Supporting Distributed Shared Memory on Multi-core Network-on-Chips Using a Dual Microcoded Controller

Xiaowen Chen, Zhonghai Lu, Axel Jantsch and Shuming Chen; DATE 2010

Resumido por **Taísa Cristina Costa dos Santos** (RA **036065**)

Atualmente há uma tendência das arquiteturas *single-chip* evoluírem de *single-core* para *multi-core* e até mesmo *many-core*. Tal evolução exige um mecanismo de comunicação e organização dos núcleos de processamento que seja eficiente e escalável com o tamanho do sistema, característica a qual os mecanismos baseados no uso de vários barramentos convencionais não conseguem satisfazer. Neste cenário surge o conceito de "rede no chip" (do inglês, *Network-on-Chip* ou NoC), que aplica teorias e sistemáticas de redes a SoCs (*System-on-Chip*).

Outra tendência é o aumento do tamanho da memória embutida no chip, fazendo com que esta ocupe uma área cada vez maior do SoC. Memórias grandes e centralizadas são um gargalo de desempenho, consumo de energia e custo em sistemas de tamanho médio a grande, o que torna as memórias distribuídas cada vez mais importantes.

Considerando as duas tendências supracitadas, o artigo tem como proposta a implementação de um suporte à Memória Distribuída Compartilhada (DSM) em NoCs por meio da utilização de um módulo de hardware programável, denominado "*Dual Microcoded Controller*" (DMC). A escolha por memória compartilhada se deve ao fato desta facilitar a programação e permitir o reuso de código legado já testado e validado. Já o DMC é flexível ao permitir que o usuário o configure programando suas funções (e, assim, atende exigências como *time-to-market*, por exemplo) e apresenta bom desempenho.

Nesta proposta, considerando uma NoC em que cada nó é um conjunto processador-memória, cada um destes conjuntos é associado a um DMC, que faz a interface do processador com a memória local e a rede. A memória local é dividida em privada (que é a memória física e só pode ser acessada pelo processador local) e compartilhada (que é vista por todos da rede e é virtual). Este modelo tem por objetivo agilizar acessos privados, que são mais frequentes, mantendo um único espaço virtual.

O DMC possui dois miniprocessadores, um para o processamento dos pedidos originados no processador local e outro para pedidos de processadores remotos à área de memória compartilhada. Tais miniprocessadores têm sua operação acionada pela chegada de um comando (local ou da rede). Quando da chegada de um comando, o miniprocessador é alimentado com microcódigos (que inicialmente estavam armazenados na memória local e posteriormente numa unidade de *storage* de microcódigo) para serem executados. Ambos os miniprocessadores possuem acesso à memória local compartilhada, e, para lidar com transações atômicas do tipo leitura-alteração há ainda um "sincronizador", que resolve conflitos de acessos dos miniprocessadores a um mesmo *lock* simultaneamente (sincronização por *mutex*). O valor acessado pelo miniprocessador é, então, enviado através da rede em forma de mensagem ou entregue ao processador local. Quando o acesso é feito à memória privada, este é realizado diretamente, sem influência de um miniprocessador.

A proposta aqui resumida possui um *overhead* marcante quanto à tradução de endereço virtual para físico e à sincronização de transações (um pedido espera outro ser finalizado quando há um acesso concorrente a um mesmo espaço compartilhado). Há também o ônus de se adicionar um DMC a cada processador introduzido ao sistema (consumo de área e energia). Entretanto, experimentos com *workloads* sintéticos realizados usando dois modelos de tráfego na rede diferentes, *uniform* e *hotspot,* sugerem que o *overhead* do controlador se torna insignificante conforme o sistema aumenta de tamanho e os *delays* de comunicação passam a dominar o desempenho. Além disso, experimentos com *workload* de aplicação mostram que a proposta alcança um bom *speedup* de desempenho conforme o sistema aumenta de tamanho.

Por fim, os resultados de síntese mostram que o controlador pode executar até 455MHz consumindo 51K *gates* em uma tecnologia de 130nm, o que sugere que o DMC é um modelo viável de solução integrada, modular e flexível para resolver as questões de memória compartilhada distribuída em NoCs multi-core. Como trabalho futuro ficou o gerenciamento do consumo de energia, importante quesito de decisão sobre que abordagem seguir na implementação de um SoC.

MO401 - Trabalho 1 - Resumo de artigo

Parallel subdivision surface rendering and animation on the Cell BE processor

R. Grottesi, S. Morigi, M. Ruggiero, and L. Benini; DATE 2010

Resumido por **Conrado Silva Miranda** (RA 070498)

O artigo descreve uma combinação de arquitetura com algoritmo para gerar rapidamente gráficos agradáveis à vista, utilizando a técnica conhecida como subdivision surface e o processador Cell BE. Apesar da metodologia ser descrita apenas para um processador específico, ela pode ser utilizada em outros processadores ou placas de vídeo.

O Cell BE é uma arquitetura com um processador PPC de 64-bits (PPE) e um número variável de co-processadores (SPEs), conectados por um barramento (EIB). Cada SPE possui unidade de processamento (SPU) e controlador de memória (MFC). A SPU possui 128 registradores de 128 bits, instruções SIMD de 4 vias e memória de 256 Kb (LS). A MFC inclui controlador DMA, unidade de gerenciamento de memória, de interface com o EIB e de sincronização entre o PPE e as SPUs.

Subdivision surface baseia-se na criação de superfícies auxiliares que substituem as originais de um objeto, tornando seus contornos mais suaves, podendo ser aplicada iterativamente ou através de uma avaliação exata. Tal melhoria necessita de um processamento paralelo intenso, justificando o uso do processador Cell, com os passos do pipeline permanecendo no mesmo core, não sofrendo limitações de comunicação ou outros dispositivos e aumentando a performance.

O pipeline de renderização recebe uma cena 3D em primitivas geométricas e produz sua projeção 2D. A implementação proposta realiza os passos do pipeline em paralelo em grupos diferentes de dados. Para aumentar a performance, o PPE realiza um pré-processamento da superfície, transformando uma face em uma estrutura, chamada patch, contendo as informações necessárias para a renderização daquela face. O patch é salvo em uma patch stream que, juntamente com os vértices, compõem os dados que serão transmitidos para os SPEs.

Em seguida, porções contínuas da patch stream são atribuídas aos SPEs, que iniciam a transferência dos dados. Os vértices e os patchs são guardados na LS, sendo o máximo possível de 4 páginas de vértices. Caso algum vértice seja requerido e não esteja na LS, ocorre uma page fault e a página necessária é transferida. A construção dos patchs e a organização dos vértices é feita de forma a minimizar esse gasto. Além disso, como cada patch pode referenciar até 3 páginas de vértices e a LS pode receber 4, enquanto um patch é processado, a página necessária para o próximo é carregada.

Após carregar as páginas necessárias, o pipeline original é executado até a rasterização, quando uma sincronização entre elas é necessária para permitir visualização correta da imagem. Quando todas as SPEs chegam no ponto, as scanlines, linhas da imagem utilizadas para decidir quais vértices não serão utilizados, são distribuídas igualmente entre as SPEs. Novamente, cada SPE processa uma quantidade contínua de scanlines, reduzindo a quantidade de transferências necessárias. O processamento continua e a imagem é salva na memória principal no fim. Para aumentar ainda mais a performance, enquanto as SPEs processam uma cena, a PPE começa a pré-processar a próxima cena.

Para testar a imagem gerada, utilizou-se um programa iterativo. Os processadores ficaram parados durante apenas 2%, mostrando que o balanceamento de carga foi bem realizado. Com um SPE, atingiu-se 17 FPS, chegando a mais de 60 FPS com seis. O FPS caiu com a complexidade da superfície, mas suportando ainda grande complexidade, e a avaliação exata do subsurface division apresentou melhor resultado do que o método recursivo. Com isso, observa-se o pipeline proposto e o processador utilizado são capazes de processar imagens de complexidade razoável sozinhos.

Titulo: Modeling GPU-CPU workloads and systems / Bibliografia: The ACM Digital Library (http://portal.acm.org/results.cfm?coll=ACM&dl=ACM&CFID=87091431&CFTOKEN=16714906) / Autores: Andrew Kerr, Gregory Diamos, Sudhakar Yalamanchili / Aluno: Vilmar Travassos RA078272.
Este resumo apresenta uma emulação e tradução de infraestrutura, e seu uso, na caracterização de cargas de trabalho em GPUs (Graphics Processing Units) da NVIDIA, líder mundial em tecnologias de computação visual e criadora da GPU. Em particular, técnicas padrão de análise de dados foram empregadas (*benchmarks*) para caracterizar padrões de referência, suas relações com a máquina e com parâmetros de aplicação, além da construção de modelos preditivos para a escolha da implementação entre CPU ou GPU, do *Kernel* baseado nos resultados de tradução do Ocelot. Em novembro de 2006, a NVIDIA CUDA, apresentou uma proposta genérica de arquitetura de computação paralela, como um novo modelo de programação paralela e conjunto de instruções, que otimiza o processador de computação paralela em GPUs NVIDIA, para resolver vários problemas computacionais complexos de uma maneira mais eficiente que em uma CPU. CUDA vem com um ambiente de *software* que permite aos desenvolvedores usar a linguagem de programação "C" como sendo uma linguagem de alto nível; outras interfaces de linguagens de programação de aplicações são suportadas, tais como *CUDA FORTRAN*, *OpenCL* e *DirectCompute*. O *NVIDIA's Parallel Thread eXecution* (PTX), é uma arquitetura de instrução virtual, com execução semântica explícita de dados em paralelo, que estão bem adaptados às GPUs da NVIDIA. O PTX é composto por um conjunto de instruções de RISC-like para tipos explícitos de cálculos aritméticos computacionais, *loads* e *stores* para um conjunto de endereços, instruções para paralelismo e sincronização, e variáveis construídas. As funções implementadas em PTX, conhecidas como *Kernell* destinam-se a ser executadas por um grande número de *threads* organizadas hierarquicamente em uma matriz de segmento cooperativo (CTAs). A infraestrutura do compilador Ocelot se esforça para dissociar aplicações CUDA de GPUs, envolvendo o *CUDA Runtime API*, analisando o *Kernel* armazenado como bytecode da aplicação em uma representação interna, para execução desse *Kernel* em dispositivos presentes e mantendo uma lista completa de alocações CUDA armazenados em memória. Ao desvincular uma aplicação CUDA, a partir do driver CUDA, o Ocelot oferece uma estrutura para simulação de GPUs colhendo métricas de desempenho, traduzindo *Kernels* para outras arquiteturas GPUs, executando instrumentação e otimização o *Kernel* para a execução em GPUs. O *framework* de tradução Ocelot fornece uma eficiente execução de *Kernels CUDA* em CPUs multicore, já na primeira tradução do Kernel de PTX para um conjunto de instruções nativas de infraestrutura Low-Level Virtual Machine (LLVM), aplicando uma série de transformações que implementam o modelo de execução PTX, com controle de estrutura dos dados disponíveis para o tipo escalar de microprocessadores. Esse *insight* fornece uma clara necessidade de se aprofundar mais na caracterização e refinamento de modelos preditivos, um trabalho bem mais extenso.

Resumo: Using Virtual Filesystem to avoid metadata bottlenecks

Gabriel Dieterich Cavalcante — RA 079738 — mo401 - 2010

E. Artiaga and T. Cortes . *Using Filesystem Virtualization to Avoid Metadata Bottlenecks. Virtualization technologies.* DATE, Março-2010, p. 562. Dresden, Alemanha.

Sistemas de *cluster* dependem do compartilhamento de recursos entre vários nós, onde, aplicações paralelas podem dividir recursos espalhados ao longo do do *cluster*. Os sistemas de arquivos distribuídos apresentaram um modelo de armazenamento compatível com esta realidade, provendo mecanismos de distribuição de dados em vários servidores de armazenamento, que fazem com que as informações fiquem disponíveis de modo que nós do *cluster* enxerguem-nas como dados locais. Entretanto, o trabalho de Artiaga & Cortes identifica alguns casos onde existe degradação de *performance* no sistema, principalmente pela herança de comportamento dos sistemas de arquivos clássicos.

Nos sistemas de arquivos clássicos os diretórios são criados para indicar algum tipo de afinidade entre os arquivos, tendendo a compactar metadados desses arquivos em um só local. Os nós de *clusters* que rodam pequenas aplicações rotineiramente armazenam arquivos de controle/*checkpoint* em um mesmo diretório compartilhado, o que acarreta a criação muitos arquivos em paralelo dentro do mesmo. Estudos quantitativos observaram queda de rendimento principalmente pelo sistema de arquivos tentar manter um pequeno e atulizado controle sobre os metadados deste diretório compartilhado. O agravante principal está na punição empregada a todos os nós do *cluster*, e não somente a aqueles que estão "infringindo" as boas práticas de acesso.

Este estudo propôs COFS—*The Composite File System*, uma camada virtual entre os nós do *cluster* e o servidor de armazenamento distribuído GPFS—*General Parallel File System*—da IBM. Esta camada visa o desacoplamento do gerenciamento de arquivos e de metadados, e, além disso, cria diferentes visões de organização dos arquivos e de usuário. Para isto, divide grande diretórios compartilhados em sub-diretórios no nível de organização, porém abstrai esta visão no nível de usuário, mostrando apenas um diretório. Desta forma as aplicações conseguem seguir seu padrão de criação de arquivos—muitos em um mesmo diretório. Transparentemente o sistema armazena metadados em uma organização—um nó adicional—que evita o excesso sincronização criado, pois não há *metadata request* em vários nós do *cluster*.

Um protótipo desenvolvido adicionou um nó, responsável por armazenar informações sobre os metadados, além de uma camada extra em cada nó, que monta a "visão do usuário" do sistema de arquivos. No momento de criação do arquivos é gerado um *hash* das seguintes informações: o nó que requisitou a criação, o diretório pai na "visão do usuário" e o processo que está criando o arquivo. Após este cálculo será decidido onde o arquivo ficará armazenado no sistema de arquivos distribuído. Isso faz com que os arquivos sejam organizados em diretórios diferentes, ou seja, evita conflitos entre nós. Porém os arquivos continuam completamente próximos na visão do usuário.

Testes realizados demonstraram que o sistema de Artiaga & Cortes otimizou de 5 à 10 vezes o processo de criação de arquivos em diretórios compartilhados por nós do *cluster*, para outras operações—*open/close, stat*—o fator de *speedup* foi menor, porém ainda considerável. Alterar as estrutura geral dos arquivos poderia causar impactos negativos nas operações de leitura/escrita, pois eventualmente a localização dos dados poderia ser alterada pela camada de virtualização, o que causaria sobrecarga da banda de rede. Testes foram realizados e comprovaram que COFS teve consumo de rede similar ao GPFS nativo.

O estudo de Artiaga & Cortes mostrou que técnicas de virtualização são uma ferramenta válida para desacoplar o controle de nomes e metadados do gerenciamento de dados em baixo nível. Para isto uma prova de conceito implementada otimizou o funcionamento de um consagrado sistema de arquivos, o GPFS da IBM. O destaque principal vai para a melhoria de *performance* quando o sistema lida com ambientes compartilhados contendo grande número de arquivos, o que é muito comum em *clusters* que possuim vários nós rodandos pequenas aplicações independentes ou paralelas.

A Case for Bufferless Routing in On-Chip Networks

Resumido por Luciano Jerez Chaves - RA079759

Moscibroda, T. and Mutlu, O. 2009. A Case for Bufferless Routing in On-Chip Networks. *In Proc. of the 36th International Symposium on Computer Architecture* (ISCA '09). ACM, New York, NY, USA, 196-207.

O artigo em questão discute acerca do processo de roteamento de mensagens nas redes de comunicação que interligam os núcleos e as memórias *caches* dos modernos *chips* multiprocessados. No modelo atual, é associado um roteador à cada núcleo do processador. Estes roteadores se conectam com alguns de seus roteadores vizinhos, formando uma rede de comunicação de múltiplos saltos onde uma mensagem enviada por alguma origem pode passar por vários outros roteadores antes de chegar ao seu destino. Quando uma mensagem chega em um roteador, ela é armazenada em um *buffer* até que seja analisada e encaminhada para o enlace de saída apropriado. O problema levantado pelos autores do artigo diz respeito ao consumo energético e ao espaço ocupado pelos *buffers* nestes roteadores. Para melhorar estes aspectos, os autores propõem uma nova abordagem que dispensa o uso dos *buffers* no processo de roteamento das mensagens.

Para utilizar a solução proposta, existem duas restrições em relação ao projeto da rede: (1) a quantidade de enlaces de entrada em cada roteador tem que ser no máximo igual a quantidade de enlaces de saída neste roteador; e (2) qualquer roteador tem que ser alcançável a partir de todos os outros, formando um grafo conexo.

Para eliminar os *buffers* dos roteadores, é preciso encaminhar todas as mensagens (ou *flits*[1]) recebidos nos enlaces de entrada para algum dos enlaces de saída, mesmo que o enlace escolhido não seja o que aproxima a mensagem de seu destino final. Essa abordagem funciona por conta da primeira restrição em relação o número de enlaces nos roteadores (nenhuma mensagem deixará de ser encaminhada). O algoritmo parte do princípio que as mensagens desviadas eventualmente irão alcançar o seu destino (segunda restrição). Dessa forma, os *links* se tornam os *buffers* da rede, diminuindo a vazão máxima alcançável. A abordagem apresentada parece audaciosa para redes em geral, mas no caso específico das redes internas dos processadores, onde a carga pode ser considerada baixa, é possível conseguir benefícios com o uso desta técnica ao custo de algum incremento no tempo gasto para a entrega das mensagens (latência).

Para realizar a escolha de qual mensagem será transferida para qual enlace, o roteador constrói uma fila de prioridades com todas as mensagens recebidas. Cada mensagem é retirado da fila e enviada pelo enlace de maior interesse da mensagem, desde que este enlace ainda esteja disponível. Caso contrário, é utilizado o próximo enlace de maior interesse que esteja disponível. Para organizar as mensagens por prioridade, utiliza-se a abordagem do "mais velho primeiro", garantindo a menor latência e evitando *deadlocks* e *livelocks* (já que toda mensagem em algum momento se tornará a mais velha na rede e será roteada pelo caminho correto). Este processo também pode ser utilizada por roteadores que possuem *buffer*, desde que as restrições apresentadas sejam atendidas.

Para o caso da fragmentação das mensagens, os autores propõem o uso de uma solução semelhante ao roteamento *wormhole*, onde os demais fragmentos são sempre encaminhados pelo mesmo enlace do primeiro, de maneira a não provocar variações na latência de chegada de cada fragmento no destino. Entretanto, o artigo propõe uma "relaxação" na proposta original, podendo truncar a sequência de *flits* sempre que um *flit* "mais velho" precisa usar o mesmo enlace de saída, de maneira a garantir a ausência dos *livelocks*. Essa abordagem demanda por um *buffer* maior no receptor, que terá que lidar com os *flits* que podem ser recebidos fora de ordem por conta da truncagem da sequência.

Após a apresentação e descrição do algoritmo proposto, os autores discutem acerca da implementação deste algoritmo dentro do *pipeline* de roteamento. Uma grande vantagem em relação as soluções existentes é que o método proposto permite reduzir de 3 para 2 o número de estágios do *pipeline*, podendo chegar a 1 com o custo de um barramento adicional entre os roteadores para trafegar as informações sobre as rotas das mensagens.

Para avaliar o desempenho da técnica proposta, foi utilizado um simulador de redes acoplado à um emulador `x86`, que executou aplicações do *benchmark* SPEC CPU2006 em diferentes configurações de redes de interconexão. Também foram simuladas cargas de tráfego sintético. Foram consideradas a latência na entrega dos *flits*, a vazão de saturação da rede, a demanda por *buffers* no receptor, e o consumo energético da rede. Além disso, também foi avaliado o desempenho das aplicações executando sobre a rede com a arquitetura proposta, utilizando o *speedup* ponderado em relação o `IPC` alcançado por cada tipo de rede como métrica de qualidade.

No geral, o uso da técnica proposta conseguiu reduzir o consumo energético entre 40% e 45%, com uma queda no desempenho do sistema de apenas 1.5% nas configurações reais consideradas pouco estressadas. Este valor pode chegar à 17% em situações atípicas. Com o uso de tráfegos sintéticos, foi possível avaliar a vazão de saturação da rede, que foi reduzida entre 20% e 35% da vazão de saturação normal (roteamento com uso de *buffers*). A abordagem proposta demanda por *buffers* até 25% maiores nos receptores, mas ainda consegue reduzir a área total utilizada no circuito em até 60%.

Para concluir o artigo, os autores apresentam uma rápida revisão dos algoritmos de roteamento existentes para estas redes de interconexão, e reforçam as inovações existentes neste trabalho.

[1] Quando uma mensagem é muito longa, ela é dividida em partes menores, denominadas *flits*.

Achieving Out-of-Order Performance with Almost In-Order Complexity

Francis Tseng Yale N. Patt
Department of Electrical and Computer Engineering
The University of Texas at Austin
{ tsengf,patt} @hps.utexas.edu
International Symposium on Computer Architecture 2008
Aluna: Gabriela Batista Leão - RA:087348 – gabileao@gmail.com

Resumo

A quantidade de instruções despachada fora de ordem nas arquiteturas modernas pode ser ainda bastante aumentada, utilizando-se métodos de otimização tradicionais, a fim de explorar melhor o desempenho. No entanto, para refletir essas melhorias, adaptações devem ser implementadas diretamente no hardware, geralmente no custo do aumento da complexidade de seu projeto e requerimentos de potência. Como estes últimos fatores são fortes limitadores de desempenho, o trabalho apresentado propõe uma abordagem combinada de implementação no nível do compilador e da microarquitetura, considerando a máquina Alpha, cuja arquitetura contém o conjutno de instruções ISA. A abordagem proposta define, inicialmente, a granularidade das instruções (tarefa), estas representam um determinado código-fonte de uma linguagem de alto nível, a ser compilada, considerando análise de desempenho das aplicações na suíte SPEC CPU2000, que foi compilada para a arquitetura já referida. Esta análise se concentra no comportamento dos valores das variáveis ao longo da execução das aplicações, nomeadamente ao *fanout,* definido pela quantidade de vezes que um valor é lido, e ao seu tempo de vida, medido pela quantidade de instruções executadas entre sua produção e consumo. Os principais resultados dessa análise mostram, em média, que 90% dos valores é lido no máximo 2 vezes (*fanout*) e 80% dos valores tem tempo de vida de 32 instruções ou menos. Notando-se que tanto o *fanout* quanto o tempo de vida são relativamente pequenos, o grafo do fluxo de dados do programa pode ser dividido em subgrafos distintos. Cada subgrafo contém uma unidade, cuja granularidade é o *braid.* Cada *braid* abrange uma operação desempenhada em linguagem de alto nível, i.e. o *braid* reside somente dentro de um bloco básico. Essa abordagem requer (1) que o compilador seja capaz de construir *braids* a partir do grafo do fluxo de dados do programa. Além disso, (2) a ISA precisa transmitir informações do compilador para a microarquitetura e, finalmente, (3) a microarquitetura deve ser consciente de *braids* para tirar proveito destes últimos. Como *braid* é uma entidade identificada em tempo de execução, para cumprir o requerimento (1) o compilador deve analisar o grafo do caminho de dados do programa e armazenar o produtor e o consumidor de cada valor produzido. É também durante este passo que o compilador sabe que valores serão usados dentro e fora do bloco básico. A partir desses perfis, o compilador deve identificar os diferentes *braids* usando um algoritmo de coloração de grafos simples. A formação de um *braid* é realizada pela seleção de uma instrução dentro de um bloco básico e a identificação do subgrafo correspondente a essa instrução. Esse procedimento é repetido até que a última instrução do bloco básico seja selecionada e adicionada ao *braid* respectivo. Na sequencia, os *braids* são rearranjados de tal forma que instruções pertencentes a um mesmo *braid* sejam executadas sequencialmente. Finalmente, registradores externos são alocados primeiro e só depois a alocação individual de registradores de cada *braid* é realizada. Para transportar informações dos *braids* anteriormente definidos, (2), foram acrescentadas 3 instruções extendendo a ISA, que contêm como campos o bit de início do *braid,* um bit associado com cada operando de origem que especifica se o valor usado é carregado de um registrador interno ou externo ao *braid* e um bit associado ao operando de destino que definde se o resultado da instrução deve ser escrito em um arquivo de registrador interno ou externo. Por último, em (3) a microarquitetura deve refletir a alocação de registradores em 2 passos como já referenciado. A principal mudança na microarquitetura é a implementação de Unidades de Execução de Braids (BEUs), responsáveis por interpretarem as instruções descritas em (2). Conclui-se que a proposta melhora o desempenho em 9%, com complexidade de projeto similar a de instruçãoes em ordem, por simplificar o projeto de hardware nas fases de alocação, já que resultados de instruçãoes que escrevem para registradores internos não precisam ser escritos em registradores externos; renomeação, pois operandos internos não precisam ser renomeados; e um arquivo de registrores externos mais simples. A utilização das BEUs também remove parte da complexidade da rede de sobreposição.

"Multi-Execution: Multicore Caching for Data-Similar Executions"
Susmit Biswas, Diana Franklin, Alan Savage, Ryan Dixon, Timothy Sherwood,
Frederic T. Chong
Page(s): 164-173
ISCA 2009

João Eduardo Pereira dos Santos RA: 087563 MO401A

Resumo

O artigo em questão mostra que para manter as expectativas de desempenho, a arquitetura de microprocessadores está se voltando para o processamento baseado em multinúcleos (*multicore architectures*). Com o avanço previsto no paralelismo dessas arquiteturas, a demanda por largura de banda e a necessidade de se quebrar a gargalo que a memória convencional proporciona se torna algo crucial.
O foco do artigo é demonstrar que é possível aumentar o desempenho dos processadores multi núcleos, se o mesmo passar a executar muitas instâncias similares do mesmo programa.
Os autores propõem uma arquitetura de cache mesclável (*Mergeable cache architecture*) que detecta dados similares e mescla blocos de cache, resultando em economias substanciais nos requisitos de armazenamento de cache.
Além disso, os autores apresentam simulações e resultados de 8 benchmarks (*6 da SPEC2000*) para demonstrar que essa técnica proporciona uma solução escalável e leva a ganhos significativos de desempenho, devido a reduções nos acessos à memória principal.

Durante a analise, os autores explicam que uma das maneiras para aproveitar o poder de processamento dos processadores multi núcleos é através da execução de múltiplas cópias do mesmo programa, com diferentes entradas de dados ou parâmetros. A arquitetura atual não explora muito esse conceito, o qual os autores desenvolveram e a apelidaram de "*multi-execution codes*". Foi proposto uma arquitetura de cache mesclável que aumenta a capacidade de cache através da fusão de linhas de cache com conteúdo idêntico utilizado por diferentes processos, melhorando o desempenho na média em 2,5×, o que resulta também em um ligeiro aumento na área e no consumo de energia.
Para demonstrar a sua teoria os autores implementam um sistema de simulação do ciclo de precisão com base no simulador de multiprocessadores, o "*PolyScalar*".
Problemas foram encontrados para a utilização dessa técnica, que busca mesclar blocos de dados de múltiplos processos.
Primeiro: encontrar dados idênticos para mesclar é uma operação cara, se cada acesso a memória resulta na comparação do dado com todas as linhas de cache válido. Essas buscas devem ser minimizadas, enquanto, a oportunidade para identificar dados mescláveis é maximizada.
Segundo: dados mesclados precisam ser organizados de tal forma, que a busca pelo mesmo se torna mais rápida, utilizando uma simples leitura no cache.
Para resolver esse problema, da busca eficaz de dados idênticos, os autores observaram que é mais provável que as aplicações, disponham os seus dados idênticos no mesmo endereço virtual (mas em diferentes endereços físicos). Assim, a busca foi limitada somente para os dados com o mesmo endereço virtual. Para executar esta busca de forma eficiente, é preciso mapear todos os relevantes endereços virtuais para o mesmo conjunto de cache. Para atingir esse objeto, foi utilizado a técnica "*page coloring technique*", para inserir páginas na DRAM.

Essa técnica, segundo os benchmarks, apesar de ter aumentado ligeiramente os ciclos, a área e o consumo de energia, trouxe ganhos significativos no trafego fora do chip (*Offenbachismo Trafica*), através da diminuição dos "L2 cache misses" e "L2 cache writebacks". Oferece ganhos de performance em arquiteturas de multi núcleos, somente é observado uma diminuição no caso de processadores com 8 núcleos, por causa, dos "L2 cache misses".
Os autores concluem que através dessa técnica é possível identificar e mesclar dados idênticos de aplicações em multiprocessadores, salvando assim espaço na cache. Os ganhos de performance variam entre 6,92 vezes a 2,5 vezes na média. Outro item interessante é que essa técnica não é invasiva, ou seja, os programas não vão precisar de modificação.

A Mechanistic Performance Model for Superscalar Out-of-Order Processors
Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis e James E. Smith

Eyerman, S., Eeckhout, L., Karkhanis, T., and Smith, J. E. **A mechanistic performance model for superscalar out-of-order processors**. ACM Trans. Comput. Syst. 27, 3, Aticle 3 (May 2009).

Resumo: Hamilton José Brumatto – RA 096389.

Este artigo propõe um modelo mecanicista para estimar a performance de processadores de execução fora de ordem. A figura 1 (apêndice) mostra um processador típico superescalar de execução fora de ordem. Este processador incorpora em sua microarquitetura um conjunto de parâmetros representando a largura de banda, ou largura (Width – *W*) medida em instruções por ciclo (IPC). Na figura estão indicadas diversas larguras: busca (*F* - fetch), despacho (*D* - dispatch), distribuição (*I* – issue), retirada (*R* – retire). Na proposta de um processador balanceado, a largura é a mesma em todos estágios e é ditada por *D*.

O modelo mecanicista proposto constrói um intervalo de análise que quebra o tempo total de execução em intervalos baseados em eventos de perdas/erros (miss events). A figura 2 (apêndice) mostra um conjunto de eventos associados a estas perdas. No modelo considera-se que em um intervalo de *N* instruções, leva-se $\lceil N/D \rceil$ ciclos para despachar as instruções, se houver uma perda, haverá uma penalidade de tempo *c* associada à perda: Perdas de I-cache/I-TLB: c_{iL1} para I-cache L1, c_{L2} I-cache L2 e c_{iTLB} I-TLB; Erro em predição de desvios: c_{fe} (estágios no pipeline de entrada) + c_{dr} (descarte de execução do desvio); Perda de cache de dados (L2), o mesmo para D-TLB: c_{L2}. Estes tempos podem ser sobrepostos na ocorrência de duas perdas (as duas se equivalem), ou serializados (somam-se os tempos). Ineficiência do despacho: como o despacho múltiplo pode encontrar um intervalo de instruções no qual o número não é um múltiplo da largura do despacho, o termo $\lceil N/D \rceil$ *pode ser escrito como:* $N_{total}/D + (D\text{-}1/2D).(m_{iL1}+m_{iL2}+m_{br}+m_{dL2}(W))$, o efeito da borda na função teto pode ser representada como uma perda por despacho ineficiente, *m* representa o número de eventos associado a cada tipo de perda.

O tempo total de execução em ciclos é: $N_{total}/D + (D\text{-}1/2D).(m_{iL1}+m_{iL2}+m_{br}+m_{dL2}(W)) + m_{iL1}.c_{iL1} + m_{iL2}.c_{L2} + m_{br}.(c_{dr} + c_{fe}) + m_{dL2}(W).c_{L2}$

A performance geral do processador pode ser analisada e estimada, desta forma, através de intervalos individuais com diferentes tipos e comprimentos. A diferença média entre o resultado obtido no modelo mecanicista e simulações é de 7% em um processador superescalar de execução fora de ordem e largura 4 de despacho.

A partir do modelo foi realizado um estudo do efeito de escala de recursos nos processadores de execução fora de ordem. Para os processadores balanceados foi explorada a configuração do pipeline: profundidade, largura e o relacionamento entre ambos. A profundidade ótima de pipeline p^* para um dado processador obtida no estudo é aproximadamente proporcional à raiz da largura *D:*

$p^*(D).D^{-1/2} \sim cte.$ Em outras palavras, ao aumentar a largura do processador de um fator *c*, deve-se diminuir a profundidade de pipeline de um fator $c^{-1/2}$.

A parte final estuda o superdimensionamento (ou desbalanceamento) do processador. Normalmente se um estágio do pipeline apresenta uma largura maior, esta não será aproveitada, pois o estágio anterior não consegue entregar em uma performance maior, nem o posterior aproveitar tal performance. Por outro lado, se um estágio é subdimensionado, todos os demais apresentam perda, pois tal estágio passa a ser um gargalo.

Uma exceção ao superdimensionamento ocorre no despacho inicial, o aumento da largura no despacho inicial, em comparação com a largura de distribuição, aumenta a performance pois o processador consegue atingir o próximo evento de perda de forma mais rápida. Os resultados mostram que para um teste de performance houve um aumento de até 9% fazendo uma largura de 6 no despacho em um processador de largura 4 na distribuição e demais estágios.

O trabalho propõe uma continuidade no estudo considerando limitações na largura de banda de memória, efeitos de pré-busca de instrução por hardware e também no estudo de modelos não balanceados de projeto de processadores.

Apêndice

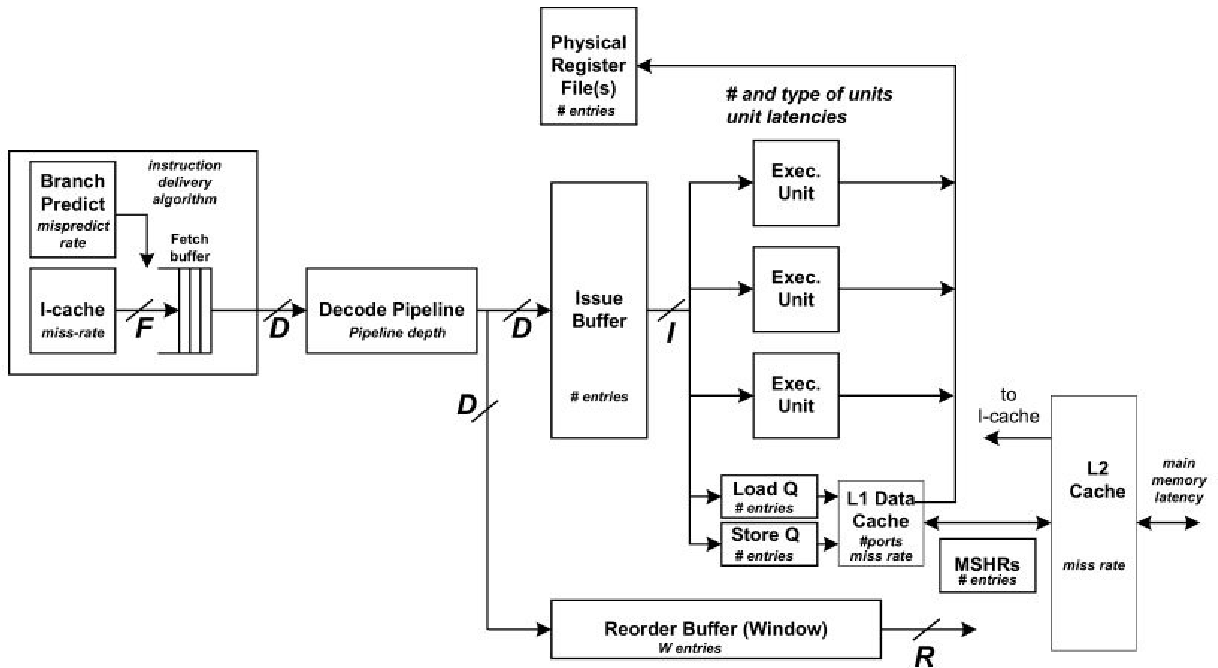

Figura 1 – Processador superescalar de execução fora de ordem parametrizado (extraído do artigo em revisão).

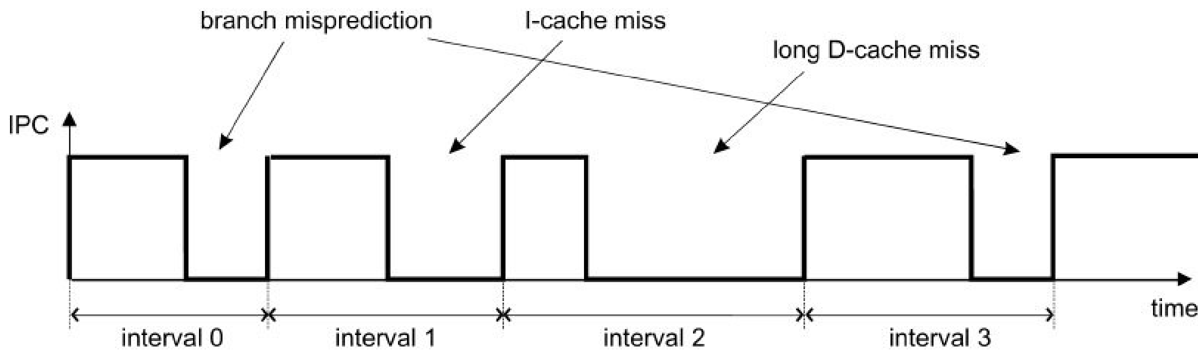

Figura 2 – Espaço de tempo dividido em intervalos de execução com perdas para análise de performance (extraído do artigo em revisão).

High-Performance Timing Simulation of Embedded Software

Artigo: J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "*High-performance timing simulation of embedded software*," in Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE, 2008, pp. 290-295.

Acadêmico: Leonardo Garcia Tampelini – **RA:** 098336

O presente artigo apresentada uma abordagem hibrida para a simulação de precisão de ciclos em softwares embarcados. Em sua proposta, os autores combinam técnicas de simulação e métodos analíticos de avaliação, propondo melhorias na velocidade da simulação sem ocasionar grande perda na precisão dos resultados.

A combinação de componentes mais simples para construção de novos sistemas é apresentada como uma tendência de mercado. Níveis de abstração - causados pela cooperação de componentes - implicam em mudanças no paradigma do projeto, pois a alteração de um único componente pode ocasionar grandes mudanças em nível global. Para avaliar tais situações, os autores enfatizam a necessidade de uma modelagem abrangente, a qual analise e simule a integração do sistema.

Para os autores, as análises estáticas de pior/melhor tempo de execução (WCET/BCET) oferecem resultados pessimistas quando o escopo considerado vai além dos blocos básicos. Este fator é agravado quando os efeitos de coerência de *cache* em arquiteturas *multicore* são considerados. Para reduzir a imprecisão (pessimismo) da analise estática WECT/BCET, a abordagem proposta aplica um esquema de *back-annotation* dos valores WCET/BCET (determinados estaticamente nos blocos básicos do código binários e gerados a partir do código-fonte em C). Além disso, impactos causados por arquiteturas distintas, como diferentes modos de previsão de desvios, poderão ser efetivamente considerados na abordagem proposta.

A abordagem proposta é composta pelas seguintes etapas: **1)** Dado à descrição do processador, traduzir o código binário (utilizando *cross-compiler)* para o processador especificado; **2)** Utilizar a ferramenta de *back-annotation* para ler o código objeto e a descrição do processador para a simulação; **3)** Considerar a descrição do processador para decodificar e traduzir o código objeto em uma representação intermediária (lista de objetos); **4)** A partir da lista de objetos, construir uma lista de blocos básicos do programa; **5)** Utilizar a descrição do *pipeline* do processador (informações capturadas na etapa 2) e calcular estatisticamente o número de ciclos utilizados por cada bloco básico; **6)** Encontrar correspondências entre o código-fonte C e o código binário; **7)** Inserir código para a geração de ciclos que contará os ciclos gastos pelos blocos básicos em sua execução (análise estática); **8)** Finalizar inserindo o código para a correção dinâmica da geração de ciclo. Apesar da arquitetura ser previamente conhecida, sua real influencia no número de ciclos gastos não é, pois certos parâmetros como *cache hit/miss* e previsão de desvios, não podem ser definidos estaticamente, enfatizando a importância da correção dinâmica.

Os testes comparativos foram efetuados utilizando dois filtros (FIR, ellip) e dois programas que fazem parte de rotinas de decodificação de áudio (DPCM, sub-bandas). O código objeto para o *Infineon Tricore* foi gerado em um compilador C. O código objeto foi utilizado para a geração do código *SystemC* anotado. Como referências para medidas foram utilizados o *Tricore TC20GP evaluation board* e o simulador do conjunto de instruções (ISS) Tricore. Foram utilizados dois tipos de anotações no código *SystemC*: **1º)** que gera os ciclos depois da execução de cada bloco básico e **2º)** que adiciona ciclos ao contador de ciclos quando for necessário (comunicação com HW).

Os testes mostram que comparando a velocidade de execução do *SystemC* com a velocidade de execução de um ISS convencional, a abordagem proposta pelos autores promove uma melhora de até 91% na velocidade. Além disso, também foram realizadas comparações da precisão de ciclo (*cycle-accuracy*): o desvio do contador de ciclos dos programas traduzidos (considerando questões de memória e previsão de desvio) em relação ao obtido pelo TC20GP *evaluation bord* variou de 4 a 7% e foi praticamente a mesma variação com relação ao ISS convencional.

Os resultados apresentados neste artigo são promissores, mostrando grande melhora no desempenho e precisão do modelo de software embarcado temporizado (*timed*). Esta abordagem possibilita uma execução rápida do código anotado, além do código *SystemC* gerado poder ser utilizado em ambientes de simulação *SystemC*. Analisar o código binário e o código fonte C pode ser encarada como uma desvantagem dessa abordagem, pois o compilador pode realizar modificações de estrutura e otimização, dificultando a identificação das correspondências, podendo ser necessário utilizar técnicas de recompilação para conseguir identifica-las.

A Method for Design of Impulse Bursts Noise Filters Optimized for FPGA Implementations

Zdenek Vasicek, Lukas Sekanina e Michal Bidlo, Designer, *Automation and Test in Europe* - DATE; Março, 2010;

www.date-conference.com/proceedings/PAPERS/2010/DATE10/PDFFILES/12.4_4.PDF

Aluna:Maíra Saboia da Silva (098338)

Filtragem de imagem é uma tarefa implementada por muitos sistemas embarcados baseados em FPGA. A qualidade da imagem filtrada influencia todo restante do processamento da imagem. Contudo, filtros com bom desempenho necessitam de mais área no FPGA que filtros padrões. Estudos recentes sobre alguns filtros revelam que algoritmos evolucionários (EA) podem gerar filtros que apresentem a mesma qualidade nos resultados; contudo, com metade do custo no FPGA.

O princípio básico sobre projetos de circuitos evolucionários é que os circuitos eletrônicos são construídos e otimizados por um algoritmo evolucionário com a finalidade de obter uma implementação que satisfaça as especificações dadas pelo projetista. Para obter um circuito candidato, uma configuração em um circuito configurável é criada. Essa configuração é analisada, e um novo circuito é projetado pela aplicação de um operador genético no circuito existente. Circuitos candidatos que tiverem mais parecidos com o projetado, tem maior probabilidade que seu material genético seja selecionado para os próximos circuitos candidatos. O processo de evolução é finalizado quando um circuito ótimo é projetado, ou quando um número máximo de iteração é atingido. Como os EA são processos estocástico, a qualidade do circuito resultante não é garantida no final da evolução.

Esse trabalho aborta um projeto evolucionário de eficientes filtros de área para remover ruídos do tipo *burst* que está comumente presente em imagens de sensoriamento remoto como como imagem de satélite. Ruído *bursts* é um tipo específico de ruído que é difícil de filtrar. Isso é devido por ambos o *pixel* central e seus vizinhos, estarem corrompidos. Então uma janela 5x5 é centrada em cada pixel, e os outros pixels que estão abaixo dessa janela são considerados vizinhos deste. A filtragem proposta, calcula para cada pixel um valor usando o correspondente pixel e alguns de seus vizinhos.

Uma estrutura FIFO (*first in, first out*) foi implementada para ler os valores dos pixels da memória da imagem. A FIFO é tipicamente implementada usando várias BRAMs que servem como *buffer* de linha. Cada BRAMs consegue armazenar 2048 bytes. Ou seja, um *buffer* composto por uma única BRAM é capaz de armazenar imagens com 2048 pixels por linha.

Todos componentes dos filtros são equipados com registradores para suportar filtragem com *pipeline*. O filtro foi implementado em descrição de VHDL usando *Precision Synthesis to Xilinx Vertex II Pro XC2VP50*. O filtro proposto ocupa 128 slices, pode operar a 242 MHz e possui latência igual a 6. Os experimentos foram conduzidos num *cluster* composto por 100 PCs (Pentium IV, 2.4 GHZ, 1GB RAM) usando *Sun Grid Engine (SGE)* que possibilita rodar até 100 experimentos independentes em paralelo.

Após a comparação entre o método proposto e outros métodos presentes na literatura, pode-se constatar que este método apresentou resultados médios de filtragem com alta qualidade nos resultados. Simultaneamente, o custo baixo associado reforça a constatação de que a implementação é favorável; pois, sistemas embarcados de baixo custo representam o alvo dessa aplicação.

An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness

Resumo de Artigo: S. Hong and H. Kim,"An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness", ISCA 2009, pp. 152-163

Giovani Chiachia, RA 098362, 25/04/2010

A importância das arquiteturas GPU tem crescido muito na era *multicore* em que nos encontramos. No entanto, a programação massiva de *threads* paralelas continua sendo um grande desafio para os engenheiros de software. Ainda mais difícil que a paralelização dos programas é o entendimento dos gargalos de desempenho desses programas nas arquiteturas GPU. As abordagens atuais baseiam-se no refinamento do programa por parte dos programadores através da exploração de cenários distintos em que as características da arquitetura são exploradas sem que haja um entendimento pleno do que está sendo feito.

Com o objetivo de melhorar a percepção sobre como os gargalos de desempenho afetam as aplicações em arquiteturas GPU, este trabalho propõe um modelo analítico simples para a estimativa do tempo de execução de programas massivamente paralelos. Uma das características-chave do modelo está no fato dele fundamentar-se no número máximo de requisições em paralelo à memória (designado "*Memory Warp Paralelism*", MWP) que o programa possibilita. Esta medida está relacionada com o número de *threads* que acessam simultaneamente a memória, com o grau de paralelismo oferecido pela memória e com a largura do barramento de memória da GPU. A idéia é que o tempo de execução das aplicações é dominado pela latência das instruções de memória.

Baseado no grau de MWP, o modelo possibilita estimar o custo das requisições à memória, o que, por sua vez, possibilita estimar o tempo total de execução do programa. Adicionalmente, ao oferecer o número estimado de ciclos por instrução (CPI) de um programa, o modelo também possibilita aos programadores e compiladores decidir quando é vantajoso aplicar determinadas otimizações.

Para validar a proposta, comparações entre o desempenho real e o previsto pelo modelo são realizadas em uma série de GPUs e programas distintos. Os experimentos foram realizados a partir de programas "*micro-benchmarks*" e de algumas aplicações do "*Merge Benchmarks*", todos escritos na linguagem de programação CUDA. Para os programas "micro-benchmarks", o erro absoluto médio obtido foi de 5.4% e para os programas "Merge Benchmarks" selecionados, o erro foi de 13.3%.

Apesar de algumas deficiências, tais como não considerar o custo de "*caches misses*" ou de instruções de desvio, acredita-se que este modelo possa oferecer uma orientação aos programadores sobre como eles devem melhorar suas aplicações. Até onde se sabe, não há outro modelo analítico para predizer em tempo de compilação o desempenho dos programas em GPU baseado apenas em informações estáticas sobre seu comportamento.

MO401 – Trabalho 1 – Resumo de Artigo

Multiprocessor System-on-Chip Designs with Active Memory Processors for Higher Memory Efficiency. Junhee Yoo, Sungjoo Yoo, Kiyoung Choi. Proceedings of the 46th Annual Design Automation Conference, July 26-31, 2009, San Francisco, California.

Autora: Flávia de Oliveira Santos. **RA: 100594**

A latência de acesso a memória e operações relacionadas são freqüentemente o gargalo de desempenho em aplicações paralelas. No artigo em questão, é apresentado o conceito de *active memory operations* (operações de memória ativa) que é uma transação de rede *on-chip* que opera baseada no micro código provido pelo engenheiro de *software*. A abordagem proposta tem por objetivo tentar reduzir o número de transações na rede *on-chip*, ao invés de reduzir a latência em si. Em resumo, o método proposto combina muitas operações de leitura/escrita simples em uma operação de alto nível chamada de operação de memória ativa.

O artigo apresenta também a implementação de um processador chamado AMP (*Active Memory Processor* - Processador de Memória Ativa) que está localizado próximo a memória e executa as operações de memória ativa. O AMP é um típico processador VLIW *in-order-issue*. Seu conjunto de instruções VLIW é de 64 bits e pode decodificar, em cada ciclo, um acesso a memória, uma computação de dados, uma computação de endereço, um desvio e uma operação de leitura/escrita na rede.

A arquitetura utilizada nos estudos de caso para validação da abordagem é composta de 36 blocos com 32 *cores* e 4 memórias. Cada *core* é um elemento de processamento (PE – *Processing Element*) e contém um processador RISC e numa memória local. Cada memória contém um processador de memória ativa.

Foram realizados estudos de caso utilizando três aplicações reais: *Fast Fourier Transform* (FFT), codificador JPEG paralelizado e indexação de textos para mineração de dados. Em cada estudo de caso foi verificado o desempenho para um caso base que utiliza apenas os PEs e um caso melhorado que utiliza o AMP.

No exemplo do FFT, quando o número de processadores é 32, o tempo de execução do caso melhorado diminui 25,6% comparado ao caso base, o que traduz para um aumento de desempenho de 34,3%. Isso é devido à diminuição de transações de acesso de dados que caíram 30,6% reduzindo assim o tráfego na rede. Houve, entretanto, um aumento no número de transações para carregar a *cache* de instruções dos PEs de 37% devido ao código adicional para utilização do AMP.

No exemplo do codificador JPEG, para o caso base, houve uma saturação na melhora do desempenho por volta de 12 processadores devido à presença de um *lock* no *buffer* de armazenamento. O caso melhorado não mostra essa saturação, visto que o tempo de execução no AMP é bem menor comparado ao caso base. Como resultado, houve uma melhoria de 198,7% no desempenho do caso melhorado para o caso de 32 processadores.

Já no exemplo da indexação de textos, o caso base requer no mínimo 10 acessos à memória para inserir um valor na tabela da aplicação enquanto que o caso melhorado faz o mesmo em apenas uma transação. Em termos de número de palavras processadas por segundo, o caso base tem seu desempenho saturado por volta de 1.000K palavras por segundo enquanto que o caso melhorado é capaz de processar até 7.000K palavras por segundo quando todos os 32 processadores são utilizados. Isso resultou em uma melhoria de 618%.

O artigo em questão apresentou o conceito de operações de memória ativa e um processador de memória ativa como implementação. Resultados experimentais mostraram que a abordagem proposta pode melhorar o desempenho em cerca de 34,3% a 618% para aplicações reais com o custo de um esforço adicional de *design* e um aumento de área moderado na interface de rede do bloco de memória.

MO401 - Trabalho 1 – Resumo de artigo

Prototyping Pipelined Applications on a Heterogeneous FPGA Multiprocessor Virtual Platform

Tumeo, A., Branca, M., Camerini, L., Ceriani, M., Monchiero, M., Palermo, G., Ferrandi, F., and Sciuto, D.; DAC 2009.

Resumido por **Marcelo Fontes Santana** (RA 100602)

O sucesso dos processadores multi-cores motivou a investigação de novas metodologias de programação para que eles sejam utilizados no seu desempenho máximo. Existem várias técnicas para explorar o ILP(*Instruction-level parallelism*) máximo das aplicações, dentre elas está a estruturação e utilização do *pipeline* em sistemas que podem ser utilizados processamentos paralelos independentes e em fluxo, como por exemplo os de áudio e vídeo.

O artigo resumido apresenta uma plataforma virtual para que sejam testadas aplicações às quais se pretende dar uma execução em pipeline, facilitando assim seus ajustes e, consequente, sua implementação mais rápida. Esta plataforma pode ter diferentes níveis de heterogeneidade, pois a arquitetura desenvolvida permite que o desenvolvedor customize cada elemento de processamento, chamado no artigo de MB ou *MicroBlaze,* para que funcione de acordo com a aplicação que ele deseja desenvolver. A plataforma integra software (*Micro-Kernel*) e hardware com diferentes conjuntos de instruções. Ela foi desenvolvida com o kit de desenvolvimento embutido (*EDK*) v9.1 da *Xilinx,* utilizando a FPGA*(Field Programming Gate Array) Xilinx Virtex-II Pro XC 2VP30.* A aplicação utiliza os dois cores *IBM Power PC 405* que a FPGA possui integrados. A comunicação entre as unidades é gerenciada por um dos processadores através de interrupções. Além disso, Há um desacoplamento para redução dos *stalls*, obtido pela alocação de porções simples da memória externa, para os dados de cada unidade de processamento, e de duas memórias interna para cada bloco com dados e instruções.

Para utilizar a plataforma é preciso: **1.** Configurar os parâmetros do *Micro-Kernel* (software), deve ser definido quais elementos (unidades de processamento disponíveis) serão utilizados e qual a tarefa realizada por cada um através das suas instruções; **2.** Definir as estruturas de dados – dados que transitam entre as unidades de processamento; **3.** Configuração do fluxo de dados - o fluxo de dados é definido, especificando se há compartilhamento dos dados ou sequenciamento e, quando necessário, quais as transformações devem ser feitas nas estruturas para que os dados transitem entre as unidades.

Para mostrar como deve ser utilizado o framework são utilizados três estudos de caso:1. ADPCM-CRC - utilizado em telecomunicações, uma extensão da codificação PCM (*Pulse Code Modulation*); 2. Modulador de Frequência; 3. Compressão JPEG no modo *baseline*. Não houve a intenção de mostrar soluções de alta performance para estas aplicações. Utilizando estes casos, fez-se uma comparação entre eles, indicando quais algumas as informações que podem ser obtidas no uso da plataforma, tais como o desbalanceamento dos estágios do pipeline. O ADPCM-CRC apresentou o melhor resultado por causa do melhor balanceamento de seus estágios, o pior resultado foi o Modulador de Frequência, que sobrecarregava uma das unidades de processamento do *pipeline*.

Desta forma, o artigo aqui resumido expôs uma plataforma multiprocessadora heterogênea para prototipagem de aplicações usando *pipeline*. A plataforma traz uma grande contribuição, pois ajuda o desenvolvedor a explorar diversas formas de implementação em pipeline para a aplicação de uma forma virtual e rápida. Assim, a plataforma ajuda a validar uma arquitetura determinando quais os estágios do pipeline devem ser melhorados para que os estágios sejam bem balanceados. Desta forma, o desempenho pode ser aumentado significativamente, como comprovado pelos resultados dos estudos de casos.

UNICAMP – Maio de 2010

MO401 - Trabalho 1 – Resumo de artigo

Generating Test Programs to Cover Pepiline Interactions
Dang, T. N., Roychoudhury, A., Mitra, T. and Mishra, P; DAC 2009

Resumido por **Camila Satsu de Amorim Yokoigawa** (RA 107006)

Os sistemas embarcados estão cada vez mais complexos devido ao aumento da demanda por altos requisitos computacionais que representam a inclusão de complexidade acompanhada de características de alta performance, como caches e pipelines, dentro da arquitetura de computadores. Porém, interações não triviais entre essas caracteristicas são a causa mais comum de erros na implementação de processadores.

Neste contexto, o artigo *Generating Test Programs to Cover Pepiline Interactions*, apresentado em 2009 no 46º evento DAC, *Design Automation Conference*, apresenta um modelo de especificação de arquitetura em alto nível baseada na comunicação estendida de máquinas de estado finito, ou FSM, *Finite State Machine*, de processadores para a geração automática de conjuntos de testes que possibilitam a simulação de todas as possibilidades de interação de pipeline. Este método de geração de testes reduz o tamanho dos conjuntos de testes para validação comparada à outras abordagens de geração de testes, reduzindo drasticamente os esforços de validação, que gastam uma significativa parte do processo de design.

Para gerar automaticamente os conjuntos de teste a arquitetura é modelada por meio de modelos de operação de máquina de estado de dois níveis: o nível operacional e o nível de hardware, que, respectivamente, descreve os movimentos de instruções através dos estágios da pipeline e recursos de hardware, como janelas de instruções e unidades funcionais. É definida a métrica de cobertura que assegura que todos os componentes de pipeline sejam usados, capturando-se todos os cenários comportamentais dentro do pacote de teste por meio de modelos de pipelines dados como uma coleção de EFSM, Extended Finite State Machine – maquina de estado finita com variáveis, para a construção de uma FSM global. É usado O Algoritmo 1 para gerar o conjunto de testes *on-the-fly*.

A área de geração de teste para arquiteturas é muito bem estudada, não faltando trabalhos relacionados como a ferramente Genesys desenvolvida pela IBM, técnicas de cobertura de testes e uso de model checkers para gerar programas de testes para processadores de pipeline.

Entretanto, essas abordagens falham na cobertura das possibilidades de interações de pipeline e na automatização do trabalho de validação, pontos fortes da abordagem trabalhada pelos autores, que garante 100% de cobertura das interações de pipeline com confiabilidade, produzindo conjunto de testes executáves pequenos, com menos de 1% do tamanho dos conjuntos de testes produzidos pelas demais abordagens.

MO401 - Trabalho 1 - Resumo de artigo

Accurate Temperature Estimation Using Noisy Thermal Sensors

Yufu Zhang Srivastava, A. - 2009 - DAC

Resumido por **Sergio Ricardo Souza Leal de Queiroz (RA 107070)**

Atualmente, circuitos integrados multi-core contam com sensores térmicos para medir em tempo de execução a temperatura do silício. Com isso, se a temperatura ultrapassa um limiar, então deve ser reduzida até os níveis aceitáveis. Estes sensores são assumidos como precisos depois de sua instalação e calibragem.

Porém, este artigo resumido mostra que as informações produzidas pelos sensores térmicos desconsideram as pertubações inerentes à leitura, causando problemas nas estimativas da temperatura real do silício. Tais ruídos ocorrem devido a aleatoriedade de fabricação, flutuações de energia, e outros fatores.

Com base neste argumento, os autores do artigo apresentam um método estatístico para corrigir o problema através de duas técnicas: uma para sensores de predição única, e outra para multi-sensores de predição. Essa última considera os valores gerados simultaneamente, e ao mesmo tempo explora as correlações entre eles.

Os resultados mostram que o erro pode ser reduzido cerca de 67% em comparação aos sensores de processadores considerados precisos, i.é., sem ruídos.

Os experimentos realizados usaram estratégias diferentes para sensores de predição única e multi-sensores de predição.

Na estimativa do sensor de predição única, o teste usou uma simulação agressiva num processador "out-of-order" com pipeline de profundidade 8 e janela de 128 instruções, caches de nível 1 (ambos são instruções e dados) de 32 Kb 4-Associativo por conjunto. Todos os caches na hierarquia usam a política de substituição de bloco de 64 bytes. Também foi utilizado todos os SPEC CPU 2000 (benchmark suite 1) compilados com parâmetros defaults. Para cada um dos benchmarks, 250 Mb de instruções, pulando o start-up.

Para a estimativa dos multi-sensores, o mesmo teste usou uma arquitetura VLSI, que usa múltiplos sensores para estimar o estado térmico de diferentes áreas afins. Também foi considerada a relação espacial entre os sensores. Se a distância entre os sensores for pequena, existirá influência mútua, e se forem distantes um do outro, a relação é baixa.

Considerando que os ruídos foram causados no experimento pela variação da voltagem, várias estratégias foram utilizadas em ambos os casos, e em todos os casos o método estatístico proposto foi eficaz.

Reducing Peak Power with a Table-Driven Adaptive Processor Core

Vasileios Kontorinis – Amirali Shayan – Rakesh Kumar – Dean M. Tullsen

MICRO – 2009

Autor do Resumo: Davi de Andrade Lima Castro – **RA**: 107072

A maioria das técnicas de redução de consumo atualmente empregadas nos processadores de alto desempenho buscam apenas reduzir o **consumo médio** de potência. Embora esta redução seja desejável e traga benefícios como o barateamento das soluções de resfriamento (além é claro da própria redução do consumo de energia), o **consumo de pico** impacta diretamente em vários aspectos, tais como a própria implementação física do circuito integrado, requerimentos de *packaging* e custos da fonte de alimentação.

A razão disto está no fato de que muitas decisões devem ser tomadas para cobrir um pior caso de consumo, embora este raramente ocorra em operações normais. Existe então uma parcela considerável de *over-design* que acarreta em maiores custos.

O artigo em questão propõe uma arquitetura altamente adaptativa, com componentes configuráveis centralizadamente controlados. É explorada uma observação experimental importante que mostra ser possível configurar minimamente alguns componentes sem prejudicar tanto o desempenho de uma aplicação, desde que os componentes mais importantes (*bottleneck*) para a aplicação estejam maximamente configurados.

Como o controle é **centralizado**, é possível então **garantir** um valor limite de **consumo de pico** escolhendo apenas certas combinações de configurações de forma a nunca ultrapassar este valor limite – apenas um certo conjunto dos componentes estará maximamente configurado. É esta garantia que possibilita tomar decisões baseadas no pico máximo escolhido (70%, 75% ou 80% do pico normal) durante o *design*.

A arquitetura é estruturada em três elementos: **componentes configuráveis, memória com as configurações possíveis** e **controle adaptativo**.

Exemplos de componentes configuráveis são: *cache*, *re-order buffer*, unidades de execução e outros. Eles são responsáveis por em torno de 50% do consumo de pico, porém como não é possível desligar completamente certos componentes, o valor mínimo possível encontrado para o valor limite de consumo de pico foi 70%.

O controle adaptativo é necessário porque cada aplicação possui sua própria configuração-ótima. As tarefas deste controle são três: *decidir quando mudar de configuração, qual configuração usar* (dentre as que se encontram na memória) e *avaliar a configuração escolhida*. O artigo propõe e avalia alguns algoritmos e os de melhor resultado utilizam *feedback* da performance atual para encontrar a configuração-ótima.

Os resultados obtidos foram muito positivos. Para a máxima redução de pico possível, 30%, a perda em desempenho é de 10%. Para uma redução de 25% do pico, temos uma penalidade de 5% no desempenho e para este caso estima-se uma redução de 5.3% da área do *die* (para a mesma variação na tensão de alimentação) ou uma redução de 26% na variação da tensão (para a mesma área). A redução de área é conseqüência direta da redução do número (ou capacitância) dos capacitores de desacoplamento, e estes estão relacionados com o pico do consumo de corrente.

Além disto, como os componentes configuráveis utilizam *power gating* (desligamento parcial de energia), a arquitetura também reduz o consumo médio de potência.

O *overhead* introduzido pelas técnicas é pequeno, visto que a memória de configuração é acessada em uma pequena parcela do tempo e possui tamanho em torno de apenas 400 bytes, e a lógica do controle adaptativo não faz parte do caminho crítico da arquitetura.

Como desvantagens têm-se o aumento da complexidade do teste e verificação do processador.

Architectural Core Salvaging in a Multi-Core Processor for Hard-Error Tolerance

M. D. Powell, A. Biswas, S. Gupta and S. S. Mukherjee, *Architectural Core Salvaging in a Multi-Core Processor for Hard-Error Tolerance*, In the Proceedings of the 36th International Symposium on Computer Architecture (ISCA), June 2009.

Autor: Hilário Viana Bacellar R.A: 107077 Disciplina: MO401

Resumo

O artigo "Architectural Core Salvaging in a Multi-Core Processor for Hard-Error Tolerance" tem em seu principal tema minimizar as alterações nos núcleos de um processador que venha a falhar em determinadas operações.

O autor do artigo explica que as CPU's atuais de alto desempenho possuem vários núcleos para o processamento de instruções e que mesmo assim, em caso de falha de um desses núcleos, o desempenho da CPU fica comprometido. Desta forma, é crucial um estudo detalhado sobre a vulnerabilidade dos núcleos em caso de falhas de um ou mais.

Existem algumas técnicas para a detecção de falhas, elas podem ser dívidas em *manufacturing time* detection e *manufacturing test*.

O *manufacturing time* se concentra na identificação de partes defeituosas, entretanto, uma adequação dessa técnica é possível ser expandido para isolar uma parte defeituosa do sistema e assim criar uma redundância de cobertura contra falhas.

O *manufacturing test* se concentra na execução de uma varredura no sistema e mecanismos de depuração baseados em granulosidade fina a fim de encontrar falhas conhecidas na micro-arquitetura.

Para implementar o *core salvaging in a multi-core processor* foram levados em consideração alguns pontos chaves:

- *Minimal Core Changes* – o objetivo desse ponto é minimizar as alterações sofridas no núcleo.
- *Migration and Overhead* – o objetivo desse ponto é a detecção de carga dos núcleos.
- *Operating System Transparency* – o objetivo é manter a técnica totalmente baseada em hardware, assim o ponto chave é deixar o mais transparente possível ao SO.

Após a implementação, também foi descrito os passos para a otimização dos recursos:

- *Migration Policy* - O objetivo dessa política de migração é garantir que os segmentos sejam migrados de um núcleo para o outro sem comprometer o sistema.
- *Migrações Triggering* – O objetivo desse passo é acelerar o processo de migração e decodificação de segmentos.
- *Fall-back to Core Disabling* – O objetivo é manter uma janela de tempo para evitar uma migração excessiva entre os núcleos.
- *Running Less Than the Max Number of Threads* – O objetivo é manter o núcleo que ocorreu a falha em modo de espera o maior tempo possível.

Ao termino do artigo o autor mostra que *salvaging core* tem potencial para cobrir 86% da área da unidade de execução e que mostrou provas de conceito que abrange 46% da área da unidade de execução, sendo assim esses 46% equivale à área dos núcleos vulneráveis.

AnySP: Anytime Anywhere Anyway Signal Processing

(título do artigo)

Citação bibliográfica do artigo

Autores: Mark Woh, Sangwon Seo, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti e Krisztián Flautner.
Publicado por IEEE Computer Society (IEEE MICRO) – Janeiro/Fevereiro 2010 – páginas 81-91
Conferência: ISCA'09

Autor do resumo: Luciana Bulgarelli Carvalho – RA:981561

De acordo com o artigo, as soluções utilizadas na terceira geração (3G) de tecnologia wireless não são eficientes o suficiente para serem adaptadas para a tecnologia 4G. Por esta razão, o artigo propõe uma arquitetura de alta performance para o processamento de sinais móveis: a arquitetura AnySP. Esta arquitetura é definida a partir das características dos principais algoritmos usados na comunicação 4G wireless e na decodificação de vídeo de alta definição.

A partir da análise destes algoritmos, as seguintes características foram incorporadas ao projeto da arquitetura AnySP:

- Utilização de SIMD (single instruction, multiple data) de largura 8 e a possibilidade de configurar um grupo de SIMD de largura 8 para "criar" um SIMD de largura 16, 32 ou 64.
- Paralelismo a nível de thread (thread-level parallelism – TLP) para a execução paralela de SIMDs de largura 8, 16 ou 32.
- Redução do número de acessos ao register file principal (16 registradores) de um SIMD com o objetivo de diminuir o consumo de potência devido a estes acessos. A arquitetura permite ao programador utilizar, através de instruções especificas, registradores internos (4 registradores), que não são gravados no register file principal, ao invés dos registradores do register file principal.

Para facilitar a definição da arquitetura AnySP (figura 2 do artigo) considere uma "linha" da arquitetura como um SIMD de largura 8. A arquitetura AnySP é composta por oito destas "linhas". Cada "linha" tem uma unidade funcional flexível (flexible functional unit – FFU) de largura 8 que é conectada ao seu register file (16 registradores do register file principal e 4 registradores internos). Cada uma destas unidades funcionais flexíveis de largura 8 é composta por quatro unidades funcionais flexíveis de largura 2 (figura 3 do artigo) e estas, por sua vez, contém um multiplicador, uma unidade de aritmética lógica (ALU) e um somador.

O bloco swizzle network executa as operações configuradas através de uma SRAM. Estas operações podem ser modificada após a fabricação do chip. Um exemplo de operação que pode ser realizada por este bloco é a permutação de dados.

O bloco multiple output adder tree realiza somas parciais de 4, 8, 16, 32 ou 64 elementos, faz a redução dos elementos a um valor escalar e armazena o resultado em um buffer temporário para evitar acessos de leitura e escrita ao register file principal.

A memória local da arquitetura AnySP contém 16 bancos e cada banco armazena oito "linhas". Cada grupo de oito "linhas" tem sua unidade de geração de endereço (address generation unit – AGU) dedicada.

O artigo garante que o requisito de 1.000 MOPS/mW é alcançado se a arquitetura AnySP for sintetizada com uma tecnologia de 45 nm. Entretanto, o artigo apresenta apenas os resultados quando a arquitetura é sintetizada com a tecnologia TSMC – 90 nm, e neste caso o requisito de 1.000 MOPS/mW não é alcançado.

Uma constatação importante do artigo é que as unidades funcionais flexíveis são as responsáveis pela maior parcela do consumo de potência da arquitetura AnySP.