Aluno
Heber Augusto Scachetti 004933

Artigo
Low-Power TinyOS Tuned Processor
Platform for Wireless Sensor Network Motes
R. K. RAVAL, C. H. FERNANDEZ, and C. J. BLEAKLEY
University College Dublin

Resumo
Uma rede sem fio de sensores (WSN) é uma rede com um grande número de nós de uma rede sem fio onde cada nó consiste em um processador, um sensor e um transmissor de radio freqüência. Este trabalho investiga otimizações na arquitetura de instruction set (ISA) do processador e funções do SO focando baixo consumo. Como atualmente os processadores mais utilizados em nós de WSNs não são otimizados para este tipo de aplicação este estudo ganha importância.

O TinyOs é um SO para nós de WSN desenvolvido originalmente pelo UC Berkeley. Passou a ser utilizado como referência em pesquisas de WSN. Este SO possui arquitetura baseada em componentes e utiliza linguagem C (NesC). Os componentes interagem através de comandos, eventos e tarefas.

Sentindo falta ou não conseguindo acesso a benchmarks associados a processadores de WSN o autor montou uma rede com 9 nós (utilizando aplicações específicas em processador Atmel ATmega128L) e os simulou por 300s. Os números de ciclos por função foram tabelados por aplicação do TinyOS assim como a frequência de utilização de instruções e seus pares.

Baseado nos resultados obtidos uma nova arquitetura do processador foi proposta: BlueDot. Segundo o autor, a nova plataforma poderia substituir a antiga mantendo a funcionalidade e sendo compatíveis com os nós convencionais. A arquitetura da plataforma do processador BlueDot consiste em um core processador programável específico para a aplicação e um acelerador de hardware interconectados através de um bus System onChip (SoC).

Baseado no instruction set do Atmel ATmega128L o autor: modificou as instruções multiciclos mais usadas para usar um único ciclo, transformou pares de instruções mais utilizados em instruções únicas, reduziu significativamente tempo de tratamento de interrupções, adicionou instruções para comunicação com o acelerador de hardware e, em alguns casos, acelerou o acesso a dados em RAM.

Baseado no tempo gasto com as funções de comunicação, a utilização do SoC e do acelerador de hardware permitiram ao processador gastar menos tempo e energia com o envio/recepção de mensagens utilizando frames completos (e não mais 1 byte por vez) a um hardware específico que se encarrega da transmissão e recepção da mensagem notificando o término destas atividades.

Segundo o autor do artigo, as otimizações propostas ao TinyOS diminuíram significativamente o número de ciclos e o consumo de energia. Tanto o consumo de energia como a quantidade de ciclos apresentaram melhoria em torno de 50%.

Título do Artigo: NoHype: Infraestrutura de Nuvem Virtualizada sem Virtualização
Citação bibliográfica: Eric Keller, Jakub Szefer, Jennifer Rexford, Ruby B. Lee. 2010 NoHype: Virtualized Cloud Infrastructure without the Virtualization. IEEE. The **IEEE** International Symposium on Circuits and Systems (**ISCAS**). Pages 350-361
Aluna: Luciane Politi Lotti - **RA** 012096

Objetivo geral - A computação em nuvem é uma tendência de ruptura que está mudando a forma como usamos computadores. A tecnologia-chave subjacente nas infraestruturas em nuvem é a virtualização tanto que muitos a consideram uma das principais características ao invés de simplesmente um detalhe de implementação. Infelizmente, seu uso traz preocupação com a segurança. Como executam múltiplas máquinas virtuais ao mesmo tempo no servidor e como as camadas de virtualização executam um considerável papel nas operações das máquinas virtuais, um ataque bem sucedido pode comprometer o controle da camada de virtualização, a confidencialidade e a integridade dos dados e dos softwares. Neste artigo é proposta a remoção da camada de virtualização (hypervisor), mantendo as principais características habilitadas pela virtualização.
A proposta de arquitetura - nomeada de NoHype, justamente para indicar a remoção do hypervisor, tem como objetivo endereçar cada uma das principais funções da camada de virtualização tais como: arbitragem de acesso na CPU, na memória e nos dispositivos de I/O; agir como um dispositivo de rede, como por exemplo uma switch Ethernet; gerir os *starting* e *stopping* máquinas virtuais convidadas; demonstrar que a arquitetura NoHype pode realmente ser viável desde que seus recursos estejam constantemente disponíveis (extensões de hardware, processadores, dispositivos de I/O). As características mais importantes são:
- Uma máquina virtual (VM) por Core - cada núcleo do processamento é dedicado e pode executar apenas uma única VM; estes núcleos não são compartilhados entre diferentes VMs o que atenua as ameaças de segurança relacionadas a canais laterais típicos no uso de recursos compartilhados (cache) ou em acessos a ambientes públicos hospedados, simplificando cobranças em termos de aplicações escaláveis. A arquitetura de múltiplas instâncias (partições de dados e configurações virtualizadas) é possível em virtude dos multi-core no chip. Cada VM, contudo, deve ser completamente isolada uma da outra inclusive em termos de dados, softwares, administração de trigger, dentre outros. Neste ponto os autores fazem ressalvas com relação aos modelos de ameaças adotados pelos provedores de nuvens e mencionam também que há um aumento na complexidade de gerenciamento da arquitetura proposta e que, portanto as camadas de virtualização devem ser removidas dos servidores de múltiplas instâncias dentro do papel que as mesmas possuem nos atuais modelos de tecnologia;
- Particionamento de Memória – particionamento forçado da memória física do hardware que garante a cada máquina virtual o acesso único e correto a memória física e ao intervalo que lhe foram atribuídos. O volume de particionamento é decido por cada cliente e eventuais subutilizações ficam a cargo dos mesmos. Este particionamento pertence à proposta porque ao contrário do processador que percorre um rumo à maior divisão de unidades, a memoria ainda é um recurso unificado;
- Dispositivo Virtual Dedicado de I/O: o dispositivo de modificações de I/O para poder suportar a virtualização permite que cada máquina virtual tenha acesso direto a um dispositivo virtual dedicado de I/O. As facilidades de gerenciamento de memória em conjunto com chipsets garantem que apenas as VMs autorizadas possam acessar a memória mapeada de I/O e somente em uma taxa limitada. No NoHype o acesso a taxa limitada de cada barramento de I/O é alcançado através de um mecanismo de controle de fluxo onde o dispositivo de I/O controla a taxa de transmissão, cada dispositivo da VM é configurado com a taxa na qual ele pode ser acessado. Para os componentes de rede, as modificações implicam que os roteadores de Ethernet devem executar as funções de comutação e de segurança dos dados, e não uma opção em software na camada de virtualização.
Conclusão - Os autores, ao invés de se concentrar em fazer virtualização mais segura, eliminaram a necessidade da mesma nas infraestruturas em nuvem. O NoHype fornece benefícios equiparados aos das modernas soluções de virtualização, mas sem esta camada e utiliza extensões de virtualização de hardware para eliminar tal camada. Os dispositivos de processador e I/O atuais serão muito mais requeridos, existem questões a serem resolvidas, mas a arquitetura não é um exagero e sim viável. Esperam que este trabalho estimule a investigação em many-core, ao invés de apenas investigar aspectos de desempenho e de melhorias de potência.

Characterizing Flash Memory: Anomalies, Observations, and Applications

Kaio Karam Galvão RA: 016480

23 de abril de 2011

Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. 2009. Characterizing flash memory: anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO 42). ACM, New York, NY, USA, 24-33.

Memórias flash apresentam algumas particularidades inconvenientes como durabilidade limitada, problemas de integridade de dados e diferenças de latência e de granularidade nas operações básicas de leitura, programação e exclusão. O objetivo da pesquisa descrita no artigo é superar tais características indesejadas e explorar ao máximo as características úteis. Para isso, faz-se necessário entender as relações entre desempenho, custo e confiabilidade dos dispositivos, além do efeito de padrões de uso sobre essas características. O artigo examina empiricamente tecnologias de memória flash, medindo desempenho, potência e confiabilidade. Em seguida, descreve dois exemplos de como aplicar tal compreensão das tecnologias.

Um chip de memória flash é dividido internamente em planos ou bancos. Planos podem ser bastante independentes, contendo buffer local para leitura e escrita de dados e realizando algumas operações em paralelo. Cada plano contém um conjunto de blocos, cada bloco formado por 64 ou 128 páginas. Dispositivos de memória flash provêm três operações básicas: exclusão, escrita e leitura. A exclusão é feita em blocos inteiros e torna todos os bits iguais a 1. A escrita opera em páginas inteiras e só pode passar bits e 1 para 0. E a leitura acessa uma página inteira em paralelo. O artigo lista a latência e o consumo de energia típicos para essas operações. O desempenho das operações de leitura e escrita apresentaria uma largura de banda de, teoricamente, 33 a 80 MB/s. Porém, na prática, isso é mais limitado pelas latências de leitura e escrita. Quanto ao dispêndio de energia, há uma drenagem de corrente entre 20 e 30 mA, a 2.7-3.3 V para uma potência de pico de 50 a 100 mW. Quanto à confiabilidade, o tempo de vida de um bloco é de 10.000 a 100.000 ciclos de escrita/exclusão.

Foi utilizada uma placa de testes que pode realizar medidas em dois chips de memória independentemente. Foram utilizados 11 dispositivos de 5 fabricantes. Nas medidas realizadas, a latência de leitura variou muito pouco entre fabricantes e chips. A latência de exclusão apresentou variações menores. Já os chips MLC tiveram, em média, latências de escrita longas e muito variáveis. A velocidade variou muito, e de maneira previsível, entre páginas de um mesmo bloco. Os chips SLC não mostraram tal variação. Tal efeito é, na realidade, bem conhecido dos projetistas de memórias flash. O desempenho dos dispositivos aumenta com o tempo de uso, tendo a operação de escrita um aumento entre 10 e 50 % na sua velocidade. Quanto ao gasto de energia, as medidas mostraram que as páginas que são mais lentas para escrever consomem mais energia nos chips MLC. Os chips SLC novamente não mostraram variação entre páginas. OS chips MLC em geral consomem mais energia nas operações básicas e quando estão "idle"do que os chips SLC. Os testes de confiabilidade mostraram que os chips MLC em geral apresentam erros logo após o termino do tempo de vida útil, a uma taxa de crescimento alta. Já os chips SLC apresentam baixa taxa de erros até 6 vezes o tempo de vida útil especificado.

O artigo propõe duas aplicações como novas maneiras de explorar o desempenho das memórias flash e superar suas limitações. Uma delas é uma Camada de Translação de Flash. Tal camada provê uma interface baseada em blocos, simulando um disco, que distribui as operações igualmente pela flash para garantir uma degradação uniforme. Os dados coletados permitem avaliar o impacto de padrões de acesso à memória na sua longevidade. A segunda aplicação consiste em definir um esquema de codificação que aumente a vida útil do chip.

O estudo descrito no artigo mostrou que, ao se conhecer mais do que o que está exposto nos datasheets das memórias pelos fabricantes, é possível fazer melhorias muito significativas nos armazenamentos baseados em flash. Particularmente, foi possível diminuir as latências de leitura e escrita e estender o tempo de vida útil dos chips de memória flash.

Resenha: Reducing Peak Power with a Table-Driven Adaptive Processor Core

V. Kontorinis, A. Shayan, R. Kumar e D. M. Tullsen

Aluno: José Arnaldo Bianco Filho RA: 019222

De uma forma geral os artigos que tratam de métodos de redução de consumo abordam apenas o aspecto da redução de consumo médio negligenciando o consumo de pico dos circuitos envolvidos. Esta abordagem está diretamente relacionada ao custo de soluções de resfriamento e confiabilidade. Porém o consumo de pico costuma ser negligenciado, embora este esteja diretamente relacionado à área do CI, encapsulamento e custo da fonte de alimentação.

O artigo em questão trata de um método de redução de consumo de pico com *power shutoff domain* baseado em uma tabela de alocação de recursos, o que diferentemente de outros estudos relacionados é, além de extremamente versátil em termos de configurações, proativo em relação à temperatura, uma vez que o mecanismo pode operar a todo o instante e não sendo acionado por monitores de temperatura, por exemplo.

Durante a realização deste estudo elaborou-se uma arquitetura modular de um processador no qual a maioria dos componentes é configurável e a partir daí uma tabela com uma lista dos recursos e suas configurações possíveis. Após levantar todas as combinações de recursos possíveis uma filtragem destas combinações foi realizada com base em determinados critérios: combinações que excedem o limite máximo de potência, combinações redundantes e combinações desnecessárias.

Adicionando área e consumo desprezíveis as combinações levantadas são adicionadas a uma ROM, que por sua vez é inserida no circuito juntamente com uma lógica de controle dos vetores de configuração e estatísticas do processador. Uma vez com estas informações poderemos optar entre uma configuração ou outra se baseando em eventos, distribuição de calor pelo DIE, contrabalançando variações de processo, isolando componentes com erros de fabricação ou mantendo o consumo em soluções multiprocessadas virtualmente constante com o uso de vetores diferentes para cada core.

Para validar os resultados teóricos o simulador SMTSIM e um modelo de processador com características de consumo similares a um Intel Core Solo e as capacidades já citadas foram utilizados. Uma grande variedade de *benchmarks* foi empregada a fim de explorar os benefícios da adaptabilidade. Após uma série de testes os resultados mais interessantes vieram com os modelos de técnicas de adaptação, tais como: INTV_RANDOM_NONE no qual um novo vetor sempre é escolhido randomicamente após um determinado intervalo de ciclos, INTV_SCORE_NONE no qual após um determinado período de tempo um novo vetor é escolhido com base em um sistema de pontuação, INTV_SCORE_SAMPLE cujo mecanismo difere do INTV_SCORE_NONE apenas pelo fato de que amostras são colhidas durante a operação em uma configuração e participam da análise de busca da melhor configuração ao final do período, EVDRIV_SCORE_SAMPLE no qual a analise do próximo vetor de operação ocorre com base em um determinado evento, não em um período e por último o INTVAD_SCORE_SAMPLE no qual quando uma alteração resulta em um desempenho inferior ao corrente o intervalo entre alterações é estendido, já quando resulta em melhor resultado o intervalo é reduzido.

Nos gráficos apresentados vê-se claramente a superioridade tanto do EVDRIV_SCORE_SAMPLE quanto do INTVAD_SCORE_SAMPLE sobre os outros mecanismos apresentados e, além disto, o quão eficiente é o mecanismo proposto, uma vez que a 75% de energia de pico a degradação de desempenho em relação ao processador com configuração máxima é de apenas 5%.

A técnica apresentada trás como grande benefício a possibilidade de reduzir a quantidade de capacitores de desacoplamento o que possibilitaria uma redução no custo dos processadores, devido à redução de área além de redução de custo de fontes de alimentação, que poderiam passar a operar com tensões de pico bem menores do que as correntes.

Título do Artigo: An Abstraction-Guided Simulation Approach using Markov Models for Microprocessor Verification
Aluno: Enos Aderval Vaciloto Ferreira de Lima RA: 032455

O contexto do presente artigo é a verificação funcional de ASIC/SOC, mais precisamente a verificação funcional de microprocessadores.

Na introdução do artigo o autor cita a complexidade crescente desta tarefa, devido à crescente complexidade dos circuitos e das microarquiteturas envolvidas. Os métodos tradiconais de verificação, aqueles baseados em simulação ou em métodos formais, esbarram em muitas limitações ao tentar cobrir a verificação de estados muito difíceis de serem alcançados (corner-cases). No intuito de transpor tais limitações alguns pesquisadores propõem uma combinação dos dois métodos mencionados que culmina numa nova técnica denominada semi-formal, uma dentre estas técnicas, que se mostra muito promissora e´ a simulação guiada por abstração (abstraction-guided simulation).

A idéia é fazer com que algum tipo de modelo, baseado de alguma forma no design, possa guiar a geração de estímulos para que se alcancem os corner-cases de forma mais eficiente; a cada passo de simulação o modelo abstrato avalia os proximos estados afim de escolher aquele que mais se aproxima de um determinado estado alvo da verificação. Para tal, o autor apresenta uma abordagem de simulaçao guiada por um modelo de abstração, onde o modelo é baseado em um modelo de Markov. O modelo de Markov é baseado na especificação do design, a simulação é guiada por métodos heurísticos que ajustam dinamicamente o modelo.

O modelo de Markov pode ser representado como um grafo. No modelo proposto pelo autor, cada vértice do grafo representa uma instrução do ISA de algum microprocessador, e cada aresta tem um peso. Na simulação, o design é estimulado por vetores de teste gerados pelo modelo de Markov, o controle de simulação se encarrega de analisar o estado do RTL e então gerar atualizações no peso probabilistico de cada aresta do grafo, fazendo com que o modelo seja atualizado antes da proxima geração de estímulo.

O autor aplicou a técnica mencionada utilizando um modelo de Markov implementado em C++ e utilizando PLI do Verilog para verificar um design de um microprocessador DLX e um microprocessador MIPS32. Em cada caso, o espço de estados considerado foi composto de todas as combinações possíveis dos registradores. Alguns estados alvos foram selecionados e a geração de testes foi repetida 20 vezes, cada uma com uma seed aleatória a fim de avaliar o desempenho.

Os resultados obtidos, quando comparados com outros métodos, tais como simulação puramente randômica e simulação guiada e com estímulo randômico demonstram que a técnica apresentada traz ganhos significativos tanto em termos de tempo de simulação quanto em número de estados alvos atingidos.

Task Superscalar: An Out-of-Order Task Pipeline

Yoav Etsion – Felipe Cabarcas – Alejandro Rico – Alex Ramirez
Rosa M. Badia – Eduard Ayguade – Jesus Labarta – Mateo Valero

Resumido por Ian Liu Rodrigues – RA061485

April 20, 2011

Introdução A computação paralela em HARDWARE está bastante famosa, o que motivou a pesquisa de modelos simplificados de programação paralela. Um método que vem crescendo é a paralelização baseada em tarefas, como OpenMP e CUDA. Esse método de paralelização é inconveniente pois deixa a cargo do programador lidar com as dependências de dados entre as tarefas, algo que pode ficar muito complicado.

Outro modelo que também está crescendo, consegue resolver dinâmicamente as dependências de dados entre tarefas baseando-se em anotações no código do programa. O programador indica operandos de entrada e saída que são usados para compor um grafo de dependência e executados paralelamente. Já existem soluções baseadas em software, como Jade e StarSs, entretanto são muito lentos.

Pipelines "Out-of-Order" resolvem dependência de dados entre instruções de maneira muito rápida. A ideia deste artigo é adaptar este método a nível de tarefas, o que simplificaria muito a implementação de paralelismo sem pagar o custo de velocidade do software.

Motivação: Rápida Decodificação de Dependências Após a criação de uma tarefa, suas dependências de dados devem ser identificadas – isto é chamado de *decodificação de tarefa*. Afim de obter máxima velocidade, as tarefas devem ser decodificadas mais rapidamente do que são consumidas.

Se uma máquina possui P processadores e cada tarefa leva um tempo T para finalizar, então devemos decodificar uma tarefa a cada $\frac{T}{P}$ de tempo. Se aumentarmos o número de processadores, devemos aumentar a taxa de decodificação ou diminuir a taxa de execução, ie aumentar o tempo de execução de cada tarefa.

Aumentar o tempo de execução não é uma boa alternativa, pois o programa pode ficar limitado pela memória do sistema. Programas com tempo de execução mais longo demandam mais memória, podendo ultrapassar o cache L1 de cada processador.

Com isto em mente, afim de obtermos máxima velocidade, devemos otimizar os programas de maneira que estes não excedam a memória cache L1 do processador.

Tarefas como Instruções Aplicando estes dois princípios nas tarefas, podemos desenvolver um sistema de decodificação de dependência de dados:

- O mecanismo de decodificação precisa identificar todas as modificações que a tarefa pode realizar na memória compartilhada.
- As tarefas devem ser decodificadas em ordem, garantindo a ordem entre produtores de dados e consumidores.

Em nível de instrução, todos os operandos de entrada e saída são conhecidos, por isso é possível realizar o grafo de dependências sem mais intervenções do programador. Já no nível de tarefas, cada tarefa modifica o estado do processador de maneira arbitrária. Cabe ao programador (ou compilador) dizer como cada tarefa atua no estado compartilhado da memória.

Algorithm 1 Decomposição de Cholesky no modelo StarSs

```
#pragma css task input(a, b) inout(c)
void sgemm_t(float a[M][M], float b[M][M],
             float c[M][M]);

#pragma css task inout(a)
void spotrf_t(float a[M][M]);

#pragma css task input(a) inout(b)
void strsm_t(float a[M][M], float b[M][M]);

#pragma css task input(a) inout(b)
void ssyrk_t(float a[M][M], float b[M][M]);

float A[N][N][M][M];
for (int j = 0; j<N; j++) {
  for (int k = 0; k<j; k++
    for (int i = j+1; i<N; i++)
      sgemm_t(A[i][k], A[j][k], A[i][j]);
  for (int i = 0; i<j; i++)
    ssyrk_t(A[j][i], A[j][j]);
  spotrf_t(A[j][j]);
  for (int i = j+1; i<N; i++)
    strsm_t(A[j][j], A[i][j]);
}
```

Modelo de programação StarSs O modelo de programação StarSs suporta execução "Out-of-Order" com anotações nos parâmetros de funções, indicando se o parâmetro é de entra, de saída ou ambos. Veja a utilização das anotações no Algoritmo 1.

Frontend do Pipeline O pipeline possui vários módulos que possibilitam o processo. São eles

ORT Object renaming tables

OVT Object versioning tables

TRS Task reservation stations

Veja na figura 1 a relação entre os vários módulos.

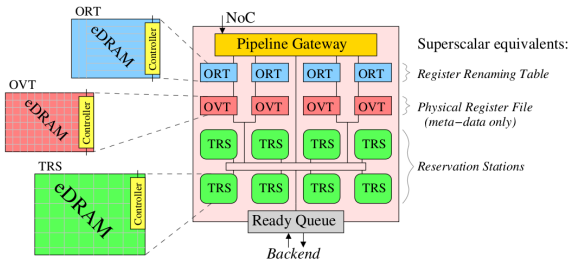

Figure 1: Organização dos diferentes módulos

A co-design approach for embedded system modeling and code generation with UML and MARTE

Jorgiano Vidal*, Florent de Lamotte*, Guy Gogniat*, Philippe Soulard+, Jean-Philippe Diguet*
*European University of Brittany – UBS – CNRS, UMR 3192, Lab-STICC
Centre de Recherche – BP 92116 – F-56321 Lorient Cedex - FRANCE
+SODIUS – 6 rue de Cornouaille – F-44300 NANTES - FRANCE
Artigo resumido por Wilson Vendramel – RA:075642

O artigo em questão propõe uma abordagem UML/MDA denominada metodologia MoPCoM para projetar sistemas embarcados de tempo real de alta qualidade.

A Unified Modeling Language (UML) foi utilizada para modelagem de aplicações desde a sua primeira definição. Um conjunto de propriedades UML foi identificado para a modelagem de sistemas embarcados. Essas propriedades incentivaram a adoção da UML para projeto de sistemas embarcados, mas ainda existem algumas lacunas, como a modelagem de uma plataforma. Para contornar o problema, Modelling and Analysis of Real-Time and Embedded Systems (MARTE) profile foi definido e adotado pelo Object Management Group (OMG).

Sobre os trabalhos relacionados com o artigo, os autores mostram que as metodologias de design anteriores comprovam que a UML pode sintetizar projetos de sistemas embarcados. Cada metodologia fez suas próprias extensões para adaptar a UML às suas necessidades, mas as extensões utilizadas por essas metodologias geralmente limitam a geração de código. Já este artigo utiliza a UML e extensões que permitem o uso de ferramentas genéricas e portabilidade de modelos. A metodologia apresentada pelo artigo define aplicação e plataforma separadamente, além de não estar vinculada com nenhuma linguagem de implementação, possibilitando a geração de código em várias linguagens tais como SystemC, VHSIC Hardware Description Language (VHDL) entre outras.

A abordagem Modélisation et spécialisaton de Plates-formes et Composants MDA (MoPCoM) é uma metodologia de co-design que segue os padrões OMG, e as técnicas Model Driven Architecture (MDA) possibilitam a geração de código. A metodologia é baseada no SysML profile, sendo melhorada com alguns elementos do MARTE profile. A MoPCoM define três níveis de projeto em modelos de sistemas embarcados de tempo real:

- Abstract Modeling Level (AML): o nível de modelagem de abstração é o primeiro do projeto, sendo que o objetivo é modelar a aplicação;
- Execution Modeling Level (EML): o nível de modelagem de execução é o intermediário do projeto, sendo que a análise de desempenho pode ser realizada;
- Detailed Modeling Level (DML): O nível de modelagem detalhada é o último nível do projeto que exibe a geração de código a ser feita. Outros níveis permitem a geração de código para simular o sistema, por outro lado o DML permite que a implementação de código seja gerada automaticamente.

A metodologia MoPCoM define três modelos para serem especificados em cada nível:

- Aplicação: contêm a especificação funcional do sistema em que objetos conectados se comunicam através de mensagens e sinais. Alguns elementos do MARTE são usados para expressar restrições de tempo real. No modelo funcional, o arquiteto especifica a aplicação do sistema por meio de um modelo orientado a objetos;
- Plataforma: é composta por um conjunto de componentes conectados, sem o envolvimento da aplicação. A plataforma é um conjunto de componentes de hardware onde a aplicação será alocada;
- Alocação: conecta a aplicação com a plataforma. A alocação mapeia a aplicação sobre os componentes da plataforma, onde a partição hardware/software é feita.

A MoPCoM mostra que a UML é apropriada para modelagem envolvendo hardware e software. Uma boa definição da metodologia ajuda na adoção da MDA em co-design, permitindo a geração de código facilmente. As experiências relatadas no artigo mostram que é possível a geração de código VHDL a partir dos modelos de sistema em UML.

A ferramenta de geração de código extrai os novos componentes de hardware a serem geradas e escreve o código VHDL para cada um. Para gerar o código, o artigo define três partes diferentes a serem geradas: estrutura, aplicação e comunicação. A aplicação é bem simples e assume a máquina de estado e linguagem de ação a partir do modelo funcional. A estrutura é integrada a partir da definição da plataforma, onde os componentes UML são traduzidos sobre as entidades VHDL. As portas VHDL são definidas a partir de protocolos e parâmetros. A comunicação é um elemento chave na geração de código e depende tanto do modelo funcional da aplicação quanto da plataforma.

Para finalizar, os autores afirmam que a principal dificuldade dessa pesquisa está relacionada às questões de comunicação entre os componentes. A pesquisa considera o sistema inteiro a ser modelado. As regras de design precisam ser bem definidas para possibilitar a geração do código VHDL da aplicação. O artigo não apresenta trabalhos futuros.

Título: Characterizing the Resource-Sharing Levels in the UltraSPARC T2 Processor

Citação bibliográfica: *Vladimir Cakarevic, Petar Radojkovic, Javier Verdu, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky and Mateo Valero.* ***Characterizing the resource-sharing levels in the UltraSPARC T2 Processor****. In 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), New York, USA, December 2009. Pages 481-492*

Autor do Resumo: Alexander Peter Sieh (RA:087984)

As limitações do paralelismo no nível de instrução (ILP) motivaram o paralelismo no nível de linha de execução (TLP). As técnicas de paralelismo no nível de linha de execução têm diferentes paradigmas como, por exemplo, os processadores SMT compartilham a maioria dos recursos em hardware, mas processadores CMP compartilham apenas memória cachê. Já que cada técnica TLP oferece diferentes benefícios, muitos desenvolvem processadores que combinam várias técnicas e paradigmas de TLP. Esta combinação de técnicas e paradigmas de TLP acrescentou complexidades em nível de software, pois a interação entre duas linhas de execução varia dependendo da disponibilidade dos recursos compartilhados. Linhas de execução disparados no mesmo núcleo podem compartilhar mais recursos de hardware que linhas de execução disparadas em diferentes núcleos. Para atingir o melhor desempenho em um sistema TLP, o sistema operacional precisa explorar todas as capacidades do processador de linhas múltiplas de execução, e para isso, o sistema operacional precisa considerar os recursos de hardware compartilhados em diferentes níveis ao fazer o agendamento das linhas de execução para alavancar um melhor o desempenho.

O foco de estudo e teste para esta conclusão foi o processador UltraSPARC T2. Este processador representa a recente tendência nos processadores TLP, pois ele une técnicas de FGMT e CMP ao mesmo tempo. O processador T2 que tem oito núcleos, sendo que cada núcleo tem oito estágios que podem compor dois pipelines. Os oito núcleos são conectados em rede crossbar e compartilham uma memória cachê L2. Esta arquitetura de processador pode ser dividida em três camadas de compartilhamento de recursos: InterCore, IntraCore e IntraPipe. Foi criado um benchmark para cada uma das três camadas de recursos compartilhados. Estressando cada camada de recurso compartilhado, os itens que mais afetam o agendamento de tarefa e o balanceamento da carga no processador T2 foram identificados.

No lado de hardware, em nível de IntraPipe, a unidade de Instruction Fetch foi o mais crítico como recurso compartilhado, pois esta unidade causou uma lentidão em até 4x. Em nível de IntraCore, as tarefas que não executam instruções de ponto inteiro ou flutuante com pipeline sofreram lentidão de até 7.6x. Em nível de InterCore, tarefas com muitas colisões no cachê L2 sofreram lentidão de até 9.2x, porém colisões na rede crossbar em X tiveram efeito de menos que 15%. Portanto, no lado da aplicação, as tarefas que tinham CPI baixo foram bem sensíveis aos recursos compartilhados em nível de IntraPipe. No entanto, as tarefas que não permitiram pipeline e com operações de longa latência foram as mais sensíveis aos recursos compartilhados em nível de IntraCore devido as colisões ocorridas na unidade de ponto flutuante. E por fim, as aplicações que usaram grande quantidade de dados no cachê L2 foram altamente sensíveis aos recursos compartilhados em nível de InterCore.

A caracterização deste processador T2 permitiu que o programador de tarefas no sistema operacional defina uma carga ótima e distribua os processos entre diferentes domínios de hardware (pipelines e núcleos) para evitar interferência entre eles.

Estas análises de benchmark foram reconfirmadas com testes usando uma real aplicação de rede. O programador de tarefas ciente dos recursos compartilhados em hardware no seu processador obteve um resultado com taxa de transferência superior em até 55%. Além disso, mesmo com testes em que a distribuição de linhas de execução foi balanceada, o desempenho é melhor em 17% quando o programador de tarefas também é ciente das características da aplicação. Sabe-se que esta melhora no desempenho aumenta com o número de linhas de execução simultâneas, com mais recursos de hardware disponíveis, com mais de núcleos nos processadores ou estágios por núcleos. Portanto espera-se que caracterização como feitas neste artigo ajudem os programador de tarefas explorarem ao máximo os recursos compartilhados dos processadores de “multithreading”.

MO401 - TRABALHO 1

Autor: Paulo Henrique da Cunha Pedrosa – RA 095354

Titulo do Artigo: Low Power Branch for Embedded Aplication Processors

Citação Bibliográfica: Levison, N.; Weiss, S. Low Power Branch Prediction for Embedded Application Processors. ISLPED 2010, Austin, 67-72, 2010.

Autor do Resumo: Paulo Henrique da Cunha Pedrosa – RA 095354

Com o objetivo de reduzir o consumo de potência em processadores, os autores desenvolveram um estudo focando em um componente responsável por uma parte relevante do consumo de potência, a estrutura de predição de desvios. Este estudo resultou em um artigo no qual é proposto um novo método de elaboração da micro-arquitetura de processadores, capaz de reduzir a potência dinâmica e estática, tendo como conseqüência uma pequena queda no rendimento do sistema.

O estudo concentrou-se em uma estrutura de predição de desvios que necessita de um alto consumo de potência, o Branch Target Buffer (BTB). Foi observado que a maneira como uma instrução de desvio é localizada no BTB faz com que duas instruções consecutivas sejam mapeadas para diferentes conjuntos BTB. A partir de simulações os autores perceberam que ao promover o deslocamento do campo *index* do endereço da instrução, o qual é responsável por indicar onde o desvio pode ser encontrado, aumentava-se a probabilidade de dois acessos seqüenciais serem feitos ao mesmo conjunto BTB. Além deste novo mapeamento, foi proposto o armazenamento de todo o conjunto BTB em um pequeno e rápido *buffer*, denominado *Set-Buffer*. Como resultado desta estrutura proposta, os autores afirmam que a potência dinâmica foi reduzida devido à eliminação de uma alta porcentagem de acessos BTB redundantes. Também foi assumido que o acesso ao componente BTB com a estrutura *Set-Buffer* terá um acesso infreqüente, o que torna possível a redução da potência estática através da mudança de estado de alguns conjuntos BTB para o estado de baixa fuga de corrente (low-leakage power), energizando os mesmos somente quando ocorre um acesso BTB.

O artigo também expõe técnicas de projeto para obter o baixo consumo estático e dinâmico, focando naquelas utilizadas no projeto proposto, com suas particularidades, apresentando equações e gráficos para elucidar os resultados obtidos. A performance obtida recebeu atenção especial, a qual apresentou uma pequena queda, tendo como motivo principal os conflitos por perdas (miss) do BTB.

O resultado apresentado se mostrou bastante expressivo. Usando um processador ARM Cortex-A8, a configuração da estrutura proposta foi composta por um *2-way* BTB com deslocamento de três bits do campo *índex* e janela de amostragem com tamanho de 2000 ciclos. Em média, o consumo de potência dinâmica foi reduzido em 75%, o consumo de potência estática apresentou redução de 58% e a redução de performance em 0.6%.

Ao final do artigo os autores afirmam que esta solução é especialmente atraente para o uso em sistemas embarcados que possuem capacidade limitada de potência, mas salientam a importância da cuidadosa escolha de dois parâmetros primários de projeto, o intervalo de janela e a distância do deslocamento do campo *índex*.

Nome: Paulo Henrique Junqueira Amorim – RA: 095431

Sistema de memória principal de alta performance escalável utilizando tecnologia de memórias por alteração de fase

Scalable High Performance Main Memory System Using Phase-Change Memory Technology. Moinuddin K. Qureshi; Vijayalakshmi Srinivasan; Jude A. Rivers; ISCA 2009

Atualmente sistemas computacionais consistem de vários *cores* em um único chip e as vezes vários chips em um único sistema. Como o número de *core* aumentou, consequentemente o número de aplicações que são executadas simultaneamente aumentaram, que por sua vez a quantidade de memória também aumentou. Por várias décadas a DRAM é responsável por formar o bloco de memória dos sistemas computacionais. De qualquer modo, com o aumento do sistema de memória, uma porção significativa do custo e do consumo de energia é gasto pelo sistema de memória. Com isso pesquisadores estudam novos tipos de memórias que são mais competitivas em relação a DRAM. Duas tecnologias são promissoras a flash e a PCM (*Phase-Change Memory*), a memória flash armazena os dados em portas lógicas e a PCM, foco deste trabalho, armazena os dados por alteração de fase de material (vidros calcogenetos). Para representar o conjunto de bits, a memória PCM é divida por células, cada célula pode ter seu estado alterado para cristalino ou amorfo. O estado cristalino, chamado também de SET é obtido por pulso elétrico moderado e de alta duração, esse estado, deixa o material com baixa resistência, e armazena o bit 1. Já o estado amorfo ou também chamado de RESET, é controlado por pulso elétrico de alta potência, deixando o material com alta resistência, representando o bit 0. Os dados são recuperados detectando a resistência da célula, aplicando uma corrente de baixa potência. O material pode está em diferentes graus de cristalização, podendo armazenar mais de um bit, foi criado um protótipo que consegue armazenar 2 bits em uma única área física, ou seja, é possível representar 4 estados.

A proposta do trabalho é criar um sistema híbrido com memória DRAM e PCM. O modelo proposto é utilizar a memória PCM entre a memória DRAM e o disco rígido, diminuindo a latência causada pelos acessos em disco por causa dos arquivos de paginação. O armazenamento dos dados na PCM seria controlado pelo sistema operacional, utilizando uma tabela de páginas, assim como é feito hoje na DRAM.

As técnicas que serão apresentadas a seguir são as maiores contribuições do trabalho, foi possível aumentar a vida útil da memória PCM de 3 para 9,7 anos, um dos fatores primordiais para o mercado. A primeira técnica é um sistema de escrita chamado de *Lazy-Write* com o objetivo de reduzir o número de escritas na PCM, para isso, quando é apresentada uma página com falha, o sistema grava diretamente no *buffer* da DRAM, já para localizar as páginas presentes na DRAM, o diretório de *tag* da DRAM é estendido com um bit (P) de presença, quando a página que está no HD é armazenada no cache da DRAM o bit P é 0, no modo *lazy write*, só é escrito na PCM, quando o dado é enviado da DRAM e o bit P é 0. Se em um *miss* de DRAM ocorrer, a página é buscada na PCM, em seguida o bit P é escrito como 1, com isso evita-se de escrever novamente o mesmo dado na PCM. Normalmente a memória principal é escrita em pedaços, nesse trabalho também foi proposto a escrita por partes da página, caso um parte dela já esteja presente na memória PCM, só será escrito as linhas alteradas da página, essa técnica foi batizada de *Line Level WriteBack* (LLWB). Uma outra técnica é empregada para que as células sejam desgastadas de maneira uniforme, emprega-se um algoritmo que mapeia setores lógicos para diferentes setores físicos que são menos utilizados. Em algumas aplicações não existem vantagens para o uso de PCM, como aplicações que fazem *streming* de dados, a maior parte dos dados não são aproveitados da memória, com isso também é proposto um outra técnica chamada de *Page Level By-pass (PLB)*, o sistema operacional é responsável por ignorar a gravação na memória PCM.

Observou-se também que para um sistema híbrido com 32 GB, o sistema de memória consumiu em média 45% de energia, contra 59% de um sistema não híbrido. Com as técnicas apresentadas, os autores acreditam que esse modelo servirá de ponto de partida para novas pesquisas em memórias PCM, já que a mesma é 4x mais densa que a memória DRAM, e possui economia de energia significativa.

Huffman-Based Code Compression Techniques for Embedded Processors

Bruno Ribeiro da Costa
RA: 096149

Neste artigo, os autores introduzem um método para compressão de código executável em plataformas embarcadas. O trabalho, denominado de “Combined Compression Technique”, é resultado da combinação de outras duas técnicas elaboradas pelos autores: “Instruction Splitting Technique” e “Instruction Re-encoding Technique”.

Instruction Splitting Technique

A Instruction Splitting Techique (IST) é uma técnica independente do conjunto de instruções de arquitetura. Nessa técnica, as instruções são fragmentadas em padrões de tamanho variável, sobre os quais é aplicada uma codificação baseada em Huffman. Os autores afirmam que a vantagem de se fragmentar as instruções é que o tamanho do dicionário é reduzido significativamente, uma vez que fragmentos de uma instrução são padrões que possuem maior probabilidade de repetição do que instruções inteiras.

Pelo fato dos padrões, ou fragmentos, não estarem relacionados ao formato de instrução utilizado na arquitetura alvo, essa técnica pode ser utilizada em conjunto com outras técnicas que exploram particularidades do conjunto de instruções da arquitetura.

Instruction Re-encoding Technique

A “Instruction Re-encoding Technique” (IRT), ao contrário da IST, é uma técnica dependente do conjunto de instruções de arquitetura. Essa técnica consiste em, primeiramente, analisar o formato original das instruções para uma aplicação específica a fim de detectar os bits recodificáveis. Os bits recodificáveis são bits que podem ser recodificados sem alterar a funcionalidade de uma instrução (em geral são campos não usados por uma instrução, ou seja, que existem apenas para promover o alinhamento em memória). Depois de identificados, os bits recodificáveis são substituídos por símbolos *don’t care*. Esse novo código é denominado de *código modificado*. O segundo passo consiste em comprimir o código modificado utilizando codificação Huffman, gerando assim as *instruções codificadas* e a *tabela de decodificação*. Por fim, é aplicado um método para reduzir a tabela de decodificação, onde cada símbolo *don’t care* de uma instrução é codificado da mesma forma que os bits correspondentes da instrução anterior.

A IRT apresenta melhores resultados em termos de taxa de compressão do que a IST, entretanto possui maior *overhead* em desempenho. Além disso, a aplicação da codificação Huffman após a recodificação dos bits recodificáveis faz com que o número de transições em cada coluna da tabela seja reduzido e provê uma maior taxa de compressão final.

Combined Compression Technique

A “Combined Compression Technique” (CCT) combina as duas técnicas apresentadas a fim de obtém uma única técnica que apresente as vantagens de taxa de compressão alta e desempenho na decodificação. Nesta nova técnica, é feita a aplicação de uma variação do IST e do IRT. Os bits recodificáveis ainda são detectados e substituídos por símbolos *don’t care* e as instruções únicas (aquelas que não possuem repetições) são fragmentadas em padrões de tamanhos variáveis. Porém, antes de aplicar a codificação Huffman, os símbolos *don’t care* são, no CCT, codificados com 0 ou 1 a fim de torná-los idênticos a outros padrões.

De acordo com os autores, testes realizados comparando as técnicas (IST, IRT, CCT e somente Huffman), indicaram que o tamanho das instruções comprimidas usando CCT foi menor do que os produzidos pelas outras técnicas em todos os *benchmarks*. Em MIPS, por exemplo, obteve-se uma taxa de compressão média de 42% com CCT, enquanto que com IST obteve-se 50% e com IRT obteve-se 45%.

Referência

Talal Bonny and Jörg Henkel. 2010. *Huffman-based code compression techniques for embedded processors*. ACM Trans. Des. Autom. Electron. Syst. 15, 4, Article 31 (October 2010), 37 pages.

A Dynamically Adaptable Hardware Transactional Memory

Marc Lupon, Grigorios Magklis, and Antonio Gonzalez. 2010. **A Dynamically Adaptable Hardware Transactional Memory**. In Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43). IEEE Computer Society, Washington, DC, USA, 27-38. DOI=10.1109/MICRO.2010.23 http://dx.doi.org/10.1109/MICRO.2010.23

Alexandre de Queiroz - RA: 098321

1 Resumo

Este artigo apresenta aspectos críticos em sistemas que empregam especulação para gerenciamento de Memória Transacional e foca seu projeto em HTM sobre detecção de conflitos. A detecção de conflitos possui duas versões: pessimista (*eager*) e otimista (*lazy*). A pessimista procura detectar o conflito no momento em que é dada uma carga no registro da memória e a otimista, a detecção do conflito pode ser adiada até o final da operação. Normalmente os sistemas utilizam um método fixo, ou otimista ou pessimista, para resolver conflitos.

A proposta do trabalho é o desenvolvimento de uma nova forma para detecção de conflitos dinâmica em HTM, chamada de DynTM, que procura trabalhar com os dois métodos – o otimista e o pessimista. As operações consistem em troca de mensagens entre núcleos do processador, marcando vetores se houver algum conflito. Cada núcleo envia uma mensagem de *AbortTx* quando detecta algum conflito e aguarda as mensagens de *AbortAck* de todos os núcleos que usam a mesma linha da memória. Também utilizam *timestamp* a fim de verificar qual transação é mais recente e deverá ser descartada. Cada núcleo deve ter uma simples TMS (*Transaction Mode Selector*) para definir o melhor método de execução para cada transação.

Para a avaliação da DynTM foi assumido um CMP com 32 núcleos, com uma malha de 16 nós de interconexão com links de 64 bytes, cada um possuindo dois núcleos com 1MB de cache L2 compartilhada. Possui quatro controladores de memória de acesso a 4GB. Cada núcleo tem 2Kbit para assinaturas de leitura e escrita e 32-bit Vetores de Conflitos (um bit por núcleo) e um TMS com THT 16 de entrada.

O sistema de base, o apoio HTM e o protocolo de coerência UTCP foram simulados usando o *Simics Virtutech* e as ferramentas *GEMS* do grupo *Wisconsin's Multifacet*. Para a análise, executou-se aplicativos da suíte *benchmark STAMP* e dois *micro-benchmarks* das versões 2.0 da distribuição *GEMS*. As aplicações foram categorizadas de acordo com suas características: aplicações de baixa contenção foram executadas com 32 *threads* e com alta contenção com 16 *threads*.

Para a análise foi comparado o DynTM com quatro diferentes sistemas HTM e duas alternativas DynTM. Os quatro sistemas foram *Eager Fixed HTM* que é semelhante ao *FASTM-Sig*, *Lazy Fixed HTM*, *Dynamic Overflow HTM* e *Statically Programmed HTM*. Os resultados apresentaram um ganho de 34% sobre *Eager Fixed HTM* e 47% sobre *Lazy Fixed HTM*, superando o estado-da-arte em execuções fixas de HTM.

O ganho de DynTM em relação aos outros HTMS é devido ao tempo gasto em operações sendo 18% com execuções pessimistas em paradas/espera e 16% com *backoff*. Houve também um ganho de 17% sobre *Dynamic Overflow HTM* e 10% sobre *Statically Programmed HTM*.

Por outro lado, o *Lazy Fixed HTM* é melhor que *Eager Fixed HTM* para operações como *Intruder* ou *Yada*. No entanto, com aplicações pequenas de leitura e gravação com o *Btree* e *Vacation*, melhoram seu desempenho ao longo do processo. Para aplicações *Vacation*, o *Lazy Fixed HTM* superou o DynTM.

Testes de gerenciamento local também apontaram um ganho de 20% no desenvolvimento de *commit* distribuído com o DynTM.

Concluiu-se que DynTM é o primeiro sistema híbrido de Memória Transacional que utiliza um protocolo UTCP para coerência entre transações pessimistas e otimistas, sendo a primeira a oferecer o modo otimista que usa sistema baseado em *log* como mecanismo alternativo e mudança de contexto, que promove um ganho de velocidade média de 34% sobre sistemas que empregam versões fixas para detecção de conflitos.

MO401 – Trabalho1 - A rapid prototyping system for error-resilient multi-processor systems-on-chip

Abril/2011

M. May, N. Wehn, A. Bouajila, J. Zeppenfeld, W. Stechele, A. Herkersdorf, D. Ziener and J. Teich - A Rapid Prototyping System for Error-Resilient Multi-Processor Systems-on-Chip - DATE '10 Proceedings of the Conference on Design, Automation and Test in Europe, p375-380

Daniel Vidal RA 099049

O trabalho trata a respeito do projeto de SoC multiprocessadores tolerantes a erros não permanentes no nível de implementação física causados por exemplo por radiação ou problemas esporádicos de *timing.* Foram implementados processadores LEON3 com *datapath* auto-corrigível e com checagem do controle de fluxo assim como um barramento AMBA AHB configurável aplicados a um sistema de decodificação de canal de comunicação

Detecção e correção de erros em datapath – baseado em *Nicolaidis shadow registers.* Para detectar erros no datapath registradores "shadow" foram adicionados em todos os estágios e são comparados em cada ciclo e quando há um erro um sinal é enviado ao controle do datapath. Mais um registrador de histórico foi adicionado em cada estágio de forma que as operações possam ser re-calculadas em caso de erro, esse procedimento é chamado de micro-roll-back.

Detecção e correção de erros no controle – Assumindo que somente desvios fixos são tomados. Como os endereços de destino são calculados durante a compilação pode-se implementar um monitor no primeiro estágio de *fetch* do pipeline para vefificar os *branches, jumps, calls e rets* assim como a progressão incremental do programa (PC -> PC+1). Em caso de erro a instrução causadora do erro é executada.

Detecção e correção de erros em interconeção – O barramento AMBA AHB foi estendido para permitir diversos algoritmos de detecção e correção de erros como bit de paridade, códigos de repetição, código de hamming e outros. O modo de detecção de erro é configurado pela aplicação de forma que o algoritmo mais eficiente possa ser escolhido de acordo com tipo de dado a trafegar pelo barramento.

Plataforma de prototipagem – Foi implementado um sistema onde os processadores atuam como decodificadores *LTE Turbo* – muito utilizado em comunicações sem-fio. Os dados são gerados, codificados, transmitidos através do barramento AMBA e decodificados pelos processadores. Também foram adicionadas estruturas para emular erros nos caminhos de dados/controle e no barramento. A exploração do funcionamento foi feita em 3 cenários: normal(referência), com falhas e com correção automática de erro.

A injeção de erro sem os esquemas de proteção causam falhas no sistema ou geração de saídas erradas. Quando os artifícios de correção de erros são ativados o sistema funciona corretamente porém com impacto na latência de acordo com taxa de erros e do tipo de erro ocorrido.

A prototipagem em FPGA possibilita exploração rápida da implementação das proteções contra erros na microarquitetura versus custo na latência, substituindo simulações lógicas para avaliação do comportamento do sistema num longo período de tempo – simulações muito demoradas.

Título do Artigo: Graphical Model Debugger Framework for Embedded Systems
Citação Bibliografica: Graphical model debugger framework for embedded systems; Kebin Zeng, Yu Guo, and Christo K. Angelov; DATE '10
Autor do Resumo: Aloísio de Menezes Vilas Bôas **RA**: 100579

1 - Introdução

Nas pesquisas sobre Desenvolvimento de Software Dirigido por Modelos (DSDM), a certificação da qualidade da modelagem é realizada pelo emprego de técnicas como validação, verificação e teste. Esse artigo apresenta uma técnica complementar, o Framework de Depuração Gráfica de Modelos ou *Graphical Model Debugger Framework* (GMDF) que permite a certificação da qualidade da modelagem em tempo de execução.

2 - *Graphical Model Debugger Framework*

O objetivo do depurador é permitir que os desenvolvedores, ainda em nível de modelagem, obtenham informação para monitorar e gerenciar o sistema. Para isso o Sistema deveria ter as seguintes funcionalidades: *Execução em passos e breakpoint em nível de modelos; Animação do comportamento dos modelos; Templates* de modelos gráficos personalizados; Abstração e geração automática de modelos; Compatibilidade de diversos tipos de modelos e entradas.
Dessa forma o GMDF consiste em três partes:
Entrada do Usuário: Modelo do Sistema e Código Executável (Obtido através da transformação dos Modelos).
Depurador Gráfico de Modelos (DGM): É o núcleo do GMDF. O código executável que está funcionando no controlador embarcado envia comandos ao DGM continuamente. O DGM provê uma máquina de estados finitos baseada no modelo do sistema introduzido pelo usuário. O GMDF através de um "Guia de Abstração" permite que a representação desses modelos possa ser personalizada escolhendo, por exemplo, as formas das figuras que compõem a máquina de estados.
Sistema de Execução (SE): Inicialmente o SE recebe o DGM como entrada e o apresenta graficamente. Em seguida, o SE espera os comandos a serem enviados pelo código no controlador embarcado, uma vez recebido, o comando é convertido em ações tais como animações da máquina de estados.

3 – Implementação e Demonstração

O framework foi implementado como um protótipo na plataforma Eclipse que é muito utilizada para DSDM. Na versão atual, o único modelo de entrada que pode ser usada é o COMDES que pode ser criado e manipulado através da COMDES Development Toolset.

4 – Trabalhos Relacionados

Durante a elaboração do GMDF não foram encontrados trabalhos que compartilhassem o mesmo objetivo especifico, porém, a idéia de implementação foi inspirada em algumas técnicas e ferramentas como *LabVIEW* , que provê funções como geração de código, simulação e visualização para DSDM e Data Display Debugger (DDD), um depurador gráfico para linhas de comando que apresenta os dados de forma estruturada.

5 – Conclusões

O artigo apresenta um framework de depuração para softwares embarcados dirigido por modelos gráficos. A intenção é possibilitar que desenvolvedores de sistemas embarcados possam testar graficamente o *design* do sistema possibilitando a depuração em um alto nível de abstração.

Titulo do Artigo: Forward Flow: A Scalable Core for Power-Constrained CMPs

Citação Bibliografica: Dan Gibson and David A. Wood. 2010. Forwardflow: a scalable core for power-constrained CMPs. In *Proceedings of the 37th annual international symposium on Computer architecture* (ISCA '10). Pages 14-25.

Autor do Resumo: Ricardo Guimarães Correa – RA 107101

O Artigo estudado propõe uma nova arquitetura para chips de vários núcleos (CMP) de forma a criar um processador escalável de acordo com a necessidade. As arquiteturas atuais têm dado foco na exploração do paralelismo no nível de threads (TLP) por receio do consumo e dissipação de energia. No entanto, o artigo estudado propõem mais flexibilidade e melhorarias na eficiência dos processadores pelo paralelismo no nível de instruções (ILP) também. A idéia de processadores escaláveis é conseguir fazer um bom balanço entre performance e gasto de energia de acordo com a situação, já que os núcleos escaláveis podem reservam mais recursos para aumentar o ILP, ou desligar recursos para reduzir o consumo energético e aumentar o TLP.

A arquitetura do ForwardFlow, possui uma estrutura de lista de links entre as instruções, chamada de *Dataflow Queue* (DQ), que pode ser ativada ou não conforme a necessidade. As instruções, dados e dependência entre dados são armazenadas no DQ.

Durante o estudo da arquitetura, foi levado em conta que embora os processadores escaláveis possam aumentar sua capacidade de processamento de acordo com a necessidade, eles ainda devem oferecer uma performance e consumo equivalentes ao de um processador tradicional, de forma a se manter competitivo. Sendo assim, o CMP não pode perder desempenho para prover melhor escalabilidade. Outro ponto considerado foi que a maioria dos *workloads* de um pipeline não usam de forma efetiva fetchs em paralelo. Para melhorar o desempenho de um processador, seria necessário manter um grande numero de instruções dentro do pipeline e assim conseguir aproveitar o resultado de um fetch em paralelo. Estas instruções mantidas dentro do pipeline são chamadas de *Instruction window*. Por ultimo, o escalonador de instruções também foi analisado, já que o mesmo não é facilmente escalável por não estar dentro da SRAM. Neste caso, para melhor explorar o paralelismo, o ideal seria trabalhar com um grande numero de acessos a memórias simultâneas. No entanto um escalonador de instruções não escalável, limita bastante a exploração pelo paralelismo no nível de memória (MLP) e impede a melhoria de desempenho. A estratégia utilizada é através de ponteiros que indicam o sucessor do valor guardado (*Forward Pointers*). Estes ponteiros indicam também toda e qualquer dependência de registradores e memória, eliminando assim broadcasts, já que o próprio ponteiro indica quem aguarda pelo valor.

O coração da arquitetura do ForwardFlow é o *Dataflow Queue*, e é onde fica a cadeia de ponteiros de dependência. Este módulo esta envolvido no despacho, execução, finalização e commit das instruções. Cada um dos ponteiros do DQ indica a dependência de valor para o operando. Conforme uma nova operação é iniciada com dependência em um outro valor a ser calculado, um novo ponteiro é criado para indicar esta dependência. Assim que uma instrução é inserida no DQ, a mesma aguarda a disponibilidade de seus operandos para que possa rodar. Quando uma instrução é finalizada, o hardware de busca de ponteiros faz a leitura do destino do resultado da instrução e determina todos os registros que aguardam este resultado. Com esta informação, todos os registros dependentes são atualizados dentro do DQ. Conforme outras instruções possuem todos os operandos disponíveis, elas também podem entrar em execução.

A organização do DQ é baseada em bancos, de forma a aumentar o suporte a acessos concorrentes. Cada campo do DQ é implementado em uma SRAM separada, de forma a simplificar o design e também melhorar o paralelismo. Pelo mesmo motivo, pode-se escalar o DQ de uma forma simples. No entanto existe uma maior latência para que a atualização de informações em uma cadeia de ponteiros chegue de um banco para outro. Sendo assim, algumas atualizações de dados podem levar mais de um ciclo.

Assim como em outras arquiteturas, o ForwardFlow também possui especulação de branch e destino. Para executar tais tarefas são usados checkpoints do estado dos registradores, que podem restaurar o estado ao momento antes do branch.

Nesta arquitetura, a janela de instruções pode ainda ter seu tamanho configurado para o ajuste de desempenho versus consumo.

Como resultado da avaliação de desempenho do ForwardFlow, foi identificado que para execuções em thread única, houve uma redução de 23% no tempo de execução com relação a arquitetura tradicional. Durante os testes, enquanto a arquitetura tradicional sofre do chamado *IQ Clog*, que é a falta de capacidade para escalonar mais instruções, o ForwardFlow não sofre do mesmo problema. Mostrando que a nova arquitetura é realmente capaz de escalonar mais instruções simultâneas. Outras verificações indicam que conforme a janela de instruções é aumentada, outras partes do processador são mais bem aproveitadas, como o fetch, o controlador de memória e os links internos do chip. Isso tudo é refletido com um aumento do *functional unit throughput* (FUs).

Observou-se que alguns dos benchmarks (bzip e hmmer) não são sensíveis ao aumento da janela de instruções, pois são testes de controle intensivo e o ganho de desempenho com a janela de instruções é rapidamente perdido com a má previsão de branches. Nestes casos o aumento do tamanho da janela causou inclusive uma perda maior no desempenho.

Segundo a analise de consumo de energia do ForwardFlow, tomando como base a arquitetura tradicional, houve uma melhoria de consumo para janelas de instruções pequenas. No entanto, com o aumento da janela, o consumo de energia também aumenta, e não houve ganho em nenhum dos testes com janela de instruções grande com relação a arquitetura tradicional. Esse aumento de consumo é devido ao aumento de atividade em outros pontos do processador. Assim o ForwardFlow consegue melhorar a performance sem concentrar a dissipação de energia em um único lugar, o que é desejável. Embora a configuração mais simples do ForwardFlow consiga mostrar um ganho de performance com relação a arquitetura tradicional para vários dos testes, não houve nenhuma configuração estudada que mostrasse uma melhoria para todos os testes.

Outra avaliação realizada foi para a execução de uma aplicação multithread. Os dados também têm como base a arquitetura tradicional. Assim como no caso de thread única, houve um aumento geral no consumo e desempenho. O tempo médio de runtime foi reduzido em 12%, mas foi acompanhado de um aumento de energia de 32%. Como não foi estabelecido nenhum teto de consumo de energia, não foi possível informar qual configuração seria melhor. No entanto, assumindo que o teto de consumo aceitável fosse estabelecido em 8 vezes maior, pode-se dizer que em 9 dos 14 benchmarks o ForwardFlow poderia ser considerado. Assim como na execução em thread única, nenhuma configuração é mais eficiente para todos os benchmarks.

A conclusão do artigo é que provavelmente o cenário de threads únicas ainda continue a ser muito utilizada, e portanto não devemos criar nenhuma limitação no hardware que reduza a exploração do ILP. Com a arquitetura proposta, a execução em thread única já traz um ganho de 21% com relação a arquitetura tradicional e ainda pode ser ajustada para melhorar a performance em múltiplas threads. Esta arquitetura se mostrou mais eficiente em 44 dos 47 casos estudados.

Universidade Estadual de Campinas
Instituto de Computação – IC
MO401 – T1

Título Artigo: Closing the Gap Between UML-based Modeling, Simulation and Systhesis of Combined HW/SW Systems

Referências Bibliográficas:

[1] Y. Vanderperren, W. Mueller, and W. Dehaene, “Uml for electronic systems design: a comprehensive overview,” Design Automation for Embedded Systems, vol. 12, no. 4, pp. 261–292, 2008.
[2] M. Pauwels et al., “A design methodology for the development of a complex system-on-chip using uml and executable system models,” in System Specification & Design Languages. Springer US, 2004.
[3] Fujitsu, “New soc design methodology based on uml and c programming languages,” FIND, vol. 20, no. 4, pp. 3–6, 2002.
[4] E. Riccobene, A. Rosti, and P. Scandurra, “Improving soc design flow by means of mda and uml profiles,” in Proc. 3rd Workshop in Software Model Engineering, 2004.
[5] R. Boudour and M. T. Kimour, “From design specification to systemc,” Journal of Computer Science, vol. 2, no. 2, pp. 201–204, 2006.
[6] K. D. Nguyen, Z. Sun, P. Thiagarajan, and W.-F. Wong, “Model-driven soc design via executable uml to systemc,” in IEEE RTSS’04, 2004.
[7] W. Tan et al., “Synthesizable systemc code from uml models,” in UML for Soc Design, DAC 2004 Workshop, 2004.
[8] T. Kangas et al., “Uml-based multiprocessor soc design framework,” ACM Trans. Embed. Comput. Syst., vol. 5, no. 2, pp. 281–320, 2006.
[9] Agility compiler. [Online]. Available: www.msc.rl.ac.uk/europractice/ software/mentor.html
[10] OSCI, SystemC Synthesizable Subset (draft 1.1.18), 2004.
[11] Greensocs. [Online]. Available: www.greensocs.com

Autor do Resumo: Ivelize Rocha Bernardo **RA:** 109222

Resumo:

Devido à grande complexidade dos sistemas eletrônicos tem se verificado grande expansão em estudos envolvendo linguagens SLDL (*System Level Design Languages*) e UML com o objetivo de projetar sistemas eletrônicos.

É neste foco que o artigo em questão apresenta a uma modelagem baseada em UML e uma simulação baseada em SystemC, integrando diferentes ferramentas de projeto de sistemas eletrônicos para simular uma aplicação no FPGA(*Field Programmable Gate Array*).

Assim, o fluxo se inicia com UML/SysML utilizando a ferramenta Artisan Studio. Após esta etapa, a geração de código um-para-um é aplicada traduzindo os componentes de HW E SW automaticamente para SystemC.

A simulação é realizada por um simulador - Modelsim - e um software emulador - QEMU - que são sincronizados pelos padrões de comunicação determinado no código gerado. O Modelsim é utilizado como mixer das linguagens SystemC e VHDL, já o QEMU executa o sistema operacional linux e binários de software baseado no PowerPC 405.

Após o término da simulação, a ferramenta Agility SC Compiler é aplicada na síntese do SystemC para VHDL.

O VHDL gerado é a entrada para o FPGA. Em paralelo, os binários do software já compilados são diretamente encaminhados para fazer o download para o processador de destino individual, que por sua vez são executados com o FPGA.

O fluxo apresentado acima é uma visão macro que pode ser dividida nas seguintes etapas:

- UML *profiles*: com o objetivo de ampliar a capacidade do Artisan Studio, o artigo propõe a criação de três UML perfis: (i) para o SystemC, (ii) para a extensão, (iii) e para a integração C com o SysML.
- Geração automatizada do código baseado nos três perfis modelados.

Deste modo, o artigo introduz um conjunto de perfis UML para modelar sistemas de HW/SW e um esquema de geração de código automático para suporte a simulação, esta abordagem resolve a lacuna entre a modelagem UML/SysML e de simulação, além de automatizar significativamente a etapa de transferir manualmente modelos UML para SystemC.

Universidade Estadual de Campinas
Instituto de Computação

MO401 – Arquitetura de Computadores I
Trabalho I – Resumo de artigo
Revisor: Filipe de Oliveira Costa — RA 109230

Elastic Cooperative Caching: An Autonomous Dynamically Adaptive Memory Hierarchy for Chip Multiprocessors

Enric Herrero, José González & Ramon Canal

1 Resumo do artigo

Primeiramente os autores consideram que aplicações distintas possuem comportamentos e necessidades de acesso à memória diferentes, e que atualmente esta diferença dificulta o acerto em acesso à cache. Um exemplo apresentado pelo autor é a execução simultânea de *streaming* de vídeo e aplicações vinculadas à cache (e.g. editores de texto, antivirus, etc.) em computadores *desktop*. Em geral, os aplicativos de streaming não tiram proveito dos níveis superiores da hierarquia de memória e, geralmente, escrevem na cache uma grande quantidade de dados que não serão utilizados. No caso de caches compartilhadas o problema se agrava, pois a escrita destes dados pode apagar da cache blocos de dados de outras aplicações que seriam utilizadas posteriormente.

Visando ter um controle inteligente para distribuir os recursos de forma justa e tirar proveito das diferenças entre os aplicativos e reduzir a taxa de *misses* na busca de um dado na memória cache, os autores propõem nesse trabalho uma hierarquia de memória distribuída e dinâmica denominada *Elastic Cooperative Caching - ElasticCC* que se adapta automaticamente ao comportamento de cada aplicação. Os autores procuram gerenciar a realocação de recursos forma autônoma por meio de controladores de hardware para evitar a adição de uma complexidade extra para a camada de software e, ao mesmo tempo, reduzir os problemas de baixa escalabilidade de outras propostas existentes.

A abordagem ElasticCC proposta consiste em dividir logicamente a memória cache em várias caches L2 e separá-las em duas regiões (compartilhada e privada) que competem para ter espaço na cache. As regiões privadas armazenam todos os blocos excluídos da cache L1 e as regiões compartilhadas "espalham" estes dados para as caches vizinhas.

Este procedimento é realizado em duas etapas. A primeira é o reparticionamento de cache, que ajusta a proporção de espaço privado e compartilhado localmente para cada cache L2, evitando estruturas centralizadas que limitam a escalabilidade. Os autores apontam resultados mostrando que a sobrecarga de hardware da unidade de reparticionamento é mínima.

A segunda etapa trata-se do espalhamento de blocos de dados de maneira autônoma. Cada vez que a cache é reparticionada, uma mensagem em *broadcast* é enviada para todos os nós comunicando-os sobre este particionamento. Esta informação é utilizada posteriormente por um "Alocador de Blocos Espalhados", que irá distribuir os dados entre as caches de modo a enviando mais blocos excluídos para caches com mais espaço compartilhado. Sendo assim, blocos de quando algum bloco for excluído da cache L1, uma cópia dele pode ser encontrada na cache L2, evitando assim o acesso à memória.

Os autores ainda propõem uma extensão do trabalho que procura verificar se o espalhamento dos dados será necessário ou não, dependendo da aplicação. Assim, os autores procuram evitar que

as caches compartilhadas sejam preenchidas com blocos que não serão utilizadas pelo processador.

Os experimentos realizados apontaram bons resultados. Em média, a abordagem proposta apresentou uma redução de *misses* de cache de 18,6% e um aumento de eficiência de energia de aproximadamente 71%.

2 Citação Completa do Artigo

E. Herrero, J. González & R. Canal. *Elastic Cooperative Caching: An Autonomous Dynamically Adaptive Memory Hierarchy for Chip Multiprocessors.* Proc. ACM International Symposium on Computer Architecture (ISCA), New York, NY, USA, 2010.

Artigo: Complexity effective memory access scheduling for many-core accelerator architectures, George L. Yuan, Ali Bakhoda, Tor M. Aamodt, MICRO-42
Aluno: Felipe Alexandre Soares
RA: 109547

Nas arquiteturas computacionais de hoje o sistema de memória DRAM é compartilhada entre os vários núcleos de processamento, como utilizado no exemplo do artigo uma unidade de processamento gráfico. Nesse sistema a memória compartilhada recebe vários acessos que se intercalam e interagem de uma forma que destroem a localização de acessos inerentes de cada fluxo de acesso individual, reduzindo assim a eficiência com que a DRAM pode transferir dados através de seu barramento de dados. A eficiência da DRAM é definida como a porcentagem de tempo que um chip DRAM passa a transferência de dados sobre os pinos de dados, dividido pelo tempo que há pedidos pendentes de memória, para maximiza a eficiência, diversos controladores de memória foram projetados baseados em tono do princípio de agendamento fora de ordem, onde o controlador de memória pesquisa através de uma janela de pedidos e toma decisões para maximizar o acesso. Este artigo trata da observação de aplicações altamente paralelas não gráficas executadas em uma arquitetura de GPU tem alto nível de execução de buffer DRAM, mas a rede de interconexão que liga os vários núcleos serve para intercalar o pedido de memória, destruindo sua localidade de linha de acesso. Propondo uma solução para alcançar o agendamento da DRAM comparável ao de escalonadores fora de ordem sob encomenda. Utilizando simples controladores de memória de forma, juntamente com um esquema de arbitragem mais inteligentes nos roteadores da rede de interconexão, que pose se recuperar muito do desempenho perdido quando um controlador de memória é usado em comparação com um escalonador fora de ordem. O artigo mostra que o acesso a localização do fluxo de solicitação de memória pode ser destruído pelas redes de interconexão, e introduz um novo esquema que preserva o acesso desta localização, permitindo um controlador de DRAM muito mais simples. Apresenta também uma analise qualitativa e quantitativa do desempenho, e apresenta uma solução simples para lidar com a intercalação que pode ocorrer devido ao roteamento da rede de interligação dos múltiplos canais virtuais.

O progresso tecnológico permitiu o aumento do throughput de processador exponencialmente, mas as características físicas impediram que a quantidade de pinos pudesse ser aumentada. Para amenizar esse problema os projetistas maximizam a quantidade de pinos utilizados para transferência de dados. Aumentar a quantidade de memória fora do chip exige o aumento de número de canais de memória, que necessita de um conjuto de pinos de endereço para comunicação e pinos de controle para transmitir o comando para controlar os chips DRAM. Uma forma de reduzir o número de canais de memória é ter um controlado de memória para múltiplos chips DRAM. No entanto existe um limite para esse “paralelismo em nível de chip”. Alem disso, aumentar o número de chips DRAM por controlador de memória reduz o numero de leitura/gravação de comandos por linha ativa para cada chip. A fim de maximizar a eficiência DRAM, controladores DRAM irão agendar as execuções fora de ordem.

O artigo demonstra que o esquema de arbitragem de interconexão junto com agendador de ordem obtém até 91% do desempenho obtido com um agendador fora de ordem para uma rede de barramento cruzado com oito fila de entradas.

Tree Register Allocation

Hongbo Rong (Autor do Artigo),
Divino César Soares - Ra: 115121 (Autor do Resumo)

A alocação de registradores é o processo de fazer o mapeamento das variáveis de um programa para os registradores disponíveis para o programa durante a execução. Existem dois escopos para o algoritmo de alocação de registradores; alocação local a qual faz a alocação das variáveis de um mesmo bloco básico e a alocação global que faz a alocação das variáveis de toda uma rotina.

Alocação global de registradores é um problema que pode ser reduzido para o problema de colocaração de grafos de interferência. Um grafo de interferência é um grafo de intersecção do escopo de vida de uma variável (*lifetime* em inglês), onde um *nó* do grafo representa o escopo de vida de uma variável e existe uma aresta entre quaisquer dois *nós* cujas variáveis que eles representam possuem uma interseção nos seus tempos de vida durante a execução do um programa. A alocação de k registradores para armazenar as variáveis do programa torna-se então o problema de colorir o grafo de interferência com k diferentes cores. Embora o problema de decidir se um grafo G pode ser colorido com k cores e construir esta coloração seja um problema NP-Completo, sob certas restrições o problema pode ser resolvido com tempo linear no número de arestas e *nós* do grafo. Este algoritmo é conhecido como Busca da Máxima Cardinalidade (*MCS - Maximum Cardinality Search* algorithm).

O algoritmo proposto no artigo (**Algoritmo 1**) constitui-se de três partes principais:

1. Um conjunto de árvores é criado representando todos os possíveis escopos de vida das variáveis do programa. Mais de um escopo é possível devido as estruturas condicionais do programa.
2. Executar o algoritmo de MCS para fazer a coloração dos *nós* das árvores.
3. A mesma variável pode ser mapeada para registradores diferentes quando existir mais de um escopo de vida para uma variável. Quando houver um fluxo de dados entre estes dois escopos é necessário que esta variável receba o mesmo registrador. O terceiro passo aplica uma permutação nos registradores atribuídos quando ocorrer o caso descrito.

Algoritmo 1 Allocation

```
   program : a given program.
   registers : the set of registers.
   root(t) : the root basic block of tree t.
1: Construct tree's for program
2: for all t in tree's do
3:    Active = { }
4:    R = registers
5:    Color(root(t), Active, R)
6: end for
```

A rotina *Color* refere-se ao algoritmo MCS para coloração de grafos. O algoritmo proposto exibe as seguintes vantagens:

- Generalidade: O algoritmo desenvolvido generaliza os conceitos presentes em diversos algoritmos existentes e que até então não se conhecia relação entre eles.
- Novas possibilidades: O autor analisa algumas premissas exigidas por algoritmos existentes e 'relaxa' estas premissas com base em novas observações.
- Fluxo de controle e informação de profile: A flexibilidade na escolha do algoritmo utilizado para construir as árvores utilizadas no primeiro passo do algoritmo permite que informações do fluxo e de profile do programa sejam utilizadas para gerar uma melhor alocação dos registradores.
- Simplicidade e vazão: O algoritmo desenvolvido possui a simplicidade e vazão conseguida por outros algoritmos baseados em grafos cordais.

Os resultados apresentados pelo autor conduzem a conclusão de que o algoritmo desenvolvido é um forte candidato para resolver o problema de alocação de registradores.

Evolution of Thread-Level Parallelism in Desktop Applications

G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, Evolution of Thread-Level Parallelism in Desktop Applications, In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*, pages 302–313, 2010.
Raphael Moreira Zinsly, RA 115153

Em 2000, Flautner *et. al.* realizaram um estudo sobre paralelismo em nível de *Thread* (TLP) em aplicações para computadores pessoais. Em 2010 Blake *et. al.* repetiram os experimentos de Flautner *et. al.* e apresentaram uma perspectiva da evolução do TLP em dez anos. Além disso, os autores verificaram o impacto do suporte a *multi-threading* simultâneo, das unidades de processamento gráfico (GPU's) e os efeitos da mudança para uma arquitetura de baixa potência, realizando os mesmos testes em um *desktop* baseado em processador embarcado.
Os autores estudaram até que ponto vai o potencial de execução concorrente em todo o sistema e como isso mudou em 10 anos. Também pesquisaram qual o impacto que o SMT causa no desempenho do paralelismo e como as GPU's estão sendo usadas para melhorar o desempenho do sistema, além de procurar quais as oportunidades de explorá-los ainda mais. Por fim, verificaram qual o impacto causado pela sofisticação da arquitetura e da frequência do *clock* no TLP. Os *benchmarks* utilizados foram aplicações comuns para computadores pessoais, como jogos 3D, criação de imagem e vídeo, reprodução de multimídia, navegação na internet e programas *Office*.
Como resultado, foi constatado que apenas dois ou três núcleos são suficientes para a maioria das aplicações para computadores pessoais. Todos os *benchmarks* mostraram uma porção de execução concorrente, mas em uma porcentagem muito baixa. Somente se tratando de criação de vídeo houve uma grande utilização de TLP. Constatou-se também que as GPU's, na maioria das vezes, não são utilizadas plenamente. Somente os jogos e programas CUDA exploraram bem as GPU's, para as outras aplicações os recursos permaneceram subutilizados.
Se essa falta de paralelismo nas aplicações para computadores pessoais não for inerente, é preciso que os programadores mudem sua forma de programar e utilizem significativamente mais paralelismo para tirar proveito dos futuros chips multiprocessadores e com vários núcleos.

Título: Way Stealing: Cache-assisted Automatic Instruction Set Extensions
Citação bibliográfica: Kluter, T.; Brisk, P.; Ienne, P.; Charbon, E.; , "Way Stealing: Cache-assisted automatic Instruction Set Extensions," Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE, vol., no., pp.31-36, 26-31 July 2009.
Autor do resumo: Lucas Martins De Marchi – **RA**: 121461

Way Stealing é uma modificação proposta neste artigo para a hierarquia de memória na execução de instruções ISE (Instruction Set Extension). O objetivo é aumentar a velocidade de obtenção dos dados necessários e ao mesmo tempo reduzir o consumo de energia.

Outros métodos foram propostos anteriormente, sendo um dos mais promissores o *Architecturally Visible Storage* (AVS), que utiliza memória local para aumentar a velocidade de acesso aos dados. Mesmo havendo um único processador, essa técnica demanda o uso de um protocolo para manter a coerência entre os dados mantidos no AVS e aqueles da cache L1, o que aumenta a complexidade da solução e o consumo de memória. Através da inserção de registradores de pipeline entre a unidade de ISE e a de memória e algumas mudanças estruturais (Fig. 1), esse trabalho mostra que é possível obter os mesmos benefícios de um AVS coerente, porém sem o overhead de um protocolo para manter a coerência.

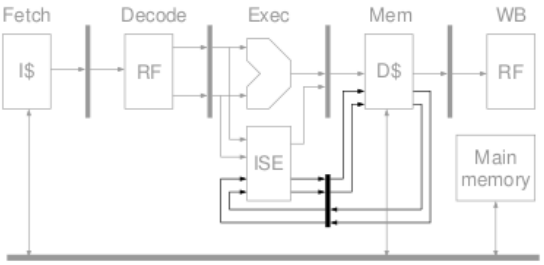

Fig. 1: Esquema arquitetural do método Way Stealing

Way Stealing funciona através da modificação da maneira como os dados são acessados numa cache. Normalmente em uma cache ***n*-*way set associative***, todos os ways são acessados da mesma maneira e a política de substituição de dados da cache não leva em conta em qual way um dado está armazenado. Por exemplo, com a política de substuição LRU, o dado usado menos frequente será o escolhido pra ser substituído, independemente de qual way ele esteja. Way Stealing força que todos os dados de um array a serem usados em uma determinada ISE estejam armazenados no mesmo way. Isso é possível graças à modificação no compilador, que gera as instruções de preload e lock para que os dados sejam carregados e não sejam substituídos, bem como à introdução de uma AGU (Address Generation Unit). Desta forma, é possível que as instruções ISE sejam executadas mais rapidamente já que o acesso aos dados necessários deixam de ser o gargalo.

Os testes experimentais foram conduzidos em um simulador de FPGA e benchmarks EEMBC, dando especial ênfase ao CJPEG. Além da versão original do código, compara-se também com a versão com ISE, mas sem Way Stealing. Para esse benchmark é possível notar que tanto o benchmark é executado mais rapidamente quanto menos energia é consumida.

Minimal Multi-Threading: Finding and Removing Redundant Instructions in Multi-Threaded Processors

Long, G., Franklin, D., Biswas, S., Ortiz, P., Oberg, J., Fan, D., Chong, F.T.
2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture

Resumido por Luiz Augusto Biazotto Filho, R.A. 121493

O presente artigo propõe uma micro-arquitetura *multi-threading MMT*, que utiliza o compartilhamento dinâmico de recursos para explorar a redundância de instruções entre múltiplos *threads*, com o objetivo de atingir ganho de desempenho e economia de energia.

O autor observa que, quando um programa *SPMD (single-program-multiple-data)* é executado em uma arquitetura *SMT*, seus diferentes *threads* buscam grandes seções de instruções idênticas (aproximadamente 88% das instruções), com um subconjunto dessas também tendo os mesmos valores de entrada de dados (35%, em média). É também observado que aplicações do tipo *multi-execution* possuem instruções to tipo *execute-identical* principalmente porque parte do seu grafo de execução é o mesmo, enquanto que nas aplicações *multi-threaded*, instruções do tipo *execute-identical* existem porque alguns dados são compartilhados por todos os *threads*.

Para encontrar e remover as instruções redundantes, são propostas modificações na arquitetura *SMT*, em hardware, que facilitem a sincronização de múltiplos *threads* com o objetivo de buscar e executar instruções idênticas. Também são propostas alterações nos estágios de decodificação e renomeação de registradores, com o objetivo de identificar e gerenciar instruções com a mesma entrada de dados, realizando execução única e aplicando o resultado a todos os *threads*. O foco do trabalho é em programas do tipo *multi-threaded* (*threads* se comunicam por meio de memória compartilhada) e *multi-execution* (*threads* não se comunicam na execução, e um programa separado acumula os resultados assim que todos os *threads* terminam sua execução).

O mecanismo de sincronização proposto funciona da seguinte forma: quando executados, todos os *threads* buscam instruções no mesmo local (estado *merge*); se um dos *threads* toma um desvio diferente do outro, há perda de sincronia, então, com o objetivo de re-sincronia, é armazenado o endereço do desvio tomado, em um *buffer* chamado *Fetch History Buffer (FHB)*, para cada *thread*; ao mesmo tempo, é verificado se algum outro *thread* possui o mesmo endereço-alvo em seu *buffer* (estado *detect*); se o endereço alvo é encontrado, os dois tentam uma re-sincronia (estado *catchup*), utilizando para isso um mecanismo de priorização de instruções; se a re-sincronia não for viável, volta-se ao estado *detect*.

Como há um aumento da janela de instruções em 4 bits *(ITID, Instruction Thread ID)*, durante o estágio de busca, se uma instrução do tipo *fetch-identical* é encontrada, os 4 bits identificarão os threads que utilizarão a mesma instrução. Um próximo estágio identifica se ela é também *execute-identical*. Caso ela seja, ela continua a execução e aplica o resultado nos *threads*, senão, será dividida em uma ou mais instruções, tendo seus *ITIDs* modificados. Essa divisão é feita por um novo estágio adicionado entre o estágio de decodificação e busca na tabela *RAT (Register Alias Table)*. É importante observar que é utilizada também uma tabela que armazena informações sobre o compartilhamento dos registradores entre *threads*, *RST (Register Sharing Table),* é há também um mecanismo de predição de *load*, para facilitar a decisão de divisão ou não de instruções para programas *multi-execution*.

Foram testadas diversas configurações *MMT*, tomando como base uma configuração *SMT* tradicional com *trace cache*. Diminuição do tempo de execução e do consumo de energia foram observados. Comparado a uma arquitetura *SMT* tradicional com *trace cache* e mesmo número de *threads* que a *MMT*, é alcançado *speedup* de 1.15 e 1.25, para dois e quatro *threads*, respectivamente. O ganho na economia de energia é da ordem de 10%-20%, comparando a MMT com a *SMT*, devido a redução de acesso a *cache* e redução de execução de instruções. Para poucas aplicações, a *MMT* não conseguiu sincronizar os *threads* suficientemente para identificar muitas das instruções idênticas. Não foi testada outra classe de aplicação, *message-passing*, que poderia se beneficiar do hardware *MMT* proposto, ficando como sugestão para trabalhos futuros.

MO401 - Trabalho 1

Register Cache System Not for Latency Reduction Purpose

Ryota Shioya, Kazuo Horio, Masahiro Goshima and Shuichi Sakai,
in Proc. MICRO, 2010, pp.301-312.

Autor do Resumo: **Francisco Romulo da Silva Araújo** – RA **121498**

O banco de registradores (register file) e a rede de encaminhamento (network bypass) têm sido uns dos gargalos dos atuais processadores superescalares. Para alcançar um alto IPC, tais processadores requerem um banco de registradores de alta capacidade para atender o número de instruções in-flight. Além disso, os cores SMT (simultaneous multithreading) requerem um banco de registradores proporcional ao número de threads. Como mostrado na figura 1 referente ao Pentium 4, a área do banco de registradores é comparável a cache de dados L1, apesar da grande diferença de capacidade, e a latência de acesso é quase a mesma, sendo necessário 2 ou 3 estágios de pipeline. O pipelining de acesso ao banco de registradores requer uma rede de encaminhamento maior, pois os resultados de instrução produzidos nos 2L ciclos (onde L é a latência do banco de registradores) devem ser encaminhados através da rede, podendo assim limitar a freqüência do clock e consumir uma quantidade maior de energia. Visto que o consumo de energia de uma RAM é proporcional a área do circuito e a freqüência de acesso, o banco de registradores, que tem uma área comparável a cache de dados L1, consome muito mais energia, pois quase todas as instruções acessam o banco de registradores mais de duas vezes (uma para leitura e uma para escrita). O consumo de energia e o calor resultante são uns dos problemas mais críticos (hot spot) dos núcleos de processadores atuais.

Em trabalhos anteriores, um sistema de cache de registrador foi proposto para resolver o problema dos grandes arquivos de registradores dos processadores superescalares. A cache de registrador reduz a latência do acesso ao arquivo de registradores para melhoria do IPC, simplifica a rede de encaminhamento e reduz a quantidade de portas do banco de registradores. Embora a cache de registrador possa resolver esses problemas, o escalonamento fora de ordem (out-of-order) de processadores superescalares não tem a capacidade de ocultar a latência de acesso a cache de registrador na presença de cache misses. Por esta razão, o artigo propõe um sistema de cache de registrador sem fins de redução de latência, Non-Latency-Oriented Register Cache System (NORCS), que é livre das penalidades (miss penalties) que sistemas convencionais com cache de registrador (LORCS[1]) sofrem.

LORCS e NORCS referem-se a um sistema de acesso a cache de registrador, composto pela cache de registrador, o arquivo de registrador principal e os estágios do pipeline de instrução para acessá-los. A figura 2 mostra o comportamento e os estágios (SC, IS, CR, EX, RW e CW)[2] do pipeline de instrução do LORCS. Os resultados de instruções escritos nos registradores são temporariamente armazenados em um buffer e utilizam a política write-through. O uso do buffer de escrita reduz o número de portas de escrita sem caminhos de forwarding adicionais. O pipeline do LORCS assume hit e tem um estágio para acessar a cache de registrador (CR), mas nenhum estágio para acessar o banco de registradores (similar a uma cache L1). Em cache misses, stall ou flush são geralmente adicionados. Visto que LORCS reduz a latência, isso degrada o desempenho consideravelmente devido ao distúrbio do pipeline.

Os diagramas de blocos (figuras 7 e 8) e os pipelines (figuras 2 e 4) mostram que NORCS tem uma estrutura similar ao LORCS, com algumas diferenças: (1) NORCS é caracterizado pelo seu pipeline único que assume miss da cache de registrador; (2) NORCS adicionou um conjunto de latches (destacados na figura 8) de pipeline ao sistema convencional, a fim de introduzir um estágio para ler o register file principal independente do hit/miss da cache (o pipeline não é imediatamente afetado pelos misses na cache). Os latches (figura 8) estão posicionados entre o array de tag e o array de dados da cache de registrador, ambos sendo acessados em estágios diferentes. Em um miss, o banco de registradores fornece um valor no fim do estágio de acesso ao banco de registradores. Em um hit, a cache de registrador fornece o valor também no fim do estágio de acesso ao banco de registradores.

Para avaliar o desempenho do sistema proposto, foi utilizado o simulador de processador (cycle accurate) *Onikiri 2*, que executou 29 aplicações do benchmark SPEC CPU2006. Os seguintes modelos foram avaliados em um processador superescalar 4-way e um (ultra-wide) 8-way: (I) banco de registradores pipelined com rede de encaminhamento completa; (II) banco de registradores pipelined com rede de encaminhamento incompleta; (III) NORCS; (IV) LORCS.

No geral, o uso do sistema proposto (NORCS) conseguiu reduzir a área e o consumo de energia para 24.9% e 31.9%, respectivamente, em relação ao banco de registradores pipelined, com um custo mínimo de 2.1% de degradação do IPC. Apesar desses resultados não serem diferentes de sistemas convencionais com cache de registrador (LORCS), o IPC difere bastante. NORCS e LORCS, ambos com cache de registrador 8-entry LRU, apresentam quase o mesmo consumo. Entretanto, NORCS melhora o IPC em 23% de LORCS.

Um detalhe importante, é que os autores mostram (utilizando dependência de dados e DFG) que um sistema de cache que não reduz a latência só funciona para cache de registradores, mas não para cache de dados usuais.

Por fim, foi proposto um sistema de cache que não reduz a latência de acesso ao banco de registradores e, o IPC não é melhorado por natureza. Em vez disso, NORCS é livre dos distúrbios de pipeline sofridos por LORCS, visto que o tempo para acessar o banco de registradores é fornecido como estágios no pipeline. Ou seja, o IPC não é necessariamente melhorado, mas não consideravelmente degradado. Mantendo o IPC, NORCS reduz a complexidade basicamente da mesma maneira que LORCS.

---

[1] Para simplificar, os sistemas convencionais de cache de registrador foram nomeados de LORCS (Latency-Oriented Register Cache System).

[2] Estágios do pipeline LORCS: scheduling (SC), issue (IS), register cache read (CR), execute (EX), main register file write (RW) e register cache write (CW).

High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)

Aamer Jaleel[1], Kevin B. Theobald[2], Simon C. Steely Jr.[1], Joel Emer[1]
[1]Intel Corporation VSSAD
Hudson, MA
[2]Intel Corporation, MGD
Hillsboro, OR

Resumo realizado por Marcio Machado Pereira – RA 855351

Abstract—The commonly used LRU replacement police always predict a near-immediate re-reference interval on cache hits and misses. Applications that exibit a distant re-reference interval perform badly under this police. This paper proposes a new cache replacement police using Re-Reference Interval Prediction (RRIP) that requires only 2 bits per cache block and easily integrate into existing LRU approximations found in modern processors and improve the perfomance up to 10% of such worloads.

***Index Terms*—Cache Memories, Parallel architectures, Cache Performance.**

I. RESUMO

Políticas de substituição de *cache* baseiam suas decisões de substituição em uma previsão de qual bloco será referenciado o mais distante no futuro e selecionam este bloco para ser substituído. Típicamente, em um *miss*, as políticas de substituição fazem a predição de quando o bloco ausente será referenciado novamente. Estas predições podem ser atualizadas quando novas informações sobre o bloco estiverem disponíveis, como por exemplo, em uma re-referencia.

Os autores citam que uma das motivações do trabalho de pesquisa deles foi o de que o uso eficiente do cache de último nível (*LLC – Last Level Cache*) é crucial para se evitar longas latências de *miss* de acesso a memória. Muitos estudos apontaram que, sobre a política de substituição LRU (*Least Recently Used*) comumente usada, filtros de localidade temporal sobre *caches* pequenos fazem com que a maioria dos blocos inseridos na LLC nunca sejam referenciados novamente. Esta utilização ineficiente da *cache* é devido ao fato da LRU ter um desempenho fraco para estes padrões de acesso. Para uma melhor compreenssão do leitor, os autores classificaram estes padrões comumente encontrados nas aplicações em 4 categorias: *Recency-friendly Access Patterns, Streaming Access Patterns, Thrashing Access Patterns* e *Mixed Access Patterns*. Depois, mostraram que há espaço para melhorias nas políticas baseadas em LRU para os dois últimos padrões como é o exemplo das políticas DIP – *Dynamic Insertion Police* e LFU – *Least Frequently Used*. DIP melhora a LRU em situações onde o intervalo de re-referencia está num futuro distante enquanto que a LFU prediz que os blocos frequentemente acessados serão re-referenciados num futuro mais próximo enquanto os blocos pouco acessados serão re-referenciados num futuro mais distante. No entanto, quando o intervalo de re-referencia dos blocos acessados pela aplicação consiste num padrão de acesso misto (*Mixed Access Patterns*) as políticas citadas não preservam blocos com intervalo de re-referencia quase-imediato. O artigo mostra que aplicações do mundo real tais como servidores, games e aplicativos de multimedia apresentam este comportamento. Estas aplicações experienciam explosões (bursts) de referencia de dados não temporais (chamados *scans*). Para melhorar a performance destas aplicações, os autores propuseram as seguintes políticas de substituição de cache utilizando *Re-Reference Interval Prediction* (RRIP):

RRIP prediz estaticamente os intervalos de re-referencia dos blocos faltantes (*missing cache blocks*) como intervalo de referencia intermediária, isto é, que estão entre as re-referencias quase-imediatas e as re-referencias distantes. RRIP atualiza esta predição para ser menor do que a predição anterior em cima de uma re-referencia. O artigo chama esta política de Static RRIP (SRRIP) e mostra que ela é *scan-resistant* e requer somente 2 bits por bloco da cache. O artigo descreve ainda duas categorias: SSRIP-Hit Priority (SSRIP-HP) e SSRIP-Frequency Priority (SSRIP-FP). SSRIP-HP prediz que qualquer bloco da cache que recebe um *hit* terá uma re-referencia quase-imediata e então ela deverá ser retida na cache por um período maior. Por outro lado, SSRIP-FP prediz que blocos da *cache* que são referenciados frequentemente terão uma re-referencia quase-imediata e portanto deverão ser retidos por mais tempo na *cache*. O artigo trata ainda da *Dynamic* RRIP (DRRIP) como um aprimoramento do SRRIP-HP. DRRIP é *scan-resistan* e *thrash-resistant* usando para isto um único contador de saturação para selecionar dinamicamente intervalos de re-referencia intermediaria e re-referencia distante.

Por fim, o artigo mostra que SSRIP e DRRIP tem desempenho superiores a LRU em média de 4% a 10% em um processador *single-core* com *16-way 2MB LLC* e de 7% a 9% em um *4-core CMP* com *16-way 8MB shared LLC* e ainda em 2,5% na política de substituição, *scan-resistant*, estado-da-arte, LFU. Para os estudos de performance foram utilizados o simulador CMP$im e programas do benchmark SPEC CPU2006.

Este trabalho foi apresentado na conferência ISCA'10 realizado entre os dias 19 e 23 de junho de 2010 na cidade de Saint-Malo na França. O resumo é requisito de avaliação da disciplina de Arquitetura de Computadores (MO401) do Instituto de Computação da UNICAMP (IC-Unicamp) e foi elaborado por Pereira, M. M. (e-mail: mpereira@ic.unicamp.br).

MO401 – Arquitetura de Computadores 1S2011
Alysson Bolognesi Prado RA 931544

Trabalho 1 - Resumo do Artigo:

Understanding Sources of Inefficiency in General-Purpose Chips

Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis and Mark Horowitz. ISCA 2010 pp. 37-47.

Objetivo

Este artigo compara o desempenho de ASICs – Aplication Specific Integrated Circuits – em relação a multiprocessadores de propósito geral – CMPs – programados para a mesma finalidade, em busca de alternativas de redução de consumo de energia e do tempo de execução.

Para quantificar o desempenho, foram analisadas implementações do algoritmo de codificação de vídeo H.246, escolhido por sua exigência de diversas características do processador: trechos do algoritmo podem se beneficiar de opções de paralelismo como SIMD – Single Instruction Multiple Data – e VLIW – Very Long Instruction Word – enquanto outros são altamente sequenciais, mas podem ser reescritos de modo a poupar instruções de controle.

Procedimentos

Inicialmente uma implementação padrão do algoritmo H.246 foi executada em um processador RISC extensível cujo fabricante (Tensilica) fornece software para *benchmark* de execução e simulação de customizações. O consumo de energia em cada uma das etapas do pipeline, fornecido por estas simulações, é a principal fonte de dados para análise das melhorias propostas. Neste cenário, o ASIC é 500 vezes mais eficiente em termos de energia e até 525 vezes em relação a velocidade.

Percebeu-se que as etapas de *fetch* e *decode* de instruções consumiam parte significativa da energia e tempo de processamento, o que é de se esperar em um processador de propósito geral. Procedeu-se então com a utilização de instruções SIMD e VLIW para que cada instrução, uma vez decodificada, pudesse ser utilizada para atuar simultaneamente sobre um conjunto maior de dados. Isso incrementa significativamente o desempenho nas etapas do algoritmo em que há muito paralelismo.

Em seguida foi dada atenção para o tempo e energia gastos no acesso ao banco de registradores. Para melhoria de desempenho, fez-se a fusão de operações que eram executadas sequencialmente, de modo que o banco de registradores é acessado apenas no início, para obter os parâmetros, e ao final, para guardar os resultados. Por exemplo, quando operações de multiplicação e soma sequenciais são fundidas, os resultados do multiplicador vão direto às entradas do somador e apenas o resultado deste volta ao banco de registradores.

Como o desempenho ainda era muito distante do verificado na implementação ASIC de referência, foi necessária a adição do que os autores chamaram de “magic operations”. Trata-se de customizações que exigem uma análise mais detalhada do algoritmo sendo executado, criando instruções que se beneficiam do conhecimento das estruturas de dados e controle usados pelo programa. Por exemplo, sabendo que um vetor de dados *X* é acessado a cada iteração *n* de um *loop* em uma janela X[n-1]...X[n+3], parte dos registradores pode ser modificada para criar uma estrutura FIFO onde apenas o valor X[n+4] precisa ser lido na próxima iteração. Outro exemplo transforma alguns *loops* aninhados em uma única instrução, que é executada em tempo linear, mais um *while* para ajuste do resultado, que raramente precisa ser executado.

Resultados

O uso de SIMD e VLIW aumenta o desempenho do sistema em 10 vezes, tanto em relação à velocidade de processamento quanto em consumo de energia. Com a fusão de operações o ganho passa a ser de 15 vezes para velocidade, mas sem ganho significativo no consumo de energia. Agregando as “magic operations”, atinge-se uma melhoria total de 600 vezes na velocidade e de 60 a 350 vezes na energia, deixando o CMP customizado com desempenho compatível com o do ASIC de referência, gastando em torno de 3 vezes mais energia. Não há diferença significativa na área usada no chip CMP em relação à superfície do ASIC.

O artigo conclui que a ineficiência de processadores de propósito geral está diretamente relacionada à presença de muitas instruções extremamente simples nas aplicações, consumindo pouca energia diretamente nas unidades funcionais por elas responsáveis e mais energia como *overhead* do restante do processador. Entretanto, os resultados obtidos encorajam a busca por soluções customizáveis com eficiência na mesma ordem de grandeza dos ASICs.