Implementação e Avaliação do Controlador de Protocolos do NCP_2 *

Rodrigo dos Santos, Malena Hor-Meyll, Marcelo De Maria,
Raquel Pinto, Lauro Whately, Ricardo Bianchini e Claudio L. Amorim

COPPE Sistemas/UFRJ
Cx. Postal 68511 CEP 21945-970 - Rio de Janeiro - Brasil

rodrigo@cos.ufrj.br

Resumo

O computador paralelo NCP_2 em desenvolvimento na COPPE/UFRJ provê suporte de *hardware* para a implementação de sistemas de memória compartilhada distribuída eficientes. O primeiro protótipo do NCP_2 encontra-se em fase de depuração no momento, mas já nos permite avaliar os componentes principais do *hardware* que projetamos. Experimentos preliminares com o protótipo identificaram que nosso *hardware* não explora o paralelismo intrínseco de operações com *diffs* extensivamente. Assim, nesse trabalho apresentamos os detalhes do *hardware* do protótipo e uma proposta de *hardware* adicional especializado para o tratamento de *diffs*: o Gerente de *Diff* Dinâmico (GDD). A arquitetura do GDD baseia-se em uma estrutura de pipeline com fluxo de dados recursivo e sua implementação em FPGAs está sendo atualmente estudada. Nesse trabalho apresentamos também um modelo analítico que sugere que o GDD fornecerá ganhos de desempenho significativos ao NCP_2. Mais especificamente, em comparação aos tempos atuais do controlador de protocolos do NCP_2, o GDD permitirá ganhos de desempenho entre 37 e 76%. Por isso, concluímos que na próxima versão do NCP_2, o controlador de protocolos deve ser composto pela dupla: Processador RISC + GDD.

Abstract

The NCP_2 parallel computer under development at COPPE/UFRJ provides hardware support for the implementation of efficient distributed shared-memory systems. The first prototype of the NCP_2 is currently undergoing debugging, but already allows us to evaluate our design of its main components. Preliminary experiments with the prototype have shown that the hardware we designed does not fully exploit the parallelism available in diff-related operations. Hence, in this paper we present the details of the hardware of our prototype and propose a novel piece of hardware that can exploit this parallelism more effectively: the Gerente de *Diff* Dinâmico (GDD). The architecture of the GDD is based on a pipeline structure with recursive data flow. We present an analytical model that suggest that the GDD will provide significant gains in the overall performance of the NCP_2. More specifically, the GDD will provide performance improvements varying from 37 to 76% with respect to the protocol controller of our first prototype. Thus, the present work demonstrates the effectiveness of the GDD in improving the performance of the NCP_2, indicating that the basic protocol controller should be extended to include both the current RISC Processor and the GDD.

*Esta pesquisa foi financiada pela FINEP, CNPq e FAPERJ.

1 Introdução

A nova geração de máquinas paralelas da COPPE/UFRJ [ABS+96], representada pelo sistema NCP_2 [SIIMM+96, WPS+96], baseia-se em três objetivos principais: alto desempenho, baixo custo e facilidade de programação. O alto desempenho e o baixo custo são alcançados através do uso de estações de trabalho e componentes de rede comerciais, num esquema conhecido como NOW (Network of Workstations). Tais componentes de *hardware* são suficientes para que aplicações baseadas em passagem de mensagens sejam implementadas diretamente no NCP_2. No entanto, para que a programação de computadores paralelos se torne mais simples é necessário prover um modelo de programação de memória compartilhada. O NCP_2 atinge esse objetivo sem perder em desempenho através de seu suporte de *hardware* barato para sistemas de memória compartilhada distribuída (DSM) baseados em *software* [ACD+96, CBZ91, BZS93, MB97, SBA97].

O primeiro protótipo do NCP_2 usa máquinas do tipo PC (Personal Computer), baseadas no processador PowerPC 604 [Mot94], interligadas por uma rede Myrinet [BCF+95]. Além desses componentes comerciais, duas placas foram desenvolvidas especialmente para minimizar os overheads do protocolo de DSM: a placa do Controlador de Protocolo (CP), que está conectada ao barramento PCI de cada nó de processamento; e uma placa para monitorar o barramento do processador, ligada diretamente ao cache L2 do PowerPC.

O protótipo do NCP_2 encontra-se em fase de depuração no momento, mas já nos permite avaliar os componentes principais do *hardware* que projetamos. Experimentos preliminares com o protótipo identificaram que nosso CP não explora o paralelismo intrínseco de operações de manutenção de coerência extensivamente. A razão disso é que, no primeiro protótipo da nossa máquina, a maior parte do processamento associado a operações de manutenção de coerência é executada por um processador RISC, o Intel 960 (i960), contido na placa do CP.

Nesse trabalho apresentamos os detalhes do *hardware* do nosso protótipo. Além disso, como operações de manutenção de coerência são de extrema importância para o bom desempenho de sistemas de memória compartilhada distribuída [BKP+96], esse trabalho propõe um *hardware* adicional especializado para o tratamento dessas operações: o Gerente de *Diff* Dinâmico (GDD). A arquitetura do GDD baseia-se em uma estrutura de pipeline com fluxo de dados recursivo e sua implementação em FPGAs está sendo atualmente estudada.

A avaliação do GDD é feita através de um modelo analítico que sugere que o *hardware* que projetamos fornecerá ganhos de desempenho significativos ao NCP_2. Mais especificamente, em comparação aos tempos atuais do controlador de protocolos do NCP_2, o GDD permitirá ganhos de desempenho entre 37 e 76%. Devido a esses resultados e a necessidade de manter a flexibilidade de programação do CP, concluímos que na próxima versão do NCP_2, o CP deve ser composto pela dupla: i960 + GDD.

O resto desse artigo está organizado da seguinte forma. Na seção 2 apresentamos alguns conceitos básicos de protocolos de DSM. Na seção 3 descrevemos detalhadamente a implementação do *hardware* da primeira versão do NCP_2. A arquitetura do GDD é discutida na seção 4 e os resultados obtidos são apresentados na seção 5. Por fim, concluímos o artigo na seção 6.

2 Protocolos de DSM

Vários protocolos de DSM exploram o *hardware* de memória virtual dos nós do sistema para garantir a coerência dos dados compartilhados. Tipicamente, a unidade básica de coerência empregada é a de uma página.

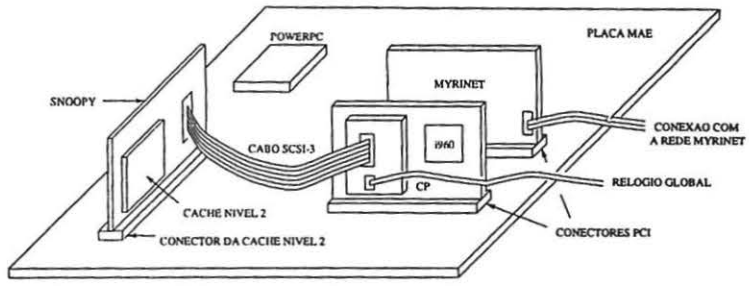

Figura 1: O *Hardware* do NCP_2.

Com a finalidade de diminuir o *overhead* de compartilhamento de dados entre os nós do sistema, os protocolos modernos de DSM utilizam duas técnicas básicas: consistência relaxada [Mos93, KCZ92] e múltiplos escritores concorrentes [ACDZ97, Kel96]. A consistência relaxada só garante a consistência da memória em pontos de sincronização, enquanto que a técnica de múltiplos escritores permite que nós distintos de uma máquina paralela possam escrever uma mesma página simultaneamente. A técnica de múltiplos escritores leva tais sistemas a utilizarem os mecanismos de *twinning* e *diffing* para detectar as modificações realizadas em páginas.

Os mecanismos de *twinning* e *diffing* funcionam da seguinte forma. As páginas locais de um nó são inicialmente protegidas contra escrita. Na primeira falta de escrita numa página, o processador gera uma cópia exata da página, chamada de *twin*, e desprotege a página contra escrita. Quando é necessário determinar as modificações feitas na página, i.e. quando outro nó precisa da última versão da página, o processador executa uma comparação palavra-por-palavra do conteúdo atual da página e o seu *twin*, numa operação chamada geração de *diff*. Esse *diff* pode ser então enviado ao outro nó, o qual aplica essas modificações a sua versão local da página.

Trabalhos anteriores [ABS+96, BKP+96, WPS+96] mostram que grande parte do *overhead* dos protocolos modernos de DSM em ambientes NOW é proveniente de operações com *diffs* e *twins*. Protocolos com múltiplos escritores encontram forte suporte de *hardware* no NCP_2, já que o nosso sistema elimina totalmente a manipulação de *twins* e otimiza operações de geração e aplicação de *diffs*. A seção seguinte descreve o *hardware* do NCP_2 em detalhe.

3 O *Hardware* do NCP_2

Quatro módulos distintos compõem o *hardware* do primeiro protótipo do NCP_2, como pode ser observado na Figura 1:

- Uma máquina mono-processada comercial (um PC)
- Uma placa de rede comercial (Myrinet)
- O Controlador de Protocolo (CP)
- O Módulo Snoopy ligado ao conector de Cache L2 do PC

Cada nó PC utilizado possui um processador PowerPC 604, rodando a uma frequência de 100 MHz, com 32 MBytes de memória principal (DRAM) e 256 KBytes de Cache L2. O barramento de I/O é o PCI, barramento de 32 bits que garante uma banda máxima de

transferência de dados de 132 Mbytes/s. A placa de rede comercial Myrinet com desempenho de comunicação de 1.28 Gbits/s é conectada ao barramento PCI. Essas placas se ligam ao *switch* Myrinet, com capacidade para até 8 nós conectados diretamente; máquinas maiores podem ser contruídas conectando switches em cascata. Nosso CP se conecta ao barramento PCI e ao barramento do processador, através do módulo Snoopy. É graças ao uso das placas CP e Snoopy que o NCP_2 consegue eliminar *twins* e otimizar o uso de *diffs*. Descrevemos o módulo Snoopy e o CP nas duas próximas sub-seções.

3.1 Módulo Snoopy

Este módulo do NCP_2 foi desenvolvido especialmente para fazer a monitoração do barramento de memória do processador principal do PC. Nesta primeira versão do NCP_2, o módulo Snoopy está conectado a Cache L2 do PC. Os sinais de controle do processador principal que chegam ao conector da Cache L2 são usados para a detecção dos acessos de escrita do processador.

Mais especificamente, a lógica presente no módulo Snoopy observa e qualifica os acessos que são realizados no barramento do PowerPC. Os acessos de escrita qualificados, i.e. as escritas a dados compartilhados, são transferidos para o módulo CP. Esta transferência é feita através de um *flat cable* padrão SCSI-3 (20 *Mega Transfers/s*).

Além dos circuitos necessários para interface com o barramento do PowerPC e com o *flat cable* padrão SCSI-3, a lógica que controla a qualificação dos acessos está implementada numa EPLD Altera de 1200 portas, EPM7032.

3.2 Controlador de Protocolo

O CP é uma placa PCI baseada no processador Intel 80960 (i960 [Int95]) com as seguintes características (veja figura 2):

- Processador Intel 80960IIA 33MHz
- 2 MBytes de memória DRAM (SIMM expansível até 32 MBytes)
- Interface PCI através do chip PLX9060
- 8 Mbits de memória SRAM (Memória de Vetores)
- Lógica de conexão com o módulo Snoopy (EPLD EPM7256)
- Lógica do relógio global (EPLD EPM7128)

O i960 é um RISC superescalar cuja frequência de operação é de 33MHz e cujo barramento de dados externo é de 32 bits. Sua arquitetura superescalar lhe permite a execução de até 3 instruções por ciclo. O i960 possui 16 KBytes de cache de instruções e 8 KBytes de cache de dados. Em conjunto com a memória DRAM, o processador fornece flexibilidade e poder de processamento para a implementação das tarefas do CP: envio e recepção de mensagem; envio e pedido de página; envio e pedido de *diff*; e geração e aplicação de *diff*. Dessa forma, essas operações decorrentes do protocolo de DSM podem ser executadas em paralelo com a computação útil do processador PowerPC.

A interface entre o barramento PCI e o barramento local do CP é feita pelo chip PLX9060. O PLX9060 suporta transferências de dados no PCI de até 132 MBytes/s onde FIFOs internas asseguram a transferência de dados em modo *burst*. O chip pode gerar interrupções em ambos os lados (PCI e barramento local do CP) baseadas em diversas fontes. Dois canais de DMA tipo "chaining DMA" podem ser programados (tanto pelo

lado PCI quanto pelo barramento local). Dessa forma, a transferência de dados do PCI para o barramento local e/ou vice-versa podem transcorrer sem o uso dos processadores principais.

A Memória de Vetores armazena sob a forma de um vetor de bits o endereço das palavras de dados que foram alteradas pelo processador principal do PC. A cada escrita executada pelo processador, um bit da Memória de Vetores é marcado. O endereço do bit marcado na Memória de Vetores corresponde ao endereço da palavra escrita na memória principal do PC. As escritas no barramento são monitoradas pelo módulo Snoopy e a informação de escrita atinge a Memória de Vetores através do cabo SCSI-3.

A Memória de Vetores está organizada em dois bancos de 128 K × 32 bits de memória SRAM, com capacidade de mapear 32 Mbytes de memória principal do PC.

A Memória de Vetores pode ser acessada tanto pelo barramento interno do CP quanto pela placa Snoopy, através do cabo SCSI-3. A coordenação dos acessos a Memória de Vetores é efetuada pela Lógica Snoopy. Esta lógica está implementada em uma EPLD Altera de 10000 portas, a EPM7256. Além disso, a lógica de Snoopy decodifica a informação proveniente da placa Snoopy para realizar as modificações necessárias na Memória de Vetores.

Por fim, o módulo de relógio global provê uma base de tempo igual para todos os CPs do NCP_2. Este módulo pode funcionar em dois modos: transmissor ou receptor. Esta configuração é feita por *hardware* através de um jumper do CP. No NCP_2, um dos nós da máquina está setado em modo transmissor, enquanto todos os nós restantes estão em modo receptor. O transmissor gera um sinal de relógio global para os nós receptores. A frequência do relógo global é configurada por software e pode variar de 8 MHz a 1 MHz. O relógio global serve de base de tempo para contadores que são utilizados na monitoração de desempenho da maquina NCP_2.

A lógica do Relógio Global está implementada em uma EPLD Altera de 5000 portas, a EPM7128. O relógio global é transferido entre os CPs por um par diferencial, cujo alcance máximo é de 10 metros.

3.3 Geração e Aplicação Dinâmica de *Diffs*

Graças ao esquema proporcionado pela monitoração do barramento do PowerPC (placa Snoopy) e pela Memória de Vetores + i960 (placa CP), a geração do *diff* de uma página pode ser obtida sem a necessidade de comparações com seu *twin*. A medida que o processador principal altera os dados de uma página, automaticamente a informação relativa aos endereços dos dados modificados é transferida da Placa Snoopy para o CP, para então ser armazenada na Memória de Vetores. Dessa forma, cada *diff* é armazenado dinamicamente, num processo que chamamos *diff* dinâmico. Mais especificamente, o *diff* de uma página é gerado através das seguintes tarefas:

(G.I) O i960 efetua a leitura dos bits da Memória de Vetores a partir do endereço correspondente ao endereço da página. Cada 32 bits da Memória de Vetores denominamos de máscara. As máscaras lidas Memória de Vetores são então armazenadas no começo da estrutura de *diff*.

(G.II) O i960 processa cada máscara proveniente da Memória de Vetores e extrai os endereços dos dados que foram modificados pelo processador principal.

(G.III) Os dados modificados são então lidos da memória principal do PC e armazenados na estrutura de *diff*, logo após as máscaras (que foram armazenadas em (G.I)). Dessa forma, as modificações ocorridas em uma página são relatadas sob a forma da estrutura de *diff*.

Observe que na tarefa (G.I), os bits da Memória de Vetores são resetados a medida que a leitura dos mesmos é executada. Com isso, a Memória de Vetores está pronta para

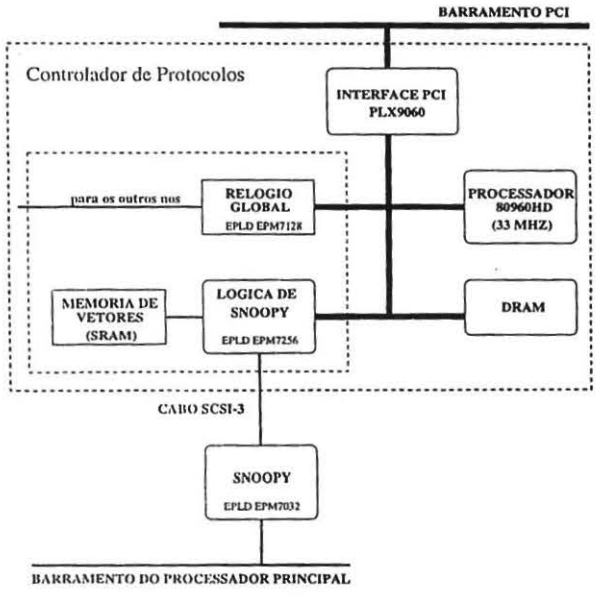

Figura 2: O Controlador de Protocolos.

armazenar informações sobre os endereços de futuras modificações realizadas sob a página em questão. Este reset da Memória de Vetores é realizado pela lógica Snoopy.

Com o *diff* gerado, a sua aplicação pode ser executada através de tarefas similares as da geração de *diff*:

(A.I) O i960 efetua a leitura das máscaras armazenadas no começo da estrutura de *diff*.

(A.II) O i960 processa cada máscara e extrai os endereços dos dados que devem ser atualizados.

(A.III) Os dados que estão presentes na estrutura de *diff* são então lidos e escritos na memória principal do PC.

4 O Gerente de Diff Dinâmico

Testes preliminares realizados com o atual *hardware* do NCP_2 mostram que as operações de aplicação e geração de *diff* realizadas pelo processador i960 poderiam ser mais eficientemente executadas, caso fosse possível eliminar o overhead associado às operações das tarefas (G.II) e (A.II) descritas na seção anterior. A idéia é que o tempo de aplicação e geração de *diffs* fique limitado apenas pelo tempo de acesso ao barramento. Para isso, desenvolvemos e avaliamos uma arquitetura altamente especializada para a execução das tarefas com *diffs*: o Gerente de *Diff* Dinâmico (GDD). O GDD foi subdivido em três unidades funcionais, que operam em paralelo (figura 3):

- Unidade de Busca de Máscara
- Unidade DMA
- Unidade de Processamento de Máscara

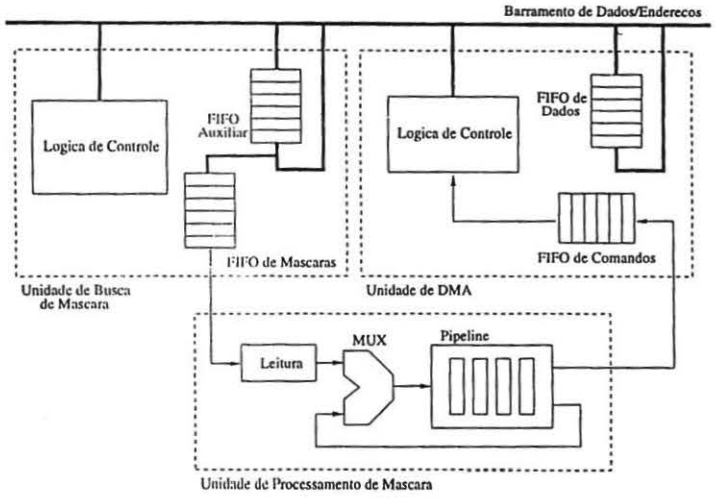

Figura 3: O Gerente de *Diff* Dinâmico.

Dessa forma, as três tarefas sequenciais descritas acima são aqui realizadas em paralelo. Podemos fazer a seguinte associação: as operações das tarefas (G.I) e (A.I) são executadas pela Unidade de Busca de Máscara; a Unidade de Processamento de Máscara executa as operações descritas em (G.II) e (A.II); e as tarefas (G.III) e (A.III) são executadas pela Unidade DMA.

O uso das FIFOs mostradas na figura 3 maximizam a independência funcional das unidades. A seguir, descrevemos a implementação das três unidades funcionais do GDD.

4.1 Unidade de Busca de Máscara

A Unidade de Busca de Máscara desempenha três tarefas principais:

- Inicialização da operação (aplicação ou geração de *diff*) através dos registradores do registro de configuração.
- Leitura das máscaras, no caso de aplicação de *diffs*.
- Operação atômica de leitura e escrita das máscaras, no caso de geração.

O registro de configuração inicializa a arquitetura do GDD para a execução de suas funções através do uso dos seguintes registradores de configuração:

R1 (28 bits) - contém o endereço de onde as máscaras serão lidas;

R2 (28 bits) - contém o endereço na memória do CP para onde as máscaras serão copiadas no caso de geração de *diff*;

R3 (23 bits) - guarda o endereço físico da página (da memória principal do PC) em questão

R4 (28 bits) - contém o endereço na memória do CP de onde os dados do *diff* serão lidos (no caso de aplicação) ou armazenados (no caso de geração).

Além disso, outros tipos de informações são passadas pelo registro de configuração, tais como: tipo de operação (aplicação ou geração), habilitação de interrupção ao término da operação e tamanho de página. As informações contidas no registro de configuração

podem ser lidas e escritas pelo processador RISC i960. Além disso, essas informações são passadas para as outras unidades funcionais do GDD, de acordo com a necessidade.

As máscaras correspondentes ao *diff* são lidas pela Unidade de Busca de Máscara em *burst* de quatro palavras e armazenadas em uma FIFO de quatro posições (FIFO Auxiliar). No caso de geração, a Unidade de Busca realiza uma operação atômica, onde, imediatamente após a leitura, as máscaras lidas são copiadas no endereço de destino fornecido pelo registro de configuração (R2). Da FIFO Auxiliar, as máscaras são transferidas para a FIFO de Máscaras (de oito posições), que funciona como interface entre a Unidade de Busca e a Unidade de Processamento de Máscara.

4.2 Unidade de DMA

A Unidade de DMA tem como principal tarefa a transferência de dados do *diff*. No caso de uma aplicação, a Unidade de DMA realiza um prefetch dos dados enquanto aguarda comandos de escrita na memória do processador principal. Esses comandos são disparados pela Unidade de Processamento de Máscara através da FIFO de Comandos. Os comandos determinam onde os dados serão armazenados (deslocamento em relação ao endereço base da página) e tipo de acesso (escrita single ou em *burst*). Os dados buscados durante o prefetch são armazenados na FIFO de Dados.

Durante uma operação de geração de *diff*, a Unidade de DMA aguarda comandos da Unidade de Processamento de Máscara para buscar os dados do *diff* na memória principal do PC. Estes dados são armazenados na FIFO de Dados e quando atingem o número de quatro são escritos em *burst* na memória principal a partir do endereço indicado no registro de configuração (R4). Assim como na operação de aplicação, os comandos passados para o DMA indicam o deslocamento em relação ao endereço da página e o tipo de acesso para a leitura dos dados.

A Unidade de DMA informa à Unidade de Busca de Máscara o término da operação, que por sua vez pode ou não disparar uma interrupção para o i960, dependendo do estado do sinal que habilita interrupção no registro de configuração.

4.3 Unidade de Processamento de Máscara

A Unidade de Processamento de Máscara tem como principal atividade a varredura das máscaras da Memória de Vetores e geração dos deslocamentos relativos ao endereço de página da memória principal. Um bit ativado dentro de uma máscara corresponde a uma modificação efetuada pelo processador principal, por isso, a posição relativa desse bit ativado deve ser identificada. Através dessa posição relativa identificada, o endereço do dado modificado é obtido.

Além da função descrita acima, a Unidade de Processamento de Máscara detecta grupos contíguos de bits setados permitindo que a Unidade de DMA manipule dados de *diff*s em acessos em *burst*, quando possível. Como o barramento do processador RISC i960 só aceita *burst*s de até 4 palavras, esse é o número máximo de blocos de bits ativados contiguamente procurados pela Unidade de Processamento de Máscara. Enfim, as seguintes tarefas são realizadas pela Unidade de Processamento de Máscaras:

1. Leitura de máscaras presentes na FIFO de Máscaras.
2. Cálculo do deslocamento relativo ao endereço da página.
3. Cálculo do tamanho do *burst* que pode ser realizado(de acordo com o número contíguo de bits setados na máscara).

4. Escrita dos resultados extraídos em (2) e (3) na FIFO de Comandos (que funciona como interface entre a Unidade de Processamento de Máscara e do DMA).

Como a figura 3 ressalta, uma arquitetura pipeline com fluxo de dados recursivo foi idealizada para a implementação das tarefas descritas acima. Nota-se que um multiplexador seleciona a entrada da máscara no pipe de n estágios. A entrada pode ser proveniente de uma máscara nova (lida da FIFO de Máscaras) ou de uma máscara realimentada (vinda do fim do pipe).

Cada vez que uma máscara entra no pipe, os estágios deste percorrem os bits da máscara (do menos significativo ao mais significativo) procurando um bloco contíguo de bits ativados na máscara. Se nenhum bit ativado é encontrado, nenhum comando é passado ao DMA. Quando algum bit ativado é encontrado, os estágios do pipe extraem o tamanho do bloco de bits ativados e o deslocamento inicial do bloco em relação ao endereço do início da página em questão. Essas duas informações são passadas então para a FIFO de Comandos.

Além disso, os estágios do pipe geram uma nova máscara a partir da máscara anterior, porém com os bits do bloco encontrado resetados. Dessa forma, se a nova máscara ainda contiver algum bit ativado, esta será recolocada no início do pipe.

A máscara realimentada tem sempre prioridade em relação a uma nova máscara. Desta maneira, várias máscaras diferentes podem estar sendo processadas por cada estágio do pipeline em um dado instante. Dependendo da configuração de bits das máscaras, isto pode provocar uma desordenação na geração dos dados do *diff*, ou seja, a ordem com que os dados são transferidos não corresponde a ordem com que os bits estão setados nas máscaras. No entanto, como o mesmo mecanismo de varredura é utilizado na hora de aplicar um *diff*, esta desordenação é desfeita na hora da aplicação. Justamente para garantir a reordenação, duas situações provocam o congelamento de todos os estágios do pipeline: FIFO de Máscaras vazia ou FIFO de Comandos cheia. Isto assegura que a reordenação não ficará sujeita à disponibilidade ou não do barramento de acesso à memória que é não-determinística tanto na geração quanto na aplicação de um *diff*.

4.4 Implementação do GDD

O Gerente de *Diff* Dinâmico foi completamente descrito em linguagem VHDL [ML93], linguagem de alto nível destinada a descrição de *hardware*. Os próximos passos que seguem a codificação em VHDL consistem da simulação lógica e síntese do projeto em questão. Atualmente, essas duas tarefas estão em andamento. Enquanto a simulação lógica realiza massivos testes, que buscam a validação lógica e funcional do código VHDL, tarefas de síntese já estão sendo executadas em paralelo. Os resultados preliminares das tarefas de síntese realizadas, nos mostram que FPGAs da família Flex10k, da Altera, conseguem implementar a arquitetura do GDD eficientemente, com frequência de operação maior que 33MHz. Devido ao baixo custo desses dispositivos de lógica programável, a inclusão do GDD no projeto não aumentará o custo total do CP.

Os testes de síntese foram fundamentais quando na escolha dos algoritmos que implementam a funcionalidade dos estágios do pipe da Unidade de Processamento de Máscaras. Diversos algoritmos foram escritos em VHDL e tiveram suas implementações testadas em FPGAs da família Flex10k. O objetivo destes testes estava na busca da implementação que apresentasse o menor número de estágios para o pipe (consequentemente a menor latência) para frequência de operação dos estágios maior que 33MHz. Para alcançar este objetivo, buscamos distribuir as funções dos estágios do pipe da melhor maneira possível, isto é, procuramos uma implementação balanceada no que diz respeito aos tempos de

execução das funções de cada estágio do pipe. A melhor implementação encontrada nos forneceu um número total de três estágios, cujas funções são descritas a seguir :

Estágio 1 - Recebe a máscara do Multiplexador. Divide a máscara em oito nibbles (4 bits), testa cada nibble em paralelo para ver se todos os bits estão em zero e caso contrário, passa para o próximo estágio o primeiro nibble que contiver pelo menos um bit setado.

Estágio 2 - Calcula o tamanho do *burst* (4,3,2 ou 1) a ser executado pelo DMA, de acordo com o tamanho do bloco de bits setados encontrados no nibble. Calcula o deslocamento do bloco de bits encontrado em relação ao endereço inicial da página. Este estágio ainda verifica se a máscara vai ser realimentada depois de passar pelo terceiro estágio. Esta condição é satisfeita se ainda restarem bits setados além do(s) bit(s) detectado(s) neste estágio.

Estágio 3 - Escreve na FIFO de Comandos o deslocamento e o tamanho do *burst*. Gera uma nova máscara filtrando os bits correspondentes e realimenta a máscara para o estágio 1, se for o caso.

4.5 Desempenho

Como mencionamos anteriormente, a arquitetura do GDD foi desenhada para que o tempo de execução de operações de aplicação e geração de *diff*s fosse limitado apenas pelo tempo de acesso ao barramento. Devido ao paralelismo das três unidades funcionais do GDD descritas acima, de fato, as operações da Unidade de Processamento de Máscara são sempre executadas em paralelo com operações das outras duas unidades do GDD. Como as funções da Unidade de Busca de Máscara estão direcionadas exclusivamente ao acesso de máscaras e como a Unidade de DMA trata exclusivamente dos acessos a dados, o objetivo a que o GDD se propõe é alcançado.

Abaixo desenvolvemos as equações de limite superior de tempo de execução das duas operações com *diff*s do GDD, sendo que a equação 1 representa o tempo de geração de *diff* e a equação 2 representa o tempo de aplicação de *diff*. Como mencionado anteriormente, todas as variáveis que participam dessas equações (descritas na tabela 1) estão relacionadas com os tempos de acesso externo, como leituras e escritas, a menos da variável tlm, que indica o tempo de latência máximo do pipeline do GDD. Na atual implementação do GDD esta latência corresponde a 5 *ticks* do clock de 33MHz, e corresponde a casos onde nenhuma das máscaras apresenta algum bit ativado ou a casos onde somente a última máscara lida apresenta algum bit ativado.

$$TGDiff = nm \times (tlm + tem) + mlp + \sum_{i=1}^{4}(ndi \times tldi) + (ndt \div 4) \times ted4 + (ndt \bmod 4) \times ted1 \quad (1)$$

$$TADiff = nm \times tlm + mlp(ndt \div 4) \times tld4 + (ndt \bmod 4) \times tld1 + \sum_{i=1}^{4}(ndi \times tedi) \quad (2)$$

5 Resultados

Nesta seção, comparamos os resultados de desempenho das operações com *diff*s do atual *hardware* do NCP_2 com os resultados extraídos do GDD. Devido à semelhança das operações de aplicação e geração, sem perda de generalidade, concentramos nossos estudos comparativos na operação de geração de *diff*.

Em geral, as aplicações de memória compartilhada não apresentam padrões claros quanto ao perfil de modificações realizadas sob uma página. Por isso, para testarmos a eficiência dos algoritmos de operações dinâmicas com *diff*s do NCP_2, geramos perfis aleatórios das máscaras que representam as modificações de uma página. Lembramos que

nm	Número total de máscaras
tlm	Tempo de leitura de uma máscara da Memória de Vetores do CP (memória SRAM)
tem	Tempo de escrita de uma máscara na memória DRAM do CP
mlp	Latência máxima da arquitetura pipeline do GDD
ndi	Número de *bursts* de i palavras de dados
ndt	Número total de dados modificados na página
tedi	Tempo de escrita de dados na memória DRAM do CP através de *bursts* de i palavras
tldi	Tempo de leitura de dados na memória principal do PowerPC através de *bursts* de i palavras

Tabela 1: Definição das variáveis analíticas utilizadas.

Parâmetro	Valor Experimental
tlm	90ns
tem	120ns
tld1	450ns (acesso *single*)
tld2	480ns (*burst* de 2 palavras)
tld3	540ns (*burst* de 3 palavras)
tld4	570ns (*burst* de 4 palavras)
ted1	120ns (acesso *single*)
ted4	210ns (*burst* de 4 palavras)

Tabela 2: Valores experimentais dos parâmetros.

o tamanho da página no sistema do NCP_2 é igual a 4kBytes, por isso, 1024 bits da Memória de Vetores são suficientes para a representação das modificações de uma página. Assim, mais de 200 mil entradas foram geradas a partir do seguinte algoritmo. Considerando um número x de bits ativados em um espaço de 1024 bits, geramos mil vetores de 1024 bits, onde as posições dos x bits ativados são escolhidas aleatoriamente. Começamos com $x = 0$ até $x = 1024$. Com isso, geramos nossos vetores de teste.

Os algoritmos de aplicação e geração de *diffs* que rodam atualmente no processador i960 foram escritos em linguagem C e compilados com nível de otimização máximo. Através do timer interno do i960, medimos o tempo de execução dessas operações para os vetores de teste descritos acima. O gráfico da figura 4 mostra os resultados para a operação de geração de *diff*. Mais especificamente, mostramos o tempo médio de execução obtido para cada ponto x (onde x é o número de bits ativados).

Para a avaliação do desempenho do GDD quanto à geração de *diffs*, utilizamos a equação 1 descrita na seção anterior. Através de um analisador lógico e de testes de leitura e escritas efetuados pelo processador i960, extraímos os parâmetros da equação 1 mostrados na tabela 2.

Com todos os parâmetros extraídos, medimos o tempo de execução da geração de *diff* para os vetores de teste descritos acima. O gráfico (figura 4) mostra o tempo médio de execução obtido para cada ponto x.

O tempo de execução obtido para cada ponto x (x é o número de bits ativados) varia em relação ao número de *bursts* encontrados no vetor. Por isso, mostramos nos gráficos

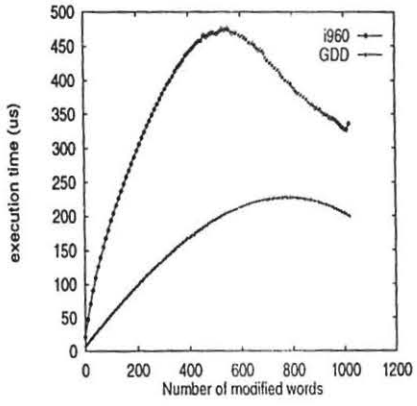

Figura 4: Tempo de Geração em Função do Número de Palavras Modificadas.

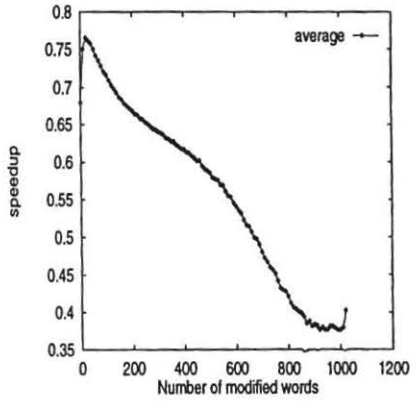

Figura 5: *Speedup* do GDD em Relação ao *Hardware* Atual.

o tempo médio de execução obtido para cada ponto x. A variação baseia-se no fato de que para um dado número de bits ativados, a forma com que estes bits aparecem no vetor de 1024 bits varia e consequentemente varia o número de *bursts* encontrados no vetor. Quanto maior o número de acessos em *burst*, menor o tempo total de execução.

As curvas do gráfico descrevem o compromisso entre o número de bits ativados e o número de acessos de dados feitos em *burst*. Obviamente, quanto maior o número de bits ativados maior o número de dados lidos da memória principal e por isso, maior o tempo de execução. Este comportamento é observado no começo das curvas. Porém, quanto maior o número de bits ativados, maior também é a probabilidade do aparecimento de blocos contíguos de bits ativados, o que acarreta em um maior número de acessos feitos em *burst*. Como acessos em *burst* custam bem menos do que acessos do tipo *single*, a partir de um certo número de bits ativados, verificamos que o tempo de execução começa a decair em função do número de bits ativados.

O gráfico (figura 5) mostra os ganhos obtidos pelo GDD em relação aos resultados de tempo de execução do atual *hardware* do NCP_2 para a operação de geração de *diff*.

Como era esperado, à medida que o número de bits ativados aumenta, o ganho relativo do GDD diminui. A explicação deste fato é simples. Quanto maior o número de bits ativados, maior a razão: tempo perdido em acessos ao barramento sobre tempo total de execução. Como o tempo de execução do GDD é limitado pelo tempo de acesso ao barramento, para vetores com muitos bits ativados o ganho relativo é menor.

O GDD obteve ganho máximo de 76% e mínimo de 37% em relação ao tempo médio das operações de geração de *diff* do atual *hardware* do NCP_2.

6 Conclusão

Atualmente, a primeira versão do NCP_2 encontra-se em testes. Os detalhes de implementação do *hardware* desta versão do NCP_2 foram descritos neste trabalho, assim como a maneira com que este oferece suporte aos protocolos de DSM, para minimizar os overheads provenientes destes protocolos de coerência implementados em software.

Testes preliminares realizados com o atual *hardware* apontaram a existência de espaços para aprimoramentos das operações com *diffs*, as quais atualmente são realizadas pelo processador RISC i960. Estudos sobre o desempenho do atual *hardware* do NCP_2 mostraram

que uma operação de aplicação ou geração de *diff* de página pode consumir um tempo de até 0.5 ms.

Para diminuir este overhead, avaliamos um *hardware* altamente especializado (o GDD), capaz de diminuir este overhead de operações com *diffs* em até 76%. A arquitetura do GDD explora ao máximo o paralelismo intrínseco das operações com *diffs*, de maneira a limitar o tempo dessas operações apenas pelo tempo de acesso ao barramento (escritas e leituras de dados externos ao GDD). Atualmente, o GDD está sendo implementado em FPGAs da família Flex10k da Altera. Resultados preliminares desta implementação mostram que a frequência de operação do GDD está acima de 33MHz.

Obviamente, a flexibilidade de programação que um processador RISC, como o i960, oferece não pode ser descartada. Por isso, concluímos que, na próxima versão do NCP_2, o processamento básico do *hardware* de apoio a protocolos DSM deve ser fornecido pela dupla: Processador RISC + GDD.

7 Agradecimentos

Gostaríamos de agradecer a Cristiana Seidel e Luiz Pessoa pela ajuda na finalização deste artigo e aos demais colaboradores do Projeto NCP_2 pelas sugestões na elaboração deste trabalho.

Referências

[ABS+96] C. L. Amorim, R. Bianchini, G. Silva, R. Pinto, M. Hor-Meyll, M. De Maria, L. Whately, and J. Barros Jr. A Segunda Geração de Computadores de Alto Desempenho da COPPE/UFRJ. *Anais do VIII SBAC-PAD*, 1996.

[ACD+96] C. Amza, A. Cox, S. Dwarkadas, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2), Feb 1996.

[ACDZ97] C. Amza, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt Between Single Writer and Multiple Writer. In *Proceedings of The 3rd International Symposium on High-Performance Computer Architecture*, February 1997.

[BCF+95] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet – A Gigabit-per-Second Local-Area Network. *IEEE Micro*, February 1995.

[BKP+96] R. Bianchini, L. I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *ASPLOS7*, pages 198–209, October 1996.

[BZS93] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the IEEE COMPCON'93 Conference*, Feb 1993.

[CBZ91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th Symposium on Operating Systems Principles*, Oct 1991.

[Int95] Intel. *i960 Hx Microprocessor User's Manual*, 1995.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[Kel96] P. Keleher. The Relative Importance of Concurrent Writers and Weak Consistency Models. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, May 1996.

[MB97] L. R. Monnerat and R. Bianchini. ADSM: A Hybrid DSM Protocol that Efficiently Adapts to Sharing Patterns. Tech. Report ES-425/97, COPPE Systems Engineering, Federal University of Rio de Janeiro, March 1997.

[ML93] S. Mazor and P. Langstraat. *A Guide do VHDL.* Kluwer Academic Publishers, 1993.

[Mos93] D. Mosberger. Memory Consistency Models. *Operating Systems Review*, pages 18–26, January 1993.

[Mot94] Motorola. *PowerPC 604 RISC Microprocessor User's Mamual*, 1994.

[SBA97] C. B. Seidel, R. Bianchini, and C.L. Amorim. The Affinity Entry Consistency Protocol. In *Proc. of the International Conference on Parallel Processing*, 1997.

[SHMM+96] G. Silva, M. Hor-Meyll, M. De Maria, R. Pinto, L. Whately, J. Barros Jr., R. Bianchini, and C. L. Amorim. O Hardware do Computador Paralelo NCP2 da COPPE/UFRJ. Relatorio Tecnico ES-394/96, COPPE Sistemas, Universidade Federal do Rio de Janeiro, Junho 1996.

[WPS+96] L. Whately, R. Pinto, G. Silva, M. Hor-Meyll, M. De Maria, J. Barros Jr., R. Bianchini, and C. L. Amorim. O Software do Computador Paralelo NCP2 da COPPE/UFRJ. Relatorio Tecnico ES-395/96, COPPE Sistemas, Universidade Federal do Rio de Janeiro, Junho 1996.