Implementação e Avaliação de um MPSoC Homogêneo Interconectado por NoC

Odair Moreira
Pontifícia Universidade Católica - PUCRS
FACIN
Porto Alegre – RS – Brasil
odair.moreira@pucrs.br

Fernando Gehm Moraes
Pontifícia Universidade Católica - PUCRS
FACIN
Porto Alegre – RS – Brasil
fernando.moraes@pucrs.br

Abstract – The increasing number of transistors on a single chip has brought new design challenges, among them, how to design new circuits using this available number of transistors. The reuse of Intellectual Property (IP) cores enables the design of such complex systems. The design of a system through the reuse of processors results in a MPSoCs (Multiprocessor System on Chip). MPSoCs are becoming increasingly popular in embedded systems due to their high performance and flexibility. In recent years universities and companies have been developing large projects in multiprocessing. The goal of the work is to develop and to evaluate a homogenous NoC-based MPSoC in order to investigate the benefits of such architecture for designing embedded applications.

Keywords: MPSoC, NoC, multicore.

I. INTRODUÇÃO

Os MPSoCs são arquiteturas que buscam um compromisso entre restrições da tecnologia VLSI e as necessidades da aplicação. Os MPSoCs representam uma evolução do conceito de SoC, e com isso se beneficiam das mesmas vantagens inerentes ao projeto de SoCs. Por exemplo, a reusabilidade de núcleos de hardware (ou simplesmente IPs - módulos pré-projetados e pré-validados desenvolvidos por diferentes empresas - [1]). Adicionalmente, MPSoCs são tendência porque amenizam a crise de projeto VLSI, através da redução do espaço entre a disponibilidade de transistores por circuito integrado nas tecnologias atuais e a capacidade de projeto de SoCs [2].

MPSoCs que empregam redes intra-chip (NoCs, *networks on chip*) como infra-estrutura de interconexão são uma realidade, pois produtos comercias como os MPSoCs desenvolvidos pela Intel [7] e Tilera [9], corroboram tal afirmativa. Vantagens como maior escalabilidade e paralelismo de comunicação quando comparadas a barramentos, justificam a adoção de NoCs como infra-estrutura de comunicação. Uma NoC possui as mesmas características de uma rede de computadores e pode ser definida como um conjunto de roteadores e canais ponto-a-ponto, que interconectam os núcleos de um sistema integrado de modo a suportar a comunicação entre esses núcleos. As principais características das NoCs que motivam o seu estudo são: (*i*) confiabilidade; (*ii*) eficiência no gerenciamento de energia; (*iii*) escalabilidade da largura de banda em relação a arquiteturas de barramento (*iv*) reusabilidade.

O presente trabalho tem por objetivo o desenvolvimento de um MPSoC homogêneo baseado em NoC, afim de avaliar os pros e contras da utilização desse tipo de arquitetura.

O presente documento está organizado como segue. A Seção 2 apresenta arquiteturas de MPSoC. A Seção 3 descreve a arquitetura de hardware e a infra-estrutura utilizada para o desenvolvimento do MPSoC. A Seção 4 apresenta a infra-estrutura de software. A Seção 5 apresenta os resultados obtidos, e a Seção 6 apresenta as conclusões e as direções para trabalhos futuros.

II. ORGANIZAÇÃO DO MPSOCS

A Tabela 1 apresenta características de MPSoCs desenvolvidos pela indústria.

Tabela 1- Características de MPSoCs industriais.

Nome	Fabricante	*Número de PEs Multiprocessamento*	*Infra-estrutura de Comunicação*	Desempenho
PC102 (2005)	***picoChip***	**322 Homogêneo**	**Híbrid** (picoBus + switches)	**-**
CELL (2006)	***IBM***	**9 Heterogêneo**	**Híbrid** (Ring + star network)	**204 GFlops/s**
Am2045 (2006)	***Ambric***	**360 Homogêneo**	**NoC mesh**	**1 TOPS**
Polaris (2007)	***Intel***	**80 Homogêneo**	**NoC mesh**	**1.28 TFlops/s**
Tile64 (2007)	***Tilera***	**64 Homogêneo**	**NoC mesh**	**443 BOPS**

O PC102 é um MPSoC proposto para aplicação em redes sem fio. Baseado na arquitetura picoArray, os processadores são interconectados através de barramentos picoBus (32bits) e chaves programáveis [4]. O PC102 possui 322 processadores, organizados na forma de uma matriz. Cada processador executa um único processo, e possui suas próprias memórias de dados e instruções.

A arquitetura *CellBE* (*Cell Broadband Engine Architecture*) é constituída por nove núcleos, sendo um PowerPC e oito núcleos SPE (*Synergistic Processor Element*) [5]. O PPE (*PowerPC Processing Element*) é a unidade central de processamento, enquanto os SPEs são responsáveis por acelerar aplicações específicas. Além dos nove núcleos, a arquitetura possui um barramento principal EIB (*Element*

Interconect Bus), um controlador de memória DEMAC (*Direct Memory Acces Controller*), um controlador de memória XDRAM, e uma entrada e saída de Rambus FlexIO.

O MPSoC Am2045 é composto por processadores RISC de 32bits [6]. O Am2045 consiste em uma matriz de *brics* (elementos de processamento) 5x9. Cada um deles possui oito núcleos processadores e 8KB de memória local. Ao todo são 360 processadores e 585KB de memória. Além disso, estão disponíveis duas interfaces SDRAM DR2, controlador PCI, 128 portas de E/S, interface serial para memória *flash*, e interface JTAG para depuração.

O Intel Core-80, ou Polaris, é um MPSoC com 80 núcleos [7][8]. Todos os núcleos compartilham uma região de *cache* de 4 MB. Cada núcleo implementa duas unidades de processamento de ponto flutuante e um circuito de roteamento com 5 portas de comunicação formando uma rede de interconexão *mesh*, com alta vazão e baixa latência para a troca de mensagens.

O TILE64 possui uma rede de 64 núcleos (*tiles*) com características idênticas, conectado por uma rede malha responsável pela comunicação entre os núcleos e com o mundo exterior [9]. Cada *tile* é um processador, incluindo uma memória *cache* integrada com L1 e L2 que interliga o *tile* à malha. A rede malha é constituída por cinco sub-redes independentes. Ao invés de um barramento centralizado, cada núcleo tem sua própria memória, podendo compartilhar dados com quatro núcleos vizinhos. A memória está dividida em duas *caches* L1 de 8KB e uma L2 de 64KB, com L3 simulado até 5MB quando necessário.

Como exemplo de MPSoC acadêmico pode-se destacar o MPSoC HeMPS [10]. Para a infra-estrutura de hardware foi utilizada a NoC HERMES [11], e o processador de domínio público Plasma [12]. Módulos de hardware foram desenvolvidos, visando conectar o processador à NoC e realizar a alocação de tarefas na memória do processador. Para a infra-estrutura de software, foi desenvolvido um *microkernel* multitarefa que é executado em cada processador escravo.

III. Arquitetura do MPSoC Proposto

A arquitetura proposta é um MPSoC homogêneo, constituído por núcleos de processamento (NP), compostos por processadores MR4 [13] e conectados através da NoC HERMES [11], com uma topologia do tipo malha 2D. A arquitetura foi implementada utilizando a linguagem de descrição de hardware VHDL.

A rede HERMES foi parametrizada utilizando as seguintes características: (*i*) controle de fluxo *credit based* (sem canais virtuais); (*ii*) dimensão 3x3; (*iii*) largura do *flit* igual a 32 bits; (*iv*) profundidade dos *buffers* igual a 8 *flits*; (*v*) algoritmo de roteamento XY.

Os núcleos de processamento (NP) contêm um processador MR4, com memórias de dados e instruções, e uma interface de rede. A arquitetura do NP é apresentada na Figura 1.

O processador MR4 tem como características gerais: (*i*) arquitetura Harvard, *load/store*; (*ii*) dados e endereços de 32 bits; (*iii*) endereço de memória orientado a byte, onde cada palavra ocupa 4 posições consecutivas de memória; (*iv*) banco de registradores com 32 registradores de uso geral; (*v*) formato de instrução regular. Todas as instruções possuem o mesmo tamanho, ocupando uma 1 palavra em memória, ou seja, 4 posições consecutivas de memória. Para adequar o processador MR4 a este trabalho foram feitas as seguintes alterações em sua arquitetura: (*i*) a inclusão de tratamento da interrupção; (*ii*) inclusão das instruções MFCO e MTCO; (*iii*) inclusão do registrador EPC.

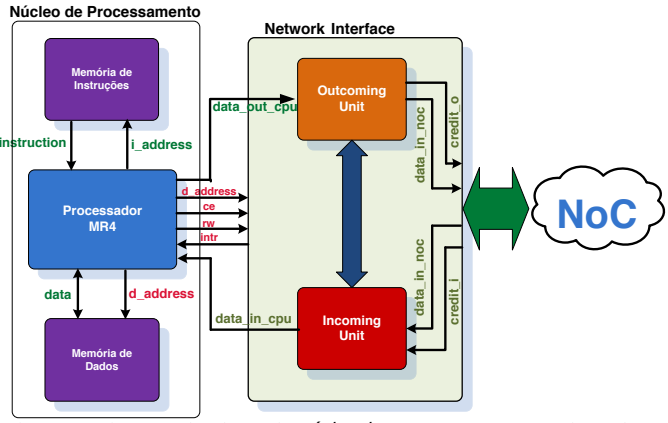


Figura 1 - Diagrama de Blocos do *núcleo de processamento* (NP), formado pelo processador MR4, memória de instruções e memória de dados e a *network interface* (NI) com seus módulos de comunicação (*Outcoming Unit* e *Incoming Unit*).

A interface de rede (NI, *Network Interface*) contém dois módulos principais: *outcoming unit* e *incoming unit*. A *outcoming unit* tem por objetivo enviar pacotes oriundos do processador para a NoC. A *outcoming unit* possui um *buffer* de até 1024 posições. O processo de envio é feito em 3 etapas:

1. O processador escreve os dados do pacote a ser enviado no *buffer*, quando a faixa de endereçamento for fornecida entre NI_TX_BASE_ADDRESS e NI_TX_HIGH_ADDRESS.
2. O processador avisa a *outcoming unit* que terminou de escrever o pacote no *buffer*, ativando o sinal denominado *end_pack*. Este sinal é ativado através de escrita no endereço NI_TX_END_PACKET.
3. Uma vez sinalizado o final de escrita do pacote, o mesmo é injetado na NoC.

A *incoming unit* tem por objetivo receber pacotes da NoC, e enviá-los para o processador. Esta interface é similar à anterior, possuindo também um *buffer* (*buf_ni_noc*) de até 1024 posições. O pacote oriundo da NoC é escrito neste *buffer*. Ao final da escrita do pacote ativa-se o sinal de interrupção do processador, fazendo com que o mesmo leia este pacote e armazene-o na área de dados.

IV. Infra-Estrutura de Software

A infra-estrutura de software desenvolvida compreende os *drivers* de recepção e transmissão de pacotes. A recepção de pacotes é feita através de interrupção, a partir da rotina denominada *interrupt handler*.

Quando ocorre uma interrupção, o registrador EPC

armazena o endereço da próxima instrução e PC recebe o endereço da rotina de tratamento de interrupção, denominada *interrupt_handler*. O endereço armazenado em EPC é novamente carregado quando a rotina *interrupt_handler* é finalizada. Os endereços de entrada e saída mapeados em memória utilizados pela *interrupt_handler* compreendem: (*i*) NI_RX_BASE_ ADDRESS: registrador que contém o endereço da interface de rede; (*ii*) PACKET_BASE_ADDRESS: registrador que retorna o endereço em memória de pacote válido (pacote que já foi recebido e transferido para a memória de dados); (*iii*) NI_RX_END_RECEPTION: registrador que informa o fim de recepção do pacote.

Para a implementação da rotina *interrupt_handler* foram criadas quatro regiões em memória para armazenar os pacotes recebidos, as quais são endereçadas por PACKET_BASE_ADDRESS. A rotina *interrupt_handler* é executada quando o sinal de interrupção externo é ativado e a interrupção está habilitada. Depois de salvar o contexto de todos os registradores, o laço apresentado na Figura 2 busca o endereço de uma das quatro regiões criadas para armazenar o novo pacote. Havendo área de memória disponível, inicia-se a leitura dos dados da NI (observar o salto na linha 5), a partir do endereço armazenado no registrador *$t1*. Se não houver espaço para armazenar um novo pacote, a rotina desabilita a interrupção através da instrução *mtc0* e depois restaura o contexto dos registradores. A interrupção é posteriormente habilitada quando algum pacote armazenado na área de dados for consumido.

No momento em que é encontrado um local para armazenar o pacote, executa-se o código a partir do rótulo *read_packet* (Figura 3), ou seja, começa-se a receber os dados da NI. Depois de recebido todos os dados da NI é enviado um sinal informando o fim da recepção dos dados através de NI_RX_END_RECEPTION e restaurado o contexto dos registradores. Em seguida depois da finalização da rotina de interrupção salta-se para o endereço armazenado no registrador EPC através da chamada à instrução *mfc0*.

O *driver* de envio possui três registradores mapeados em memória: (*i*) NI_TX_BASE_ADDRESS: registrador que contém o endereço da *outcoming unit*; (*ii*) NI_TX_BUSY: registrador que indica se a NI está ocupada; (*iii*) NI_TX_END_PACKET: registrador que informa o fim de transmissão do pacote. O código simplificado do *driver* de envio de um pacote de *n flits* é apresentado na Figura 4. O código apresentado na Figura 4 compreende salvar o contexto (linhas 1-4), dois laços principais e recuperação do contexto (linhas 19-22). O primeiro laço (linhas 8-9) verifica se a NI está livre, através da leitura do sinal *busy*, mapeado no endereço NI_TX_BUSY. O segundo laço (linhas 10-16) envia o pacote através da leitura da área de dados (linha 12), e escrita no *buffer*, no endereço NI_TX_BASE_ADDRESS (linha 13). Ao final do envio do pacote avisa-se à NI que há pacote disponível, escrevendo-se no endereço NI_TX_END_PACKET (linhas 17-18).

```
 1 find_packet:  la $t1,PACKET_BASE_ADDRESS
 2               addu $t1, $t1, $t0              # endereço do ponteiro do pacote
 3               lw $t1, 0($t1)                  # carrega o conteúdo do ponteiro, ou seja, o endereço do pacote
 4               lw $t2, 0($t1)                  # carrega a primeira palavra do pacote
 5               beq $t2,$zero, read_packet
 6               addiu $t0, $t0, 4               # contador "circular": 0,4,8,C, pois só são 4 pacotes
 7
 8               li $t5, 0x10                    ## se o contador chegou a 0x10 é sinal que os 4 espaços para
 9               bne $t0, $t5, find_packet       ## pacotes estão ocupados
10               mtc0 $zero, $12                 ## LOGO DESABILITA AS INTERRUPÇÕES E RECUPERA O CONTEXTO
11               j loop5
```

Figura 2 – Código para encontrar área de dados livre para armazenar pacote recebido.

```
read_packet:    la $t2,NI_RX_BASE_ADDRESS       # endereço do periférico (INTERFACE DE REDE)
                addiu $t2,$t2,4                 # posiciona no segundo endereço - o primeiro flit é o endereço
                lw $t3, 0($t2)                  # TAMANHO do payload + 2 flits do cont_tick

                sw $t3, 0($t1)                  # marca  pacote como ocupado
                xor $t0, $t0, $t0               # i=0

loop2:          addiu $t0,$t0,1                 # "i++"
                addiu $t2,$t2,4                 # posiciona o $s0 para o payload (3o, 4o, 5o flits)
                lw $t4, 0($t2)                  # carrega o valor apontado por $t2  (LE DA INTERFACE DE REDE)
                sw $t4, 4($t1)                  # armazeno O valor do flit recebido (GRAVA NA MEM. DE DADOS)
                addiu $t1,$t1,4                 # avança na memória de dados
                bne $t0,$t3,loop2               # se não recebeu todo o pacote, volta a ler os flits

                la $t0, NI_RX_END_RECEPTION     # endereço do registrador que indica fim do recebimento do pacote
                li $t1, 1
                sw $t1, 0($t0)                  # avisa ao hw final de recebimento de pacote

loop5:          lw $ra,0($sp)  ; lw $t0,4($sp)  ;  lw $t1,8($sp) ;  lw $t2,12($sp)
                lw $t3,16($sp) ; lw $t4,20($sp) ; lw $t5,24($sp) ;  lw $1, 28($sp)
                addiu $sp,$sp,32               ## recuperação do contexto

                mfc0 $a0,$14                   ## recuperação do  EPC ($14),e retorno para a aplicação
                jr $a0
```

Figura 3 – Código para armazenar os *flits* recebidos da NI.

```
# SALVA  CONTEXTO
1           SEND:       addiu $sp,$sp,-12
2                       sw $ra,0($sp)
3                       sw $s2,4($sp)
4                       sw $t1,8($sp)

# CARREGA ENDEREÇO DOS REGISTRADORES EM MEMORIA
5                       la $s1, NI_TX_BASE_ADDRESS
6                       la $t3, NI_TX_BUSY
7                       la $t4, NI_TX_END_PACKET

# LAÇO 1 VERIFICA SE A NI ESTÁ LIVRE (NI_TX_BUSY)
8           LOOP:       lw $t0, 0($t3)
9                       bne $t0, $zero, LOOP

# LAÇO 2 -  TRANSMISSÃO DO PACOTE ORIGINAL
10          LOOP2:      addu $s3, $s1, $
11                      addu $s4, $s2, $t0
12                      lw $t1, 0($s4)
13                      sw $t1, 0($s3)
14                      addiu $t0,$t0, 4
15                      addiu $s0,$s0,-1
16                       bne $s0, $zero, LOOP2

#  FIM DE TRANSMISSÃO DO PACOTE
17                      li $s0, 1
18                      sw $s0, 0($t4)

# RESTAURA CONTEXTO
19                      lw $ra,0($sp)
20                      lw $s2,4($sp)
21                      lw $t1,8($sp)
22                      addiu $sp,$sp,12

23                      jr $ra
```

Figura 4 – Código simplificado do *driver* de envio de pacote.

V. RESULTADOS

Para a validação do MPSoC proposto na Seção III, foi utilizada uma aplicação paralela do tipo *merge sort* executada em um PE "mestre" e uma aplicação do tipo *bubble sort* executada em PEs "escravos". As aplicações utilizam vetores de 16 a 600 elementos. Cinco cenários de simulação são apresentados:

1. Cenário 1: execução monoprocessada do algoritmo de *bubble sort*, para referência de desempenho. O tempo de execução para um vetor de 600 elementos foi de 9.184.058 *ciclos de clock*
2. Cenário 2: execução monoprocessada do *merge sort*, dividindo-se o vetor em 2 partes iguais.
3. Cenário 3: execução do *merge sort* com *bubble sort* em dois processadores escravos, cada um operando sobre uma metade do vetor. Este cenário é comparado ao *cenário 2*, para avaliação de aceleração com 2 processadores.
4. Cenário 4: execução monoprocessada do *merge sort*, dividindo-se o vetor em 4 partes iguais.
5. Cenário 5: execução do *merge sort* com *bubble sort* em quatro processadores escravos, cada um operando sobre ¼ do vetor. Este cenário é comparado ao Cenário 4, para avaliação de aceleração com 4 processadores.

No Cenário 5 o PE1 envia quatro pacotes para os processadores escravos executarem o *bubble sort* sobre um quarto do vetor, conforme a distribuição espacial apresentada na Figura 5. São utilizados 5 processadores, o PE1 é o mestre, responsável pela divisão do vetor, e a execução do *merge sort*, e os quatro outros responsáveis pelo *bubble sort*.

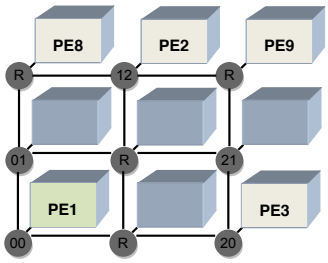


Figura 5 - Posição dos processadores para a simulação do Cenário 5.

A Figura 6 apresenta a simulação deste cenário. Os principais eventos desta simulação compreendem:

1. Injeção dos pacotes na rede. Depois que cada pacote é enviado, envia-se um sinal de fim de transmissão do pacote, habilitando o sinal de *end_pack* da NI;
2. Ativação da interrupção (*intr*), indicando que o processador está recebendo um pacote;
3. Recepção dos pacotes pelos processadores;
4. Duração da tarefa de ordenação aplicada aos pacotes recebidos;
5. O processador verifica se a NI está ocupada e envia os pacotes ordenados via *data_out_ext*. Pacotes 1 e 2;
6. O processador verifica se a NI está ocupada e envia os pacotes ordenados via *data_out_ext*. Pacotes 3 e 4;
7. A NI sinaliza o processador PE1 mediante o sinal de interrupção *intr*. O processador PE1 habilita o sinal *em_atendimento* para indicar que está atendendo a solicitação da NI;
8. Em *data_in_ext* ocorre a recepção dos pacotes; (a) primeiro *merge* executado sobre os pacotes 1 e 2;.(b) Segundo *merge* executado sobre os pacotes 3 e 4, (c) Ordenação final dos pacotes obtido via do *merge* final.

Esquematicamente, a Figura 7 apresenta os tempos de cada etapa da simulação apresentada na Figura 6. Inicialmente o PE1 envia ¼ do vetor para cada PE escravo, os quais consomem 574.818 ciclos para ordenar cada vetor. Uma vez os vetores parciais ordenados, este são enviados de volta ao PE1. Logo após a recepção de todos os pacotes, há uma primeira remontagem, correspondente à união de dois pacotes em um único pacote, com consumo de 9.009 ciclos. Sobre o vetor remontado é executado o primeiro *merge* com um tempo total de 25.546 ciclos. Na próxima etapa ocorre um tempo maior, 21.065 ciclos. Isso é devido a duas rotinas que são executadas, uma é a segunda remontagem sobre os pacotes 3 e 4 e a outra rotina é a montagem da parte alta do vetor para ser realizado o *merge* final. Logo em seguida é executado o segundo merge sobre o segundo pacote que foi remontado. Como não há mais pacotes a serem remontados, a próxima etapa a ser executada é a montagem da parte baixa do vetor. Esta etapa consome 11.259 ciclos. Na última etapa o *merge* executa a ordenação final dos pacotes com tempo final de 50.896 ciclos.

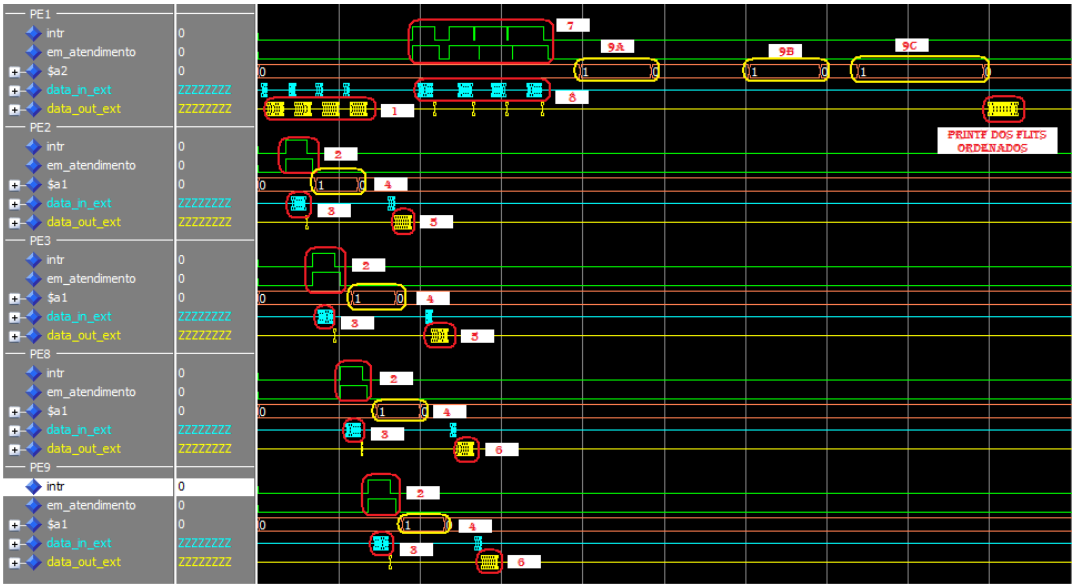


Figura 6 – Gráfico apresentando a aplicação *merge sort* executada em 5 processadores. PE1 envia ¼ do pacote de 600 *flits* para cada processador escravo (PE2, PE3, PE4, PE5). Cada processador escravo executa uma aplicação *bublle sort* e envia o pacote ordenado para PE1. PE1 remonta os pacotes recebidos e executa o *merge* final.

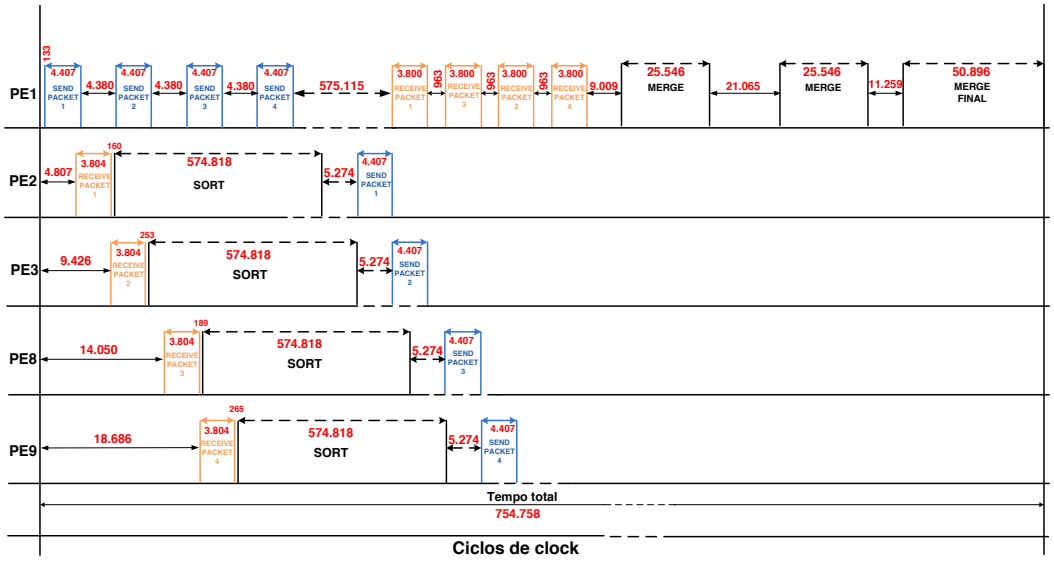


Figura 7 – Gráfico de tempos do Cenário 5. Execução dos aplicativos *merge sort* e *bubble sort* em um sistema com 5 elementos de processamento em vetor com 600 elementos.

Somando os tempos necessários para o envio e recepção dos quatro pacotes obtem-se um tempo total de 44.930 ciclos. Analisando o tempo total de execução para efetuar a ordenação de todos os pacotes de aproximadamente 754.758 ciclos, pode-se verificar que houve quase 6% de tempo gasto de comunicação na rede.

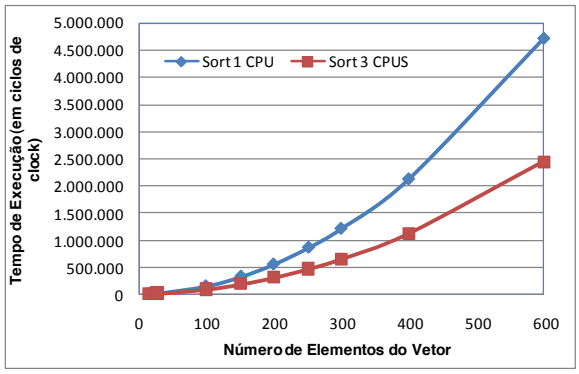
A Figura 8(a) apresenta o tempo de execução comparando-se o Cenário 2 ao Cenário 3, ou seja, aceleração por execução em 2 processadores escravos. A Figura 8(b) apresenta a comparação do Cenário 4 ao Cenário 5, ou seja, aceleração por execução em 4 processadores escravos. Notar neste tempo de execução a comparação com o Cenário 1, o qual consumiu 9,14 milhões de ciclos para ordenar o mesmo vetor de 600 elementos.

Conforme [14], a métrica de *speedup* refere-se ao ganho de desempenho obtido pela razão entre o tempo de execução do melhor algoritmo sequencial e sua forma paralela, ou seja, $S = T_1 / T_n$, onde: S é o *speedup*, T_1 é o tempo do algoritmo sequencial e T_n é o tempo de execução com n processadores.

Os valores de *speedup* na utilização de 2 CPUs e 4 CPUs são apresentados na Figura 9. Nos valores obtidos para 2 processadores o ganho de desempenho foi praticamente 2, ou seja, o impacto da comunicação da rotina *merge* foi muito

baixo. Já para 5 processadores (1 mestre e 4 escravos), o *speedup* ficou abaixo de 4, devido ao tempo de execução das rotinas *merge* e da comunicação.

(a)



(b)

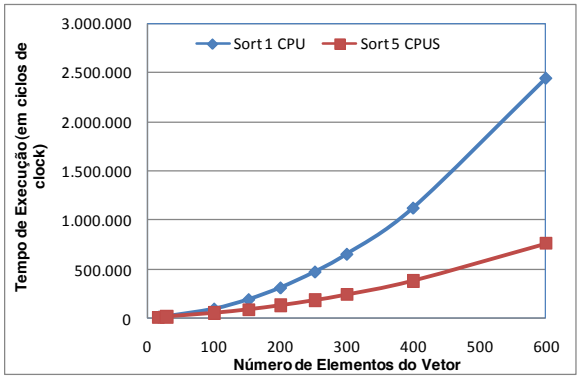


Figura 8 – Tempos execução no MPSoC desenvolvido, comparando o desempenho quando se utiliza em (a) 3 CPUs (1 mestre-2 escravas), e em (b) 5 CPUs (1 mestre-4 escravas).

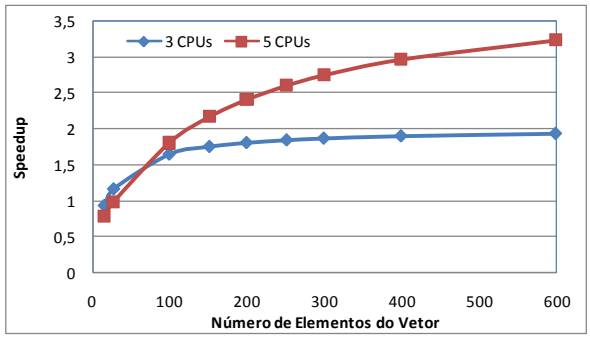


Figura 9 – Gráfico de *speedup* – *merge* para 3 (2 CPUs operando em paralelo) e 5 CPUs (4 CPUs operando em paralelo).

VI. CONCLUSÕES E TRABALHOS FUTUROS

Os atuais projetos de hardware devem aliar as restrições clássicas de projeto, como alto desempenho e baixo consumo de energia, aliado a um reduzido tempo para o produto chegar ao mercado. Uma forma de se conseguir atender a estas restrições conflitantes é através do reuso de módulos de propriedade intelectual, e em particular através do reuso de processadores, conduzindo assim aos MPSoCs.

As contribuições deste trabalho compreendem:

1. Desenvolvimento de um MPSoC homogêneo, com reuso dos módulos de processamento e NoC;
2. Desenvolvimento da interface de rede, tendo por característica o uso de *buffers* para injeção/recepção de dados. Esta característica permite uma simples integração de IPs à rede;
3. Desenvolvimento das camadas de software básico (*drivers*) para a realização da comunicação;
4. Modificação do processador MR4, com a inclusão do mecanismo de interrupção;
5. Avaliação do desempenho do MPSoC proposto utilizando uma aplicação paralela.

Como principais sugestões de trabalhos futuros pode-se citar: (*i*) integração de um módulo DMA e execução de simulações com um número maior de processadores para verificar o ganho de desempenho do sistema; (*ii*) inserir outro modelo de processador para analisar qual o impacto na mudança de arquitetura no sistema; (*iii*) pesquisar novos métodos para integrar em um mesmo ambiente a integração da NoC HERMES com bibliotecas de IPs, que venham a possibilitar a redução de tempo de geração do MPSoC.

REFERÊNCIAS

[1] Gupta, R. K., Zorian, Y. "Introducing Core-Based System Design". IEEE Design & Test of Computers, Vol.14 (4), Out./Dec. 1997, pp. 15-25.

[2] Henkel, J. "Closing the SoC design gap". IEEE Computer, Vol. 36(9), Set. 2003, pp. 119-121.

[3] Benini, L.; De Micheli, G. "Networks on Chips: a New SoC Paradigm". IEEE Computer, Vol. 35 (1), Jan. 2002, pp.70-78.

[4] Duller, A.; et al. "picoArray Technology: The Tool's Story". In: Design, Automation and Test in Europe (DATE), 2005, pp.106-111.

[5] Kahle, J.A., M. N. Day, H. P. Hofstee, et al. "Introduction to the Cell Multiprocessor". IBM J. Res & Dev., Vol.49 (4/5), Jul./Sep. 2005, pp. 589-604.

[6] Halfhill; T. R. "Ambric's New Parallel Processor - Globally Asynchronous Architecture Eases Parallel Programming". Microprocessors Report. Out. 2006, Disponível em http://www.ambric.com/pdf/MPR_Ambric_Article_10-06_204101.pdf .

[7] Vangal, S., et al. "An 80-Tile 1.28 TFLOPS Network on-chip in 65nm CMOS". In: IEEE International Solid-State Circuits Conference, 2007, pp. 5-7.

[8] Vangal, S., et al. "An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS". IEEE Journal of Solid-State Circuits, Vol.43 (1), Jan. 2008, pp.29-41.

[9] Tilera Corporation. "TILE64™ Processor". Product Brief Description. Santa Clara, CA, EUA. Agosto, 2007, 2p.

[10] Woszezenki, C. R. "Alocação de Tarefas e Comunicação entre Tarefas em MPSoCs". Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2007, 121p.

[11] Moraes, F. G., Calazans, N., Mello, A., Möller, L., Ost, L. "HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip". Integration the VLSI Journal, Amsterdam, Vol. 38(1), Out. 2004, pp. 69-93.

[12] PLASMA *Processor*. Capturado em www.opencores.org/projects.cgi/web/mip, Julho 2008.

[13] Moraes, F. G., Calazans, N. "Processador Multiciclo – MR4". Capturado em http://www.inf.pucrs.br/~moraes/ , Janeiro, 2008.

[14] Kumar, V., Grama A., Karypis, G., Gupta, A. "Introduction to Parallel Computing". Addison Wesley, 2º ed., 2003, 656p.