informa se o IP foi usado novamente depois de ser lido. Como um dado pode ser inserido em dois conjuntos da cache (há duas funções *hash*), ele sempre é colocado no conjunto que possui mais fluxos não populares. Dentro do conjunto com quatro linhas selecionado, ele é inserido sempre na última linha. Quando um fluxo é lido na cache ele troca de posição com a linha acima, tornando-se "mais popular".

Para avaliar os resultados, os autores desenvolveram um simulador de cache e testaram o novo modelo com dois traces (FUNET e PMA). A taxa de *miss cache* ficou em torno de 0.7%. Isto levou a um aumento de velocidade de aproximadamente 1.5 vezes, alcançando um *throughput* de 2Tpbs, e dissipando, em média, 1.7 vezes menos potência quando comparado a outros esquemas de cache IP.

Evolution of Thread-Level Parallelism in Desktop Applications

Blake, G., Dreslinski, R.G., Mudge, T., Flautner, K., 2010. Evolution of thread-level parallelism in desktop applications. In: Proceedings of the 37th annual international symposium on Computer architecture. ISCA '10. ACM, New York, NY, USA, pp. 302-313.
URL http://doi.acm.org/10.1145/1815961.1816000

Eduardo Theodoro
RA: 134049
eduardotheodoro@gmail.com

1. INTRODUÇÃO

Como o alcance do limite da frequência, devido a restrições de temperatura, e o alcance do paralelismo a nível de instruções foram atingidos, atualmente a estratégia de empresas produtoras de microprocessadores como Intel e AMD se voltaram ao aumento do número de processadores em um único chip. Desde então, chips com multiprocessadores se tornaram o emblema destas duas líderes de desenvolvimento, atualmente utilizando acima de 8 cores por *die*. A expectativa assim se torna de que desenvolvedores de software escrevam softwares com concorrência suficiente para tirar proveito deste hardware adicional. O artigo estudado visa determinar como o aumento do número de cores tem sido acompanhado com o nível de paralelismo de softwares desenvolvidos para desktop.

2. PROCEDIMENTOS

Como forma de avaliação, foi utilizada a métrica TLP *(Thread Level Parallelism)*, que caracteriza a quantidade média de concorrência exibida por um programa durante sua execução quando pelo menos 1 core está ativo. TLP é calculada pelo somatório de $c_i's$ que correspondem a frações de tempo que exatamente $i = 0, 1, \ldots, n$ (onde n corresponde ao número de threads sendo executadas na máquina) threads são executadas concorrentemente. Este número é então dividido pela fração de tempo ocioso $1 - c_0$. Utilizando TLP é possível identificar o número mínimo de processadores necessários para suportar um conjunto de aplicações em paralelo.
Uma variedade de experimentos utilizando benchmarks de jogos 3D *(Call of Duty 4, EA Crysis)*, editores de imagens *(Adobe Photoshop CS4, Autodesk Maya 3D)*, players multimídia *(iTunes 9, QuickTime)*, editores de vídeo *(CyberLink PowerDirector, Handbrake 0.9)*, nevagadores de internet *(Mozilla Firefox 3.5, Apple's Safari 4.0)* entre outros programas interativos, foram realizados nos sistemas operacionais *Apple's OS X Snow Leopard* e *Microsoft Windows 7*, utilizando os processadores Intel Xeon e Intel Atom.

3. RESULTADOS E CONCLUSÃO

Em média, o índice TLP médio das aplicações obtido para o Intel Xeon foi de cerca de 2.15. Isto significa que uma máquina multiprocessador com cerca de 2-3 cores é suficiente para a maioria das aplicações de uso cotidiano. Podemos notar contudo que aplicações relacionadas a área de edição de vídeo exibem uma performace muito superior, exibindo uma média de 7.4, enquanto aplicações relacionadas a jogos possuem TLP de apenas 1.6. Todavia, devido ao Intel Xeon possuir uma performace em *single thread* alta, tarefas individuais poderiam estar sendo completadas rápidas o suficiente para não rodarem concorrentemente. Assim, os mesmos aplicativos foram executados utilizando o processador Intel Atom, que possui uma performace *single thread* mais baixa, podendo levar assim a um aumento do TLP. Apesar de notado uma redução do tempo ocioso do sistema, o índice TLP também se manteve na faixa de 2 no Intel Atom.
Através dos resultados, podemos observar que em alguma porção de tempo existe execução concorrente, contudo, em muitos casos essa execução concorrente é em porcentagem pequena. Na prática, os desenvolvedores de software estão escrevendo códigos com um grande número de threads na maioria das aplicações desktop, porém estas threads raramente são executadas em paralelo ou o trabalho distribuído entre elas é muito desbalanceado. Este desbalanceamento leva a efeitos da *lei de Amdahl*, resultando assim no baixo TLP visto nos experimentos computacionais.
A maioria das fabricantes atuais tem adotado multicores como uma tentativa de aumentar a performace. Contudo, em comparação com um estudo realizado no ano 2000 por Flautner et al., que utilizou-se de aplicações similares, foi possível observar apenas uma pequena melhoria no TLP para aproximadamente 2 nos anos atuais, levando assim a acreditar que a adição de cores em CPU's pode não ser a resposta para o aumento da performace no espaço de aplicações desktop.

Trabalho 1 - Resumo do Artigo: Automatic Workload Generation for System-Level Exploration Based on Modified GCC Compiler.

Citação completa: Jari Kreku, Kari Tiensyrja, and Geert Vanmeerbeeck. 2010. Automatic workload generation for system-level exploration based on modified GCC compiler. In *Proceedings of the Conference on Design, Automation and Test in Europe* (DATE '10). European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 369-374.

Objetivo e importância

Este artigo apresenta uma técnica baseada na modificação de um compilador (**GCC**) que permite gerar automaticamente **modelos de "workload"** para a simulação do desempenho de arquiteturas candidatas ao nível do sistema (particularmente em sistemas embebidos). No contexto atual em que há sistemas complexos que executam seus programas tanto sequencialmente quanto concorrentemente, isto é importante porque se precisam de avaliações ou simulações do desempenho rápidas e eficientes que permitam evitar erros críticos do desenho. Porém, neste trabalho a facilidade de geração e "debugging" teve mais prioridade que o tempo de simulação do desempenho, assim melhorar isto é um trabalho futuro para o desenvolvimento de ABSINTH.

Modelos em "System-level exploration", ABSOLUT e ABSINTH

Os "workloads" são abstrações das aplicações ("benchmarks") que se executam num determinado computador ou arquitetura que servem para medir seu desempenho. Esta abstração é a diferença do enfoque deste trabalho com outros, porque aqui o modelo de "workload" imita realmente as estruturas de controle das aplicações, mas os dados de nível mais baixo são presentados como "traces", e também a plataforma de execução é modelada no nível de transação e não de instruções. É importante ressaltar que o artigo se descreve no contexto de outros modelos e enfoques desenvolvidos previamente (ABSOLUT).

O enfoque **ABSOLUT** abstrai os modelos de aplicações como modelos de "workload", estes são assignados aos modelos de execução na plataforma (que são abstraídos como modelos de capacidade) e simulados num nível de transação para obter resultados do desempenho. Os **modelos de "workload"** tem uma estrutura hierárquica, dividindo-se em aplicações e controle comum de concorrência (implementado em C++ e baseado em SystemC), e subsequentemente em: processos, funções, bloques básicos e "branches", e operações primitivas (as operações primitivas podem ser "read", "write" ou "execute").

O **modelo da plataforma** compreende a descrição de HW e SW para a simulação. Este tem três camadas: de componente (inclui processamento, armazenamento e comunicação), de subsistema (instancia componentes e seus conexões) e de arquitetura da plataforma (conecta os modelos de "workload" com as plataformas mediante interfaces). Na fase de alocação se assigna a cada entidade do modelo de "workload" um recurso da plataforma (que poder ser um processador). As saídas da simulação são o tempo de execução e outras medidas do desempenho (como utilização de componentes, operações de I/O, etc.) que permitem decidir se a capacidade de execução da plataforma deve ser melhorada ou a aplicação deve utiliza-la mais eficientemente.

ABSINTH é a ferramenta baseada no enfoque ABSOLUT que permite gerar modelos "workload" automaticamente tendo como entrada um código fonte. ABSINTH foi implementado como extensão de GCC e tem dois passos: o primeiro é **pass_absinth_cf**, o qual constrói a camada de funções do modelo de "workload" (fluxo de controle entre blocos básicos); e o segundo passo é **pass_absinth_bbs**, o qual atravessa o RTL (código intermédio de GCC) para extrair as operações primitivas ("read", "write" e "execute").

Para gerar o modelo de "workload" precisa-se de três fases: primeiramente, o código fonte deve ser compilado com "profiling" (isto ajuda a conhecer as probabilidades de "branches"); depois, o código binário gerado deve ser executado com um conjunto de dados adequado; e finalmente, o código fonte deve ser compilado novamente, mas além de "profiling" com ABSINTH habilitado. Um modelo de "workload" é gerado por cada função do código fonte, assim os modelos podem conter chamados a outras funções de "workload", então é preciso de um **ABSINTH manager** o qual detecta dependências e modifica os arquivos de tal maneira que sejam compilados corretamente para a simulação.

Casos de exemplo

O ABSINTH foi usado para criar modelos de "workload" do codificador de vídeo x264, e duas versões (paralela e sequencial) de um codificador JPEG. O modelo de plataforma consistiu em quatro nodos ARM conectados por roteadores, em que cada um tinha um ARM9 CPU, memoria SRAM, um barramento compartilhado e uma interface a os outros nodos.

Seguiram-se os três passos descritos anteriormente para gerar os modelos de "workload" de x264, obtendo-se um normal e outro com blocos básicos simples. Cinco simulações foram executadas com ambos os conjuntos, as velocidades das simulações foram 1 /70 e 1 /17 do tempo real respectivamente, com o segundo conjunto o tempo foi cerca de 10 segundos menos devido à menor utilização de SRAM. O segundo caso de exemplo foi gerar modelos de "workload" para JPEG seguindo também os três passos; simularam-se quatro conjuntos de modelos de "workload", o primeiro foi sequencial e os outros três foram versões paralelas que foram criadas usando MPA (uma ferramenta para mapear eficientemente aplicações em plataformas "multicore"). O tempo de execução do sequencial foi 70 ms e dos paralelos 55, 53 e 38 ms.

Releasing Efficient Beta Cores to Market Early
Sangeetha Sudhakrishnan, Rigo Dicochea, and Jose Renau
International Symposium on Computer Architecture – ISCA'11

O trabalho realizado tem, como fundamentações básicas, os intuitos de criar formas de agilizar o processo de criação e testes de novos processadores, além de que seja consumido menos tempo e energia em seus testes, considerando que os novos cores possam ter certa quantidade de bugs, os quais não inviabilizariam totalmente o seu lançamento no mercado.

Há duas maneiras básicas de realizar testes em Beta Cores. A primeira delas ("Always Re-Execute Systems") é utilizar-se de um Checker Core ligado ao Beta Core. Este Checker Core funciona 100% do tempo verificando todo o processamento que o Beta Core realiza. Com isto, bugs, mesmo que não conhecidos são encontrados, ao custo de uma grande complexidade do Checker Core e alto consumo de energia. A outra forma ("A Priori Bug Systems"), utiliza-se de uma tabela onde, a priori, sabe-se quais saídas do Beta Core podem causar erros e, caso alguma saída seja igual à da tabela, um Checker Core reexecuta o processamento para correções no Beta Core. Nesta maneira, há o problema de erros novos não serem previamente conhecidos e passarem despercebidos.

A sugestão dos autores é de se redesenhar ambos os processos e uni-los. Somente sua união não traria qualquer benefício prático, portanto, o conceito do trabalho é que, utilizando-se de uma tabela, como no Priori Bug Systems, que seja dinamicamente alterada pelo Checker Core caso haja inconsistência entre ele e o Beta Core. Além disso, códigos já executados gerariam uma assinatura que, colocada na tabela, eliminaria a sua reexecução. Com isso, há a possibilidade de se utilizar um Checker Core mais simples. Nos estudos, utilizou-se um Checker Core simples, com execução em ordem, para se checar um Beta Core complexo e com execução fora de ordem. Além disso, um Checker Core com metade do clock do Beta Core é suficiente para a tarefa.

Após os testes, concluíram que é uma maneira válida de se disponibilizar ao mercado novos processadores, desde que sejam aceitos bugs frequentes e já conhecidos, junto dos bugs pouco frequentes e desconhecidos. Na forma proposta, o ganho real esperado é a redução em checagens repetitivas, a redução da complexidade e do consumo de energia do Checker Core, redução do Overhead. Também, há a melhoria nos acertos da tabela de boas assinaturas e a redução na atividade do Checker Core e em sua influência no desempenho geral do Beta Core.

ASF: AMD64 Extension for Lock-free Data Structures and Transactional Memory

Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie e Dan Grossman

(43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Dec. 2010)

Resumo por Maycon Sambinelli – RA 134071

Processadores *multi-core* se popularizaram e hoje estão por toda parte. Porém nem todo o potencial dessa arquitetura é utilizado devido às dificuldades da programação paralela. Para tentar resolver esse problema surgiu a Memória Transacional (MT), um mecanismo onde o programador precisa apenas definir *blocos* que serão executados atomicamente, mas que necessita de algum tipo de aceleração por *hardware*.

Os autores do artigo projetaram uma extensão para a arquitetura AMD64 chamada *Advanced Synchronization Facility* (ASF), que provê regiões especulativas que executam acessos especulativos atomicamente. Essas regiões são utilizadas para construir primitivas atômicas multi-palavra para a programação *lock-free* e para prover a primeira geração de aceleração de *hardware* para memória transacional.

O ASF conta com a adição de 5 novas instruções: 1) *SPECULATE*, que inicia uma região especulativa; 2) *LOCK MOV*, que move dados entre os registradores e a memória; 3) *RELEASE*, que é uma dica para remover o isolamento de um *load* especulativo de um determinado endereço de memória; 4) *ABORT*, que reverte as alterações de uma região especulativa; e 5) *COMMIT*, que termina uma região especulativa. As regiões especulativas são criadas utilizando-se as filas *load/store* e a cache L1 como *buffers* para dados especulativos. Para sinalizar dados especulativos, cada entrada das filas recebe 1 bit, e cada linha da cache recebe 2 bits. O protocolo *Coherent HyperTransport* é utilizado para detectar conflitos de acesso à memória especulativa/não especulativa comparando mensagens que chegam do protocolo de coerência de cache contra esses bits. Se um conflito é detectado, a região especulativa é abortada, descartando-se os dados especulativos e limpados esses bits. Se a instrução *COMMIT* é atingida, os dados especulativos dos *buffers* são efetivados limpando-se o grupo de bits, e enviando os dados da fila de *store* para a cache L1.

Para avaliar o desempenho, os autores utilizaram um simulador interno da AMD e escreveram 4 versões do ASF: 1) ASF-LC, que usa tanto a cache L1 quanto a unidade LS como *buffer* para dados especulativos; 2) ASF-C8, que usa apenas a cache L1 *8-way* associativa como *buffer*; 3) ASF-C4, que usa uma cache L1 *4-way* associativa; e 4) ASF-AS, que é igual ao ASF-LC, porém não suporta o uso da instrução *LOCK MOV* e trata todos os acessos a regiões especulativas como acessos especulativos. Uma região especulativa pode esgotar os *buffers* de dados especulativos; nesse caso ocorre uma exceção de *overflow*. Nos testes, o manipulador de exceções usa um mecanismo de *software*, que trata o problema da seguinte maneira: 1) espera todos os outros núcleos saírem de suas regiões especulativas; 2) impede que outros núcleos entrem em regiões especulativas; 3) executa uma versão não especulativa da região especulativa que sofreu *overflow*; e 4) permite que os outros núcleos voltem a entrar em regiões especulativas.

O ASF foi testado utilizando programação *lock-free* e transacional. O teste utilizando programação *lock-free* teve por intenção avaliar como o ASF lida com um conjunto de dados randômicos. Uma tabela *hash* que trata os conflitos com uma lista ligada teve as chaves inseridas randomicamente por diversas *threads* até o primeiro *overflow* de capacidade. Os resultados mostraram que o ASF-LC pode conter 12,3x mais blocos do que ASF-C8 e 63,9x mais blocos do que o ASF-C4. Os testes transacionais foram utilizados para avaliar o *speedup*. Foram utilizados 3 grupos de aplicações do *benchmark* STRAMP, categorizados de acordo com suas escalabilidades. O primeiro grupo consistia das aplicações *bayes*, *labyrinth* e *yada*. Os resultados mostraram que o *ASF-AS* escala pobremente em comparação ao ASF-LC/C8/C4, devido ao fato dele não suportar a instrução *LOCK MOV*, que habilita o uso seletivo do acesso especulativo para reduzir o *footprint* da memória especulativa. O segundo grupo era formado pelas aplicações *intruder*, *kmenans* e *SSCA2*; obtiveram um resultado muito parecido, pois todas possuíam apenas transações de vida curta, e, desta forma, elas não tiraram vantagem do uso das instruções *LOCK MOVs*, nem de um conjunto associativo maior ou do uso das filas como um *buffer* especulativo. O terceiro grupo contém os problemas *Genome* e *Vacation*, que escalam melhor com ASF-LC e ASF-C8. Analisando esses resultados, os autores concluíram que o uso da unidade LS e L1 como *buffers* especulativos contribui muito para a robustez do sistema, e que o uso seletivo de acesso especulativo melhora sua escalabilidade.

Resumo: Idempotent Processor Architecture

DE KRUIJF, M.; SANKARALINGAM, K. Idempotent Processor Architecture. MICRO-44 '11.

Autor do resumo: Rodolfo Guilherme Wottrich – 134077

Este trabalho utiliza o conceito matemático de *idempotência* para propor uma nova arquitetura de processadores, a *Arquitetura Idempotente de Processadores*. Idempotência diz respeito a operações matemáticas que podem ser executadas infinitas vezes e, ainda assim, apresentam o mesmo resultado de que se fossem executadas apenas uma vez (como a multiplicação por 1). Processadores idempotentes executam sequências idempotentes de instruções, permitindo a reprodução precisa do estado do processador através de simples re-execução da região de código idempotente, tornando desnecessário todo o suporte de hardware para recuperação de misspeculation. Isso simplifica o design de processadores in-order e reduz significativamente o consumo de energia, inclusive com ganho de desempenho com relação a processadores in-order comuns.

Dentro de uma região idempotente, tanto a execução quanto o retirement de instruções podem ser realizados out-of-order. Isso permite a utilização imediata de dados produzidos por instruções de baixa latência sem a necessidade de complexos esquemas de staging e bypassing. Além disso, simplifica a implementação de suporte preciso de exceções. Por fim, permite o retirement out-of-order com respeito a operações com latência imprevisível, como loads.

O artigo descreve como identificar e construir regiões de código idempotente. Intuitivamente, uma região idempotente não pode sobrescrever suas entradas, para que possa ser re-executada do início em qualquer momento de sua execução. Isso se traduz, basicamente, em não conter dependências de dados do tipo WAR sem haver antes uma RAW. A esse tipo de dependência, os autores dão o nome de *clobber antidependence*. Há *clobber antidependences* que pertencem à semântica do programa e outras artificiais, geradas por técnicas de register renaming do compilador. Essas últimas podem ser evitadas por compiladores específicos para gerar código com mais idempotência, gerando regiões idempotentes maiores e garantindo melhores resultados com a arquitetura idempotente. Os autores criaram um compilador com essas características usando LLVM e submeteram os códigos a benchmarks SPEC 2006, PARSEC e Parboil e demonstraram que é possível gerar código com muito mais idempotência na maioria dos casos, para arquiteturas ARM e x86.

Processadores idempotentes eliminam a necessidade de muitos elementos do hardware convencional, então os autores ainda propõem a adição de um slice data buffer (SDB) de quatro entradas para uma leve melhora no desempenho. O SDB mantém instruções dependentes de um load miss, desbloqueando o pipeline e permitindo o issue de instruções próximas com respeito às que estão no SDB.

O trabalho ainda realiza uma avaliação experimental da arquitetura, através das mesmas suítes de benchmark, comparando através de simulação combinações de processador in-order comum, processador idempotente simples, processador idempotente com a otimização do SDB, processador out-of-order comum e código original e idempotente. Os resultados demonstram que a arquitetura idempotente otimizada obtém ganho médio de 4.4% no desempenho sobre arquitetura in-order comum, além de trazer redução na complexidade do hardware e redução significativa no consumo de energia (apesar de o artigo não realizar avaliação experimental desse aspecto). Processadores out-of-order ainda executam muito mais eficientemente, mas o objetivo não é exceder a eficiência de arquiteturas desse tipo.

Por fim, os autores ainda discutem alguns aspectos importantes relativos à sua proposta de arquitetura idempotente.

Com este trabalho, conclui-se que repensar alguns dos princípios fundamentais da organização de hardware através de um ponto de vista de restrições de consumo de energia pode melhorar significativamente a eficiência de processadores.

FabScalar: Composing Syntetizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template

Niket Kumar Choudhary, Salil V. Wadhavkar, Tanmay A. Shah, Hiran Mayukh, Jayneel Gandhi, Brandon H. Dwiel, Sandeep Navada, Hashem Hashemi Najaf-abadi, and Eric Rotenberg. FabScalar: Composing Syntetizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template. In *38th International Symposium on Computer Architecture (ISCA 2011)*, pages 11-22, San Jose, CA, USA, 2011.

Aluno: Rogério Alves Cardoso (RA 134078)

Resumo

Um processador multicore heterogêneo (ou processador assimétrico multicore) com um único conjunto de instruções (ISA), fornece múltiplos tipos de núcleos superescalares diferenciados que, podem agilizar a execução de diversos programas e fases do programa o que simplifica a execução de programas sequenciais, paralelos e cargas de trabalho multiprogramadas, explorando a diversidade em torno das aplicações. Os tipos de núcleos podem diferir em relação à largura da busca/issue, profundidade do pipeline, agendamento das instruções (*in-order* ou *out-of-order*), tamanho das unidades envolvida na exposição do paralelismo em nível de instrução, unidades funcionais e tamanho das cache ou dos preditores.

Houve um crescente trabalho conjunto em relação a esse paradigma, no entanto, nenhuma pesquisa anterior havia focado no "Calcanhar de Aquiles" desse modelo: o esforço de projeto e verificação do projeto é multiplicado pela quantidade de tipos de núcleos diferentes. Sendo que esse fator limita a diversidade microarquitetural que pode ser implementada na prática.

Este trabalho foca em processadores superescalares em uma forma canônica. A forma canônica está no nível de lógica dos estágios (canônicos) do pipeline: *fetch, decode, rename, dispatch, issue, etc,* tornando possível o design rápido de núcleos que diferem em três dimensões superescalares: largura do superescalar, profundidade do pipeline e o tamanho das estruturas para extração de instruções em nível de paralelismo (ILP).

Para essa ideia foi desenvolvida um conjunto de ferramentas chamado FabScalar, para, automaticamente, compor modelos sintetizáveis em nível de transação de registradores (RTL) de processadores com tipos de núcleos arbitrários utilizando os chamados modelos superescalares canônicos.

O conjunto de ferramentas consiste na definição de processadores superescalares canônicos. No modelo proposto todos os processadores superescalares possuem a mesma estrutura canônica, cada um possui um conjunto de estágios canônicos e a mesma interface entre eles. Uma biblioteca chamada *Canonical Pipeline Stage Library (CPSL)* contendo diversos designs sintetizáveis em RTL para cada estágio canônico do pipeline. Um processador superescalar pode ser composto selecionando um design para cada estagio canônico e então colocá-los juntos de forma a construir um pipeline com um conjunto completo de estágios canônicos. Esta composição é automatizada a despeito das interfaces invariantes dos estágios do pipeline canônico e do confinamento da diversidade arquitetural do pipeline. Portanto a diversidade microarquitetural é focada em torno das dimensões chaves e ambas definem a arquitetura superescalar e diferencia os processadores individualmente.

Em adição, considerando que nos processadores superescalares as RAM de alto porte e as CAM prevalecem e possuem um impacto significante, foi desenvolvido o FabMem uma ferramenta para a geração automatica de designs fisicos de *Access Memory (RAM)* e *Content Adressable Memory (CAM)* de multiplas portas.

Além disso o FabScalar provê um ambiente de cosimulação com um simulador funcional, com precisão de ciclo, escrito em C++ que roda de forma concorrente com a simulação verilog do processador superescalar sintetizado. Validações e experimentos foram realizados em torno de três frentes para validar a qualidade do modelo RTL gerado pela ferramenta: funcional e performance (instruções por ciclo), validação, validação temporal (tempo de ciclo), e confirmação de adequação com o fluxo ASIC padrão.

A ferramenta FabScalar torna possível a concepção de um chip, com muitos tipos diferentes de núcleos superescalares. Embora ainda esteja em estágio acadêmico, levando em consideração que o artigo ataca o problema do esforço de projeto e verificação, a intenção é o uso da ferramenta FabScalar para o projeto, verificação, validação e fabricação de de chips composto de superescalar de microarquitetural diversificada, além de ser útil para a pesquisa de novas arquiteturas de computadores de uma forma geral. Outra aplicação promissora do FabScalar é a aceleração da simulação de processadores superescalares baseados em FPGA apresentado nesse mesmo artigo.

Uma arquitetura para auto-organização em sistemas ubíquos[1]

Julián Esteban Gutiérrez Posada. RA 134097

Neste artigo é apresentada uma interessante mas especulativa arquitetura para a auto-organização em sistemas ubíquos. Neste contexto, deve-se entender por um sistema ubíquo um sistema composto por muitos dispositivos eletrônicos independentes e invisíveis ao usuário, que colaboram entre si para desempenhar uma função determinada. Atualmente, estes sistemas necessitam de um dispositivo central que "organize" todos os demais dispositivos, em prol de uma atividade ou função. Por outro lado, esta nova arquitetura busca que os diferentes dispositivos do sistema se auto-organizem de uma forma automática sem a intervenção do dispositivo central.

Embora os experimentos realizados no artigo sejam realizados com dispositivos em uma rede com fio, a arquitetura foi pensada e criada para ambientes móveis, onde os dispositivos entram e saem do alcance físico do sistema ubíquo; alcance que depende principalmente da tecnologia utilizada para criar a rede sem fio. Independentemente da tecnologia de comunicação que seja usada, esta deve suportar mensagens em *Broadcast,* pois toda a arquitetura baseia seu funcionamento numa série de mensagens que são enviados a todos os dispositivos e só respondem aqueles que devem fazê-lo segundo seja o caso.

Existe uma série de pontos importantes e abertos à pesquisa: o primeiro está relacionado com a qualidade do serviço neste tipo de sistemas, já que ao estarem disponíveis dispositivos de qualidades diferentes, o resultado final para o usuário poderia ser afetado. O segundo está relacionado com mecanismos para administrar, compartilhar e controlar o acesso aos diferentes recursos disponíveis. O terceiro com a formalização de procedimentos e o quarto está relacionado com adicionar a cada dispositivo a capacidade de aprender diversos conhecimentos de outros dispositivos.

A capacidade de aprendizagem, nesta arquitetura, é devida a que cada dispositivo possui uma base de conhecimento que pode usar para determinar quais serviços pode oferecer e de quais precisa de outros, para cumprir plenamente suas funções quando fizer parte de um dispositivo virtual. Este último é o nome que recebe o subconjunto de dispositivos do sistema que se cria e se destrói de maneira dinâmica para desempenhar certas solicitações mais complexas feitas por parte do usuário, as quais não poderiam ser realizadas, de maneira individual, por nenhum dispositivo do sistema.

O conteúdo dessa base de conhecimento pode ser armazenado no momento da criação de cada dispositivo ou quando é utilizado. Dependendo da complexidade do mesmo, este teria que armazenar mais ou menos informação em sua base de conhecimento. Este conhecimento é extraído de ontologias existentes, embora o procedimento envolvido nesta extração ainda não tenha sido padronizado.

Por outro lado, os dispositivos permanecem no sistema num estado inicial (inativo) e logo da solicitação do usuário a um dispositivo especial (*Request Center*), ele envia mensagens em *Broadcast* à rede de dispositivos procurando serviços. Os dispositivos que podem ajudar passam a outros estados de configuração e execução de serviços ("*Setup"* e "*Execution and Control"*). Uma vez que a solicitação foi executada, é enviada uma mensagem que destrói o dispositivo virtual e todos os dispositivos voltam ao estado inativo.

Embora seja verdade que, ao menos em teoria, esta arquitetura é menos susceptível a falhas pela capacidade de adicionar e retirar dispositivos cujas funções podem ser fornecidas por outros, considero que o próximo passo na pesquisa é aprofundar nos problemas associados com a sincronização de dispositivos, crucial para muitos problemas de concorrência.

---

1 Aly. A. Syed, Johan Lukkien, and Roxana Frunza. 2010. An architecture for self-organization in pervasive systems. In *Proceedings of the Conference on Design, Automation and Test in Europe* (DATE '10). European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 1548-1553.

- Título do Artigo: **A Robust Mechanism for Adaptive Scheduling of Multimedia Applications**.
- Referência: CUCINOTTA, T. et allii. A robust mechanism for adaptive scheduling of multimedia applications. **TECS**, New York, v. 10, n. 3, art. 46, abr. 2011.
- Autor do resumo: Marcelo Correia – RA 134163.

Este artigo enfoca um mecanismo de agendamento de tarefas e trabalhos da CPU necessários à execução de aplicações multimídia. Como exemplo apropriado a este modelo, pode-se pensar em uma aplicação de videoconferência onde cada quadro de vídeo é captado a uma taxa fixa, encodado, transmitido, decodificado e exibido – considerando-se, portanto, a disponibilidade e maximização da utilização de processamento para possíveis múltiplas tarefas, envolvendo a alocação de largura de banda e seu respectivo processamento requisitados por cada uma dessas tarefas.

As tarefas são escalonadas simultaneamente, dentro do conceito de Arquitetura em *Pipeline*, atribuindo e reclamando largura de banda de forma progressiva e superposta, *real time*, maximizando o uso do poder computacional para atingir um nível controlado e elevado de performance *versus* tempo de execução.

Analisando o trabalho centrando o ponto de vista na Arquitetura de processamento envolvida no mecanismo apresentado, os autores propõem uma técnica de agendamento adaptativo de reserva de largura de banda para multiprogramar o tempo de execução de tarefas dinâmicas para organizar o processamento da CPU, identificando e rastreando os requisitos demandados, assumindo que múltiplas tarefas multimídia em tempo real podem estar concorrentemente ativas em simultâneo.

Os autores admitem como objeto um conjunto de tarefas de tempo real $\{T_1\}$ rodando em uma CPU compartilhada. Com a ressalva de que mesmo que a discussão refira-se a uma única CPU, a abordagem é igualmente aplicável ao caso de sistemas de arquitetura *muticore*, cujos processos são agendados por uma política de escalonamento particionado em tempo real, onde tarefas são vinculadas a cada CPU.

Evidenciou-se no estudo que, em um modelo dinâmico de agendamento de erro, “uma reserva de CPU pode ser considerada como sistema dinâmico de eventos discretos, cuja evolução é observada ao término de cada trabalho $\{J_{i,j}\}$ (trabalho, ou *job*, da tarefa $\{T_{1j}\}$), adotando-se como controle de entrada a atribuição de um período mínimo garantido de tempo $\{Q_{i,j}\}$.

O esquema de controle e análise envolvido no processo atua gerenciando: objetivos de controle formal; uma lei de controle implementada pelo controlador de tarefas, a fim de se garantir a disponibilidade mínima necessária. Um supervisor global é empregado na: maximização de compressão de dados em largura de banda disponível; para implementar um mecanismo de solicitação de banda e seu processamento em tempo real, utilizando um algoritmo desenvolvido na pesquisa, chamado SHRUB; para combinar feedback e solicitação; e na sincronização da latência correta entre a requisição de reservas adaptativas e o mecanismo de solicitação.

Sintetizando os resultados obtidos, a mais interessante contribuição dada pelo trabalho é mostrar uma sinergia benéfica entre dois mecanismos diferentes: as reservas adaptativas e um supervisor global, que implementa uma política de correção ativa ao ser combinado com o algoritmo de recuperação de recursos SHRUB, gerenciando as condições de sobrecarga e solicitando largura de banda ociosa, o que confere robustez ao processamento de tarefas multimídia pela CPU.

Prefetch-Aware Shared-Resource Management for Multi-Core Systems

Eiman Ebrahimi; Chang Joo Lee; Onur Mutlu; Yale N. Patt

ISCA'11, June 4–8, 2011, San Jose, California, USA.

INTRODUÇÃO

Circuitos integrados com múltiplos processadores internos (CMP) compartilham entre estes processadores grande parte dos subsistemas de memória como: último nível de *cache*, controladores de memória e memórias externas. Quando diferentes aplicativos estão sendo executados por diferentes *cores*, acessos de um *core* à memória podem interferir nos acessos de outros, causando atraso na execução dos aplicativos. Pesquisas recentes propõem técnicas para permitir alto desempenho e distribuição adequada (*fairness*) na utilização destes recursos, no entanto nenhuma delas leva em consideração acessos a memória antecipados (*prefetches*). *Prefetch* é uma técnica de acessar a memória antecipadamente de forma especulativa trazendo dados que o processador deverá utilizar no futuro, buscando minimizar o impacto do tempo de latência destes acessos no tempo de execução dos aplicativos.

OBJETIVO

O artigo analisa três destas técnicas de gerenciamento de recursos compartilhados, demonstrando que não trazem melhoria de desempenho e *fairness* aos CMPs quando aplicadas juntamente com a técnica de *prefetch*,

O artigo propõe então modificações aos algoritmos destas técnicas e demonstra que se adotadas é possível obter tanto a melhoria de desempenho e *fairness* no gerenciamento dos recursos bem como os benefícios da técnica de *prefetch*.

ANÁLISE TEÓRICA

Duas das técnicas analisadas são de agendamento e priorização de acesso à memória (*memory scheduling*).

A técnica PARBS (*paralelism-aware batch scheduling*) realiza esta priorização ao criar pacotes de acessos, incluindo um máximo de demandas por core/banco de memória no pacote, garantido que todos os *cores* sejam atendidos, e criando um novo pacote somente quando todos os acessos do pacote atual tiverem sido realizados.

A técnica NFQ (*network fair queuing*) realiza a priorização determinando o tempo virtual de término de cada demanda de acordo com a largura de banda alocada para cada tarefa e dá maior prioridade as demandas que tiverem o tempo virtual de término mais cedo.

Estas técnicas porém não fazem distinção entre acesso reais e *prefetches* na priorização dos acessos.

O artigo introduz então o conceito de *prefetches* úteis (*accurate prefetches*) que são acessos especulativos que posteriormente são efetivamente utilizados e de *prefetches* não úteis os quais acabam sendo descartados. Ao não distinguir *prefetches* dos acessos reais, os *prefetches* não úteis acabam sendo incluídos na priorização dos recursos compartilhados gerando perda de utilização efetiva e desequilíbrio na distribuição dos mesmos. Para solucionar este problema o artigo propõe que somente demandas reais e *prefetches* úteis sejam considerados na formação de pacotes da técnica PARBS e na priorização de tempo virtual de término mais cedo da técnica NFQ.

Ao se adicionar porém *prefetches* na priorização como sugerido, aplicações com freqüente acesso à memória podem atrasar demasiadamente aplicações de acessos pouco freqüentes. O artigo sugere então que solicitações de acessos de aplicações de baixa freqüência sejam atendidas prontamente (*demanding boosting*) nas técnicas PARBS e NFQ.

Desta forma elas não sofrerão atrasos na sua execução e como elas têm baixa freqüência de acesso à memória não provocarão atrasos freqüentes nas outras tarefas.

A terceira técnica analisada é uma técnica de controle de demanda na fonte FST (*fairness via source*-throttling) do sistema de memória como um todo. Esta técnica calcula um valor estimado de atraso (*slowdown*) de cada *core* devido à interferência de outros *cores* e baseado nestes valor determina qual o *core* que está experimentando o maior atraso e qual *core* esta provocando o maior número de interferências nos acessos à memória. Ao se atingir um limite máximo de atraso para um *core*, o controlador de demanda reduz a taxa de demanda do *core* que causa maior interferência e aumenta a taxa de demanda do *core* que atingiu o limite de atraso máximo.

Nesta técnica, *prefetches* devem ser considerados de maneira diferenciada. Devem ser levados em conta tanto demandas reais como *prefetches* do core gerador de interferência, porém no *core* que sofre a interferência não se deve considerar atrasos nas solicitações de *prefetches* mas somente nas demandas reais, pois atrasar acessos de *prefetch* não necessariamente reduz o tempo de execução das aplicações. Ainda nesta técnica, é sugerida uma coordenação entre o controlador de *prefetch* com o controlador de demanda do *core*. Quando o controlador de *prefetch* decidir que deve reduzir a taxa de solicitação de um core, deve-se somente fazê-lo se o controle de demanda indicar que este *core* é o que mais gera interferência no outros *cores*, caso contrário não é necessário reduzir a taxa de demanda de *prefetch* deste core pois não está causando atraso em outros *cores*.

EXPERIMENTO

Foi utilizado simulador de multi-processadores x86 com 4 *cores* e modelamento do sistema de memória com seguintes parâmetros : core 15 estágios execução fora de ordem, decodificação 4 instruções, emissão e execução 8 micro-instruções, 128 entradas de buffer de re-ordenamento; Fetch de até 2 desvios, 4K entradas de previsão de desvio BTB e 64K entradas de previsão de desvio híbrida; L1 I e L1 D cache 32KB, 4way, 2ciclos,64B linha; cache L2 compartilhada 2M, 16-way, 16 bancos, 20 ciclos, 1 porta 64B linha.

Para a avaliação de desempenho foram utilizados programas do SPEC CPU 2000/2006 sendo agrupados de 4 em 4 para formação de 15 *workloads* misturando programas de acesso freqüente à memória e que geram *prefetches* de maneira intensa com programas de acesso à memória pouco freqüentes.

METRICAS

Para medição de desempenho foram usadas média harmônica e média ponderada de s*peedup*. Em relação à distribuição justa (*fairness*) entre os processadores foram utilizados as medidas de *MaxSlowDown* and *Unfairness*.

CONCLUSÔES

O artigo demonstrou que utilizando a modificações sugeridas obteve-se melhoria nos resultados de desempenho de 11%, 10,9% e 11,3% de um sistema 4 cores utilizando as técnicas de NFQ, PARBS e FST respectivamente e significativa melhoria na distribuição de recursos, obtendo portanto alto desempenho no gerenciamento de recursos compartilhado juntamente com os benefícios advindos do uso da técnica de *prefetch*.

(Resumo) i-NVMM: A Secure Non-Volatile Main Memory System with Incremental Encryption

Siddhartha Chhabra and Yan Solihin. 2011. i-NVMM: a secure non-volatile main memory system with incremental encryption. In Proceedings of the 38th annual international symposium on Computer architecture (ISCA '11). ACM, New York, NY, USA, 177-188.
(artigo original)
Murilo Adriano Vasconcelos RA: 134072 (resumo)

***Resumo*—Esse artigo se trata de um modelo para a criptografia de dados contidos em memória principal não-volátil (em inglês - *non-volatile main memory system* ou *NVMM*) por motivos de segurança. Isso porquê NVMMs sofrem de uma vulnerabilidade de segurança onde a informação contida nelas não é perdida quando o sistema é desligado, tornando possível um acesso físico ao sistema com objetivo de extrair informações sensíveis da memória. Para isso os autores introduzem uma técnica de criptografia incremental chamada i-NVMM, onde a maior parte da memória fica criptografada e apenas uma pequena fatia da memória permanece não-criptografada conforme uma predição realizada se o dado será utilizado ou não pelo processador. Quando o sistema é desligado, somente a parte não-criptografada precisa ser criptografada.**

I. Motivação

A motivação na criptografia incremental é a observação de que o **conjunto de trabalho** de uma aplicação (páginas de memória que uma aplicação está usando ativamente) é muito menor que o **conjunto residente** (todas as páginas de memória pertencentes à aplicação). Identificando o conjunto de trabalho de uma aplicação e criptografando o restante do conjunto residente, o método i-NVMM mantém a maioria da memória principal criptografada sem penalizar muito a aplicação, implicando assim em um pequeno tempo de espera após o desligamento do sistema para a conclusão da criptografia dos dados.

II. Métodos

O i-NVMM insere um preditor para detectar páginas "inertes". Essa predição é realizada varrendo as páginas da memória principal periodicamente para identificar as páginas que não foram usadas há algum tempo. Quando essas páginas são identificadas, elas são criptografadas em um motor de criptografia localizado no módulo de memória. Esse processo de criptografia é realizado por completo pelo módulo de memória para que o processo não seja dependente do conjuntos de instruções de um processador (ISA), uma vez que as NVMM devem ser compatíveis com uma ampla variedade de plataformas.

Como o i-NVMM identifica as páginas inertes e as critografa previamente sem esperar que o sistema seja desligado, a janela de vulnerabilidade, que é o período em que os ataques podem acontecer com o sistema desligado, é diminuída significativamente.

Para que o processo de criptografia incremental não onere muito o tempo de processamento, é necessário que a predição seja razoável, pois no caso de *misprediction*, uma página que está criptografada terá que ser descriptografada antes de ser usada pelo processador novamente. Outra ponto é que idealmente, uma grande parte da memória deve estar criptografada (alta cobertura) para que a janela de vulnerabilidade seja pequena. Assim os autores encontram experimentalmente um intervalo de varredura por página de inertes de 5 bilhões de ciclos e um limiar de inércia de 1 bilhão de ciclos (isso é, se uma página não foi acessada nos últimos 1B ciclos, ela é considerada inerte).

III. Arquitetura

Para fazer a predição de páginas inertes, o i-NVMM mantém o *status* de cada página da memória. Por razões de desempenho, os autores escolheram o tamanho de página de 4KB. O *status* de cada página é armazenado em um componente chamado *Page Status Table* - PST que é composto pelos seguintes campos:

- **EncStatus**: 1-bit que diz se a página está criptografada ou não;
- **LastAcc**: o tempo do último acesso à página;
- **NumAcc**: o número de vezes em que a página foi acessada;
- **NextPage**: a próxima página que foi acessada após essa, e
- **Pending**: 1-bit que diz se a página está pendente para criptografia/descriptografia.

O campo *LastAcc* é utilizado para checar se a página deve ser marcada como inerte ou não. Já o campo *NumAcc* é utilizado para checar se uma página criptografada deve ser mandada para uma fila de descripografia (*misprediction*).

IV. Conclusão e Resultados

O artigo apresentou um modelo de criptografia incremental para memórias principais não-voláteis. Pelos experimentos apresentados (ver artigo completo), esse modelo se não mostrou oneroso quanto ao tempo de acesso à memória. O processo de criptografia é auto-contido no módulo de memória, ou seja, não depende de ISA específico. Na média, cerca de 78% da memória permanece criptografada, o que resulta em um processo de criptografia após o desligamento do sistema de 5 segundos, o que é plausível uma vez que se assemelha ao tempo de retenção das DRAMs atuais. Finalmente, segundo os experimentos realizados pelos autores, o i-NVMM adiciona uma média de apenas 3.7% no tempo total de execução e usa em média 5.1% mais energia e tem um impacto insignificante na escrita.