Sistema operacional embarcado eCos com suporte a SMP para o processador Nios II

Maikon Adiles Fernandez Bueno[1], **Christiane Regina Soares Brasil**[1], **Eduardo Marques**[1]

[1]Instituto de Ciências Matemáticas e de Computação - Universidade de São Paulo (USP)
Caixa Postal 668 – 13560-970 – São Carlos – SP – Brazil

Abstract. *The objective of this work consists on the exploration of the resources offered by FPGAs for the development of a multiprocessed platform with the purpose of parallel execution of tasks for robotic purpose. In this way, the eCos operating system was modified, with the addition of new characteristics to support of the Symmetric Multiprocessing model, using three soft-Core Altera Nios II processors. This platform was analyzed and validated through the execution of parallel algorithms, emphasizing aspects of performance and flexibility compared to other architectures.*

Resumo. *O propósito deste trabalho consiste no aproveitamento dos recursos oferecidos pela FPGA para o desenvolvimento de uma plataforma multiprocessada com a finalidade de execução paralela de tarefas para a robótica. Deste modo, o sistema operacional eCos foi modificado, com a agregação de novas funcionalidades, para permitir o suporte do modelo Multiprocessamento Simétrico, utilizando três processadores soft-core Nios II da Altera. Esta plataforma foi analisada e validada por meio da execução de algoritmos paralelos, enfatizando aspectos de desempenho e flexibilidade em relação a outras arquiteturas.*

1. Introdução

A computação reconfigurável está marcando o desenvolvimento de *hardware* nos últimos anos, e tornou-se um novo paradigma para a execução de tarefas em aplicações diferenciadas. A tecnologia FPGA (*Field Programmable Gate Array*) tem evoluído significativamente, alcançando elevados níveis de densidade, maior desempenho, e menor custo. Atualmente, as FPGAs podem possuir um conjunto com mais de 300.000 elementos lógicos e uma freqüência de 500MHz [Altera 2004] [Altera 2006]. Esse avanço torna a FPGA cada vez mais equiparável à tecnologia ASIC (*Application Specific Integrated Circuit*), a qual por muitos anos tem liderado a fabricação de dispositivos.

Com o avanço tecnológico, as FPGAs apresentam níveis de desempenho cada vez maiores, e muitos recursos são disponibilizados pelos fabricantes. O aumento de sua capacidade lógica motiva a utilização de vários tipos de soluções para explorar as características físicas oferecidas. Uma dessas soluções baseia-se na utilização de processadores específicos ou mesmo de propósito geral implementados em FPGAs. Esses processadores são conhecidos como *soft processors*, e eventualmente são elaborados por meio de diagramas esquemáticos ou linguagens de descrição de *hardware*.

Uma placa de FPGA pode suportar diversos *soft processors* simultaneamente. Essa arquitetura é conhecida como multiprocessador e permite a ampliação no número de processos em execução paralela, aumentando o desempenho do sistema.

O sistema operacional possui a tarefa de controlar a execução dos processos nessa plataforma, fornecendo métodos de sincronização para a exclusão mútua e ordenação de eventos entre processos. Com isso, diversos algoritmos podem ser paralelizados, de modo a utilizar a capacidade oferecida pela FPGA.

Deste modo, a expansão de ambientes propícios para execução de algoritmos, especificamente algoritmos utilizados em robótica móvel, podem oferecer grandes benefícios na relação de flexibilidade alcançada com *software* e desempenho em tarefas implementadas em *hardware*.

Com esse intuito, o principal objetivo deste trabalho consiste no projeto de um sistema operacional capaz de suportar uma arquitetura multiprocessada em FPGA, sendo capaz de gerenciar seus recursos de modo transparente. Para tanto, foi utilizando o sistema operacional eCos (*embedded Configurable operating system*), no qual algumas funcionalidades inerentes ao modelo SMP foram implementadas. A arquitetura base para execução de processos utilizando o eCos foi composta por três processadores Nios II, interligados por meio do barramento Avalon, com compartilhamento de memória, entre outros periféricos.

Este trabalho, desenvolvido no Laboratório de Computação Reconfigurável ICMC/USP, contribui para aferir a possibilidade de utilização de algoritmos da robótica na arquitetura descrita, assim como, também para atingir maior desempenho em tarefas executadas em FPGA, aumentando a aplicabilidade da computação reconfigurável.

2. Processadores em FPGA - Nios II

Processadores de propósito geral, na forma de ASICs, têm sido embutidos em placas de FPGA projetadas atualmente. Esses tipos de processadores são chamados de *hard processors*. Os processadores implementados em FPGAs utilizando computação reconfigurável são conhecidos como *soft processors*.

As principais fabricantes de FPGA Altera [Altera 2005] e Xilinx [Xilinx 2005], disponibilizam três linhas de processadores implementados em FPGA, PicoBlaze, MicroBlaze e Nios II, utilizado neste projeto.

Nios II consiste em um processador de 32-bits RISC de propósito geral, desenvolvido para atender uma grande escala de dispositivos embarcados. As principais características do Nios II são: conjunto de instruções, espaço de endereçamento e *data path* de 32-bits; 32 registradores de propósito geral; 32 fontes de interrupções externas; instruções dedicadas ao cálculo de multiplicações com 64-bits e 128-bits; acesso a uma variedade de periféricos *on-chip*, e interfaces para acesso a memórias e periféricos *off-chip*; oferece cerca de 2 GBytes de espaço de endereçamento; e customização de até 256 instruções.

O fabricante oferece três linhas de processadores, com características diferentes: Nios II/f (versão rápida), Nios II/e (versão econômica) e Nios II/s (versão padrão).

As versões Nios II/s e Nios II/f oferecem ainda respectivamente 5 e 6 estágios de *pipeline*, predições de salto estático e dinâmico. Ambas possuem *cache* de instruções e somente a versão Nios II/f possui *cache* de dados, todas parametrizáveis. A versão Nios II/e não possui muitas características para o aumento de desempenho, entretanto possui um tamanho menor em elementos lógicos (LE - *Logic Elements*), podendo ser utilizado em quantidade maior em uma FPGA para o aumento do desempenho.

3. *embedded Configurable operating system* - eCos

A proposta oferecida por este trabalho, consiste na implementação de funcionalidades no sistema operacional eCos (*embedded Configurable operating system*) que habilitem o multiprocessamento utilizando o processador Nios II.

O eCos é um sistema operacional de tempo real, sob a licença pública geral - GPL (*General Public License*), desenvolvido para atender aplicações embarcadas. Possui um sistema configurável permitindo a parametrização de seus recursos para satisfazer requisitos específicos de uma determinada aplicação[Hat 2003] [Veer and Dallaway 2001] [Garnett et al. 2003] [Massa 2002].

O eCos tem sido muito difundido e atualmente suporta muitas arquiteturas, incluindo o processador Nios II. Apesar de comportar simultaneamente vários processadores em plataformas distintas, essa característica ainda não está empregada para o processador em questão.

4. Implementação do sistema

4.1. O *hardware*

A proposta inicial de *hardware* para avaliação da arquitetura (Figura 1), consiste em três processadores Nios II/s, uma memória SDRAM, um *Timer*, três componentes JTAG UARTs e um componente *Mutex*.

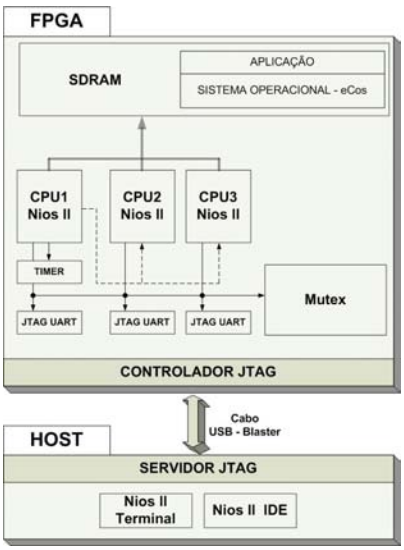

Figure 1. Arquitetura do sistema.

A plataforma utilizada foi a placa DE2, introduzida no mercado pela Altera, a qual possui grande capacidade lógica para implementação de sistemas lógicos programáveis. Sua principal finalidade é atender o mercado universitário, principalmente pela quantidade de funções disponibilizadas e pelo preço reduzido do produto. A FPGA utilizada pela placa consiste em Cyclone II 2C35, a qual possui capacidade 33.216 elementos lógicos, com osciladores de 50MHz e 27MHz para fontes de *clock* [de2 2006].

4.2. Processadores

Nesse modelo existem três processadores Nios II/s com mesma capacidade de processamento. Todos podem executar as *threads* e rotinas do sistema operacional. As distinções concentram-se na execução do processo de *boot* e das rotinas de interrupção de tempo.

A CPU1 é responsável por iniciar o sistema operacional e deixar o ambiente preparado para o início da execução dos demais processadores.

4.3. Interrupções

Na arquitetura de validação proposta, os principais componentes que geram interrupções são as ligações inter-CPUs, a interface JTAG UART e o *Timer*, apresentadas na Tabela 1.

Componente	Processadores/Prioridade		
	CPU1 (*master*)	**CPU2**	**CPU3**
Timer	0	-	-
JTAG UART	1	0	0
inter-CPUs	-	31	31

Table 1. Principais IRQs da arquitetura de validação.

O componente *Timer* emite uma interrupção a cada período de tempo pré-configurado em momento de projeto. Nesta arquitetura o período de clock é 1 *ms*. A CPU1 é o único processador que recebe eventos do *Timer*, e deve gerenciar a execução das tarefas do sistema relacionadas ao tempo. Com a centralização desse controle na CPU1, os demais processadores ficam livres para executar outras tarefas do sistema, sem preocupar-se com o tratamento da interrupção.

O componente JTAG UART é responsável pela interface entre o processador Nios II e o *host*. A utilização desse componente é obrigatório, e deve ser instanciado apenas um por processador.

A comunicação entre os processadores estabelecida consiste na faculdade de interrupção dos processadores escravos, pelo mestre. Essa interrupção é utilizada para sinalizar a necessidade de troca da *thread* corrente em execução. Isto pode ocorrer quando o tempo de execução dessa *thread* expira, ou quando existe outra com maior prioridade que deve ser executada naquele momento.

4.4. Memória

A memória de aplicações Nios II, em sistemas monoprocessados ou multiprocessados, é dividida nas seguintes seções: *.text*, *.rodata*, *.rwdata*, *heap* e *stack*, as quais são associadas a endereços fixos na memória, para cada processador. A divisão proposta para este projeto, consiste em um único conjunto contendo essas seções, conforme a Figura 2.

A seção *.text*, responsável pelo armazenamento do código executável, é única, de modo que todos os processadores podem compartilhar. Cada processador não deve executar programas diferentes em seções separadas, mas sim, executar os códigos de *threads* disparadas em um programa principal. Todo o multiprocessamento está diretamente relacionado à execução das *threads* de uma única aplicação. Assim, as seções *.rodata* e *.rwdata*, que armazenam respectivamente os dados constantes e variáveis do programa, também são fixas e únicas.

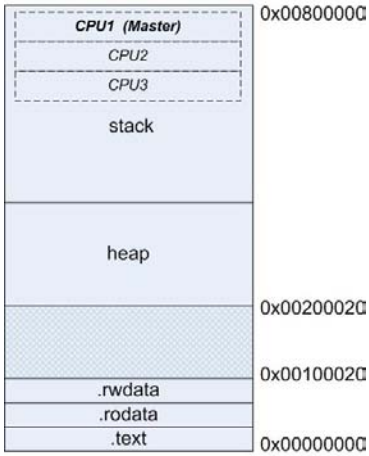

Figure 2. Disposição da memória.

Para a área de *heap*, existe um ponteiro (*__heap1*) no *linker script* que aponta para o fim da região *.rwdata*. Esse ponteiro determina o início da área de memória reservada à *heap*. Não existe um limite determinado para a área *heap*, sua extensão é determinada pelo início da área da pilha, indicada pelo ponteiro *cyg_interrupt_stack_base* no *linker script*.

No modelo implementado, cada processador possui sua própria seção *stack*. O registrador *sp* tem seu endereço decrementado a cada valor inserido na pilha. No momento de *boot*, a CPU recebe uma fatia de espaço, com tamanho fixo, utilizado para essa seção, de modo que o endereço inicial de *sp* está associado ao limite superior reservado à seção *stack*. Neste projeto, a fatia de memória reservada para a pilha de cada processador, consiste em um tamanho constante de 512 KB. Esse tamanho pode ser configurado para um valor menor, dependendo do número de processadores presentes no sistema.

O primeiro processador que executa o código do sistema operacional reserva a primeira fatia a partir do final da memória. Os valores são armazenados na pilha com endereço decrescente.

O intervalo de memória de 0x00100020 a 0x00200020 está reservado para a execução de uma aplicação de *jump* nos processadores *slave*, a qual desvia a execução para o início da memória onde estão posicionadas as instruções de inicialização do sistema operacional.

4.5. Sincronização

A base para implementação de primitivas de sincronização, normalmente consiste na instrução *Test-and-set*, a qual realiza uma leitura e escrita em uma área de memória atomicamente, disponibilizada por alguns processadores. Essa instrução é necessária para os métodos de exclusão mútua presentes no eCos, e seu suporte não é oferecido no processador Nios II.

Para a resolução desse problema, este projeto contempla a implementação de uma *macro* que realiza essa operação. A atomicidade é garantida por meio de um *mutex*, componente presente no *hardware* do sistema, que é utilizado somente para esse propósito.

O *mutex* em hardware é um componente utilizado para exclusão mútua em sis-

temas com diversos processadores Nios II. Somente um processador por vez acessa o *mutex* por meio de instruções *stwio* e *ldwio*. O componente possui os campos VALUE e OWNER. Quando VALUE tem valor zero, o *mutex* pode ser escrito por qualquer processador, incrementando esse valor. Quando um processador consegue escrever um valor no campo VALUE, seu *CPU-ID* é gravado no campo OWNER e somente esse processador terá direito de escrita nesse componente. Para um processador conseguir acesso, ele primeiramente tenta escrever um valor, e depois verifica se adquiriu a propriedade do componente por meio do campo OWNER. Se OWNER tem o valor de seu ID, significa que ele pode acessar a área de código em questão. Caso contrário não tem acesso.

Esse componente foi utilizado neste sistema operacional para implementar a instrução *Test-and-set* (*macro* TAS). No momento da execução da *macro*, todas as interrupções do processador corrente são desabilitadas, evitando que o *mutex* se torne propriedade de um processador que está atendendo uma interrupção. Posteriormente, o *mutex* é consultado, aguardando a liberação caso esteja ocupado. O valor atual da variável passada para a macro é armazenado e atualizado para 1. O valor antigo é retornado, o *mutex* é liberado e as interrupções são novamente habilitadas.

4.6. Sistema operacional

4.6.1. *Boot*

A inicialização do processador para o carregamento do sistema operacional é realizado no momento que o processo é carregado na memória. Um sinal é enviado ao processador para começar a executar.

A seqüência de inicialização do eCos está programada em *assembly* no arquivo *vector.s*. Após a compilação do sistema operacional juntamente com a aplicação, a parte binária relacionada a esse arquivo é disposta na memória logo após a seção do controle de interrupções. Assim, esse código é o primeiro executado pelo processador quando o processo é carregado.

A seguinte seqüência de inicialização do sistema operacional é executada:

1. Inicialização da *cache* e pilha: A *cache* de dados é inicializada para remover todas as referências anteriores, evitando incoerência de dados. Entretanto, esse procedimento não é necessário para o processador Nios II/s (padrão), que não possui *cache* de dados. A inicialização da *cache* não é utilizada neste projeto. Após esse passo, cada CPU recebe um espaço na memória destinado à pilha. O particionamento é realizado com um tamanho fixo de 512 KB para cada processador. O bloco com endereço de ordem mais significativa é reservado ao primeiro processador que executa esse processo, a CPU *master*. A pilha é iniciada a partir do endereço limite superior reservado para o processador.
2. Chaveamento das CPUs: Todos os processadores, com exceção da CPU *master*, ficam em espera ociosa até o momento de continuarem executando. A CPU *master* executa o restante da inicialização normalmente, ativando o escalonador e liberando as outras CPUs para execução. O controle de qual processador pode executar é realizado por meio de uma variável, na qual cada bit corresponde ao CPU-ID de um determinado processador. Um bit habilitado indica que a CPU, cujo CPU-ID correspondente à sua posição, deve sair da espera ociosa e continuar

a execução. A CPU *master* libera a execução do restante das CPUs somente após a ativação do escalonador.

3. Inicialização do eCos: Os objetos do sistema operacional são criados (escalonador, interrupção, clock, entre outros), as rotinas para atendimento de interrupção de tempo são associadas às IRQs correspondentes, e uma chamada é realizada à função *cyg_start*.
4. Início da execução da aplicação: Quando a função *cyg_start* é chamada, a rotina principal (*cyg_user_start*[1]) da aplicação é invocada fazendo inicializações, criando threads e primitivas de sincronização e, se necessário, instalando rotinas de interrupção. Após sua execução, o escalonador é iniciado, habilitando as interrupções e liberando os demais processadores para a execução.
5. SMP *Startup*: Realiza a inicialização de algumas variáveis relacionadas ao processador que está em execução, instala as rotinas de atendimento à interrupção inter-CPUs correspondente à IRQ 31 e inicia a execução das *threads* criadas pelo processador *master*.

4.6.2. Escalonamento

O eCos possui dois algoritmos de escalonamento implementados atualmente, dentre os quais um algoritmo específico pode ser utilizado de acordo com a configuração desejada. Os dois métodos de escalonamento são *bitmap* e MLQ (*multi-level queue*). Atualmente, o SMP é suportado utilizando somente o algoritmo MLQ, utilizado neste projeto.

O sistema operacional recebe uma interrupção de tempo (*tick*) a cada 1 *ms*. Essa interrupção é utilizada para o controle dos elementos de tempo real do eCos (*Counters*, *Clocks*, *Alarms* e *Timers*), como também do escalonamento das *threads* (*timeslice*).

O eCos possui uma constante chamada CYGNUM_KERNEL_SCHED_TIMESLICE_TICKS, a qual armazena o número de *ticks* de duração para a execução de uma *thread* antes que seja ativado o escalonamento de outra. Neste projeto essa constante corresponde ao valor 1000, de modo que as *threads* em execução são trocadas a cada 1 segundo. Essa troca é realizada por meio de interrupção inter-CPUs. A CPU *master* envia um pulso por meio de seu PIO de saída com o número correspondente da CPU cuja *thread* deve ser trocada.

O kernel foi programado para ativar o *timeslice* somente quando todas as CPUs entrarem em execução. Essa decisão de projeto está implementada para evitar problemas de troca de *threads* no momento em que outra CPU inicia sua execução do sistema operacional.

4.6.3. Interrupções de *hardware*

Para o controle de interrupção no eCos, cada vetor de interrupção é associado com uma ISR (*Interrupt Service Rotine*), a qual é executada sempre que um evento de *hardware* ocorre.

[1]A função *cyg_user_start()* normalmente é utilizada em aplicações para o eCos como substituta ao *main()*.

Entretanto, uma ISR não está apta a executar todos os serviços oferecidos pelo *kernel*, somente uma pequena parte deles são disponibilizados às ISRs. Uma ISR não pode desbloquear um processo. Quando é detectado o final de uma operação de E/S a ISR pode iniciar a execução de uma DSR (*Deferred Service Routine*), a qual está apta a executar mais rotinas do *kernel*.

Em qualquer evento de interrupção o processador deve executar a função *_exception_vector* presente no *vector.s*. Seu código binário está posicionado na memória no *exception address* do processador *master*.

Essa função examina os *bits* dos registradores *estatus* e *ipending*, verificando se a interrupção foi de *software* ou *hardware*. No eCos existe um vetor de ponteiros para funções chamado *hal_vsr_table*, cujos índices 0 e 1 apontam respectivamente para rotinas de interrupção de software (*_software_exception_handler*) e *hardware* (*_interrupt_handler*), declaradas em *vector.s*.

O eCos possui outro vetor, *hal_interrup_handlers*, que armazena a estrutura de tratamento de interrupções para cada IRQ. A função *_interrupt_handler* salva o contexto de execução e acessa o vetor *hal_interrup_handlers* utilizando como índice o número da interrupção (IRQ). A função ISR correspondente é executada.

Posteriormente, a função *_interrupt_end* é iniciada. Essa rotina constitui parte do controle de interrupção do *kernel*. Parte de seu código é protegido pelo chaveamento do escalonador: *cyg_scheduler::lock()*. A chamada ao *cyg_scheduler::unlock()* inicia a chamada a todas as DSRs pendentes, inclusive a DSR correspondente à interrupção em questão.

Após essa seqüência, é realizada uma verificação da necessidade de escalonamento de uma nova *thread* para a execução. O escalonamento pode ocorrer por dois motivos: tempo de execução da *thread* alcançado e chegada de uma *thread* de maior prioridade. Se uma nova *thread* é selecionada, o contexto da corrente é armazenado em sua própria pilha, e o contexto da nova *thread* é carregado. Caso contrário, o contexto da *thread* atual salvo antes da execução da interrupção é carregado, de modo que a execução retorna ao ponto onde foi interrompida.

Quando os processadores recebem uma interrupção cuja IRQ é 31, executam as funções ISR e DSR responsáveis pela troca de *thread* devido ao tempo de execução excedido. Essa interrupção é enviada aos processadores pela CPU *master* que controla o escalonador. A função DSR, relacionada a essa interrupção, executa a rotina *cyg_scheduler_timeslice_cpu* que seleciona uma nova *thread* para execução. Ao final da execução de *cyg_scheduler::unlock()*, os contextos são trocados e a nova *thread* entra em execução.

4.6.4. Coerência de dados em *cache*

A utilização de processadores com *cache* de dados carece de mecanismos que promovam a coerência dos dados lidos e armazenados na memória. Dos três tipos de processadores, o Nios II/s foi utilizado neste projeto, mas somente o Nios II/f possui *cache* de dados.

Neste caso, como não existe nenhum componente da Altera que forneça mecanis-

mos para coerência, existem dois métodos que poderiam ser utilizados para evitar que os dados fossem armazenados ou buscados na *cache* do processador:

1. *Bit* 31 dos endereços de memória: O *bit* mais significativo dos endereços de memória é utilizado para ativar o *cache bypass*. Todas as referências a endereços com esse *bit* ativado não são buscados e nem armazenados na cache.
2. Instruções *ldwio/stwio*: Existe um parâmetro de compilação *-mbypass-cache* que troca todas as intruções *ldw/stw* por *ldwio/stwio*. Essas instruções são interpretadas pelo processador como instruções de acesso a dispostivos de I/O, enquanto que *ldw/stw* são utilizadas especificamente para acesso à memória. Na execução das instruções *ldwio/stwio*, o processador despreza a existência da *cache* e envia os dados, ou requisita-os, diretamente pelo barramento.

5. Resultados

A validação desse projeto de suporte a SMP no eCos foi justificada por meio da implementação de alguns algoritmos paralelos para a prova de conceito. Dentre os quais estão o algoritmo PSRS - *Parallel Sorting by Regular Sampling* e o algoritmo de multiplicação de matrizes. Nesta seção a execução dos dois algoritmos está descrita.

5.1. O algoritmo PSRS

O PSRS consiste em um algoritmo de ordenação que considera uma arquitetura multiprocessada com p processadores [Shi and Schaeffer 1992] [Li et al. 1993].

Os tempos resultantes de execução desse algoritmo nos três processadores são comparados com os tempos do algoritmo Quicksort em um processador.

Para a implementação, foram utilizados dois vetores de inteiros armazenados na *heap*, os quais possuem o mesmo número de elementos. Esses vetores são inicializados com valores aleatórios, sendo que ambos possuem o mesmo conjunto de elementos.

A memória disponível para o armazenamento dos vetores inicia a partir do endereço 0x00200020, devido à área reservada para a aplicação de *jump*. O endereço final consiste no limite entre a pilha e a *heap*. Para os testes, a pilha de cada processador está configurada para 1 KB, a área reservada para a pilha é de 3072 Bytes. A memória possui 8 MB (seu endereço final é 0x00800000), de modo que o início da área da pilha está no endereço 0x007FF400. O espaço livre para o armazenamento de dados é (0x007FF400 - 0x00200020) = 0x5FF3E0 (6288352) Bytes. Considerando dois vetores de inteiros, sendo 4 Bytes cada elemento, poderiam ser armazendos 786044 elementos para cada vetor.

São iniciadas três *threads*, cada uma é executada em um processador. A execução do algoritmo somente é iniciada quando os três processadores estão iniciados, devido a ativação do *timeslice*. Como o evento de *timeslice* envolve operações de I/O, os resultados da execução de algoritmos sem sua ativação podem ser muito melhores. Esse planejamento assegura que os resultados obtidos pelo algoritmo Quicksort são compatíveis com os resultados do PSRS, com a mesma condição do ambiente.

A estimativa do tempo de execução dos algoritmos foi realizada por meio da função *cyg_current_time()*, a qual retorna o número de *ticks* corrente da CPU *master*. O período entre cada *tick* é 1 *ms*. Sendo assim, visando uma estimativa aproximada,

considera-se o retorno da função *cyg_current_time()* sendo o tempo do sistema mensurado em milissegundos.

As métricas utilizadas para análise da qualidade do paralelismo são *Speedup* e eficiência. O *Speedup* é utilizado para verificar o ganho de desempenho obtido com o uso de uma aplicação paralela em relação à aplicação seqüencial mais rápida que executa a mesma tarefa. A eficiência estima o aproveitamento de tempo do algoritmo paralelo nos processadores [Quinn 1994].

Os algoritmos Quicksort e PSRS foram executados sobre dois vetores distintos. Nas estimativas, os tamanhos dos vetores variam de 50 a 700000 elementos. Os experimentos foram executados dez vezes para cada número de elementos. O tempo resultante é a média dos tempos alcançados nas dez execuções. Os resultados estão apresentados na Tabela 2.

Elementos	**Quicksort** (*ms*)	**PSRS** (*ms*)	**Desvio Padrão Quick** (*ms*)	**Desvio Padrão PSRS** (*ms*)	**Speedup** ($S(3)$)	**Eficiência** ($E(3)$)
50	1.9	2	0.32	0.00	0.950	0.317
100	2.6	2.7	0.52	0.48	0.963	0.321
500	12	7.5	0.00	0.53	1.600	0.533
1000	27	16.7	0.00	0.48	1.617	0.539
5000	141.2	89.6	0.42	0.52	1.576	0.525
10000	332.6	207.1	0.52	0.32	1.606	0.535
50000	1868.7	1108.1	0.48	0.57	1.686	0.562
100000	3840.4	2329	0.52	1.33	1.649	0.550
200000	8425.6	4815.7	0.97	1.49	1.750	0.583
300000	13379.3	7500.3	0.48	1.49	1.784	0.595
400000	18152.2	10105.4	0.79	2.46	1.796	0.599
500000	24627	13427.8	0.67	4.32	1.834	0.611
600000	29657	15752	1.15	2.91	1.883	0.628
700000	35770.8	18972.8	0.42	3.36	1.885	0.628

Table 2. Tempos de execução dos algoritmos de ordenação.

De acordo com os resultados alcançados, o algoritmo PSRS tem seu aproveitamento relacionado com o número de elementos ordenados. O aumento da quantidade de elementos ocasionou um aumento nos valores de índices para *Speedup* e Eficiência. Os melhores índices foram alcançados para conjuntos de valores com os maiores números de elementos.

Os algoritmos poderiam ter tempos menores de execução se a interrupção de tempo fosse desabilitada. Entretanto, essa interrupção é necessária para mensurar o próprio tempo de execução por meio da função *cyg_current_time()*.

O armazenamento dos dois vetores em memórias *on-chip*, localizadas na própria FPGA, também acarretaria menores tempo de execução. Se os elementos lógicos dessa memória fossem utilizados para compor outros processadores no sistema, isso também poderia melhorar os tempos de execução.

5.2. O algoritmo de multiplicação de matrizes

Outro algoritmo utilizado para a validação da implementação é a multiplicação de matrizes. Para os testes, duas matrizes quadradas de elementos inteiros foram multiplicadas, cujo N variou de 5 a 600.

Foi alocado espaço na memória para três matrizes, duas delas foram iniciadas com valores aleatórios e a terceira é a matriz resultante da multiplicação. O armazenamento

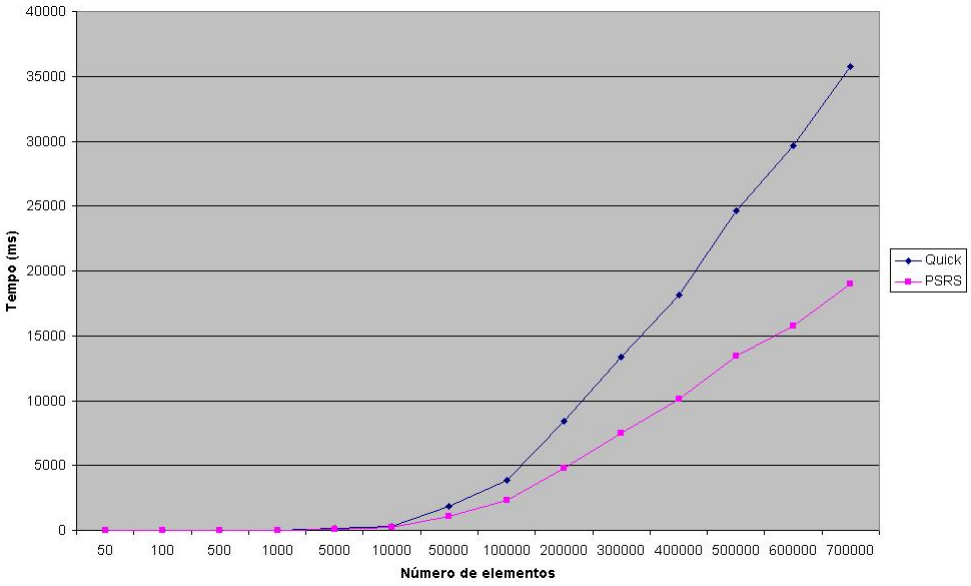

Figure 3. Tempos de execução dos algoritmos PSRS e *Quick Sort*.

é realizado a partir do endereço 0x00200020 e o número N máximo é 600, para evitar ultrapassar área máxima de 0x5FF3E0 Bytes disponível para armazenamento.

Na execução foram considerados os mesmos fatores descritos para o algoritmo PSRS. A execução seqüencial somente é iniciada quando os três processadores são iniciados, pois neste momento o *timeslice* é ativado. Deste modo, o algoritmo seqüencial possui as mesmas condições de execução do algoritmo paralelo de multiplicação.

Após a execução do algoritmo seqüencial é inicada a multiplicação paralela. A estimativa de todos os tempos de execução foi realizada por meio da função *cyg_current_time()*, sendo que os algoritmos foram executados dez vezes para cada N, e os valores considerados consistem em médias dessas execuções. Os resultados estão apresentados na Tabela 3.

Elementos (N)	**Multiplicação Seqüêncial** (*ms*)	**Multiplicação Paralela** (*ms*)	**Desvio Padrão Seqüêncial** (*ms*)	**Desvio Padrão Paralela** (*ms*)	**Speedup** ($S(3)$)	**Eficiência** ($E(3)$)
5	1	1	0.00	0.00	1.000	0.333
10	3	1	0.00	0.00	3.000	1.000
20	17	9.4	0.00	0.84	1.809	0.603
40	126.6	83.8	1.96	0.42	1.511	0.504
100	2012.2	1288.5	0.42	21.01	1.562	0.521
200	16225.3	9208.2	36.00	365.54	1.762	0.587
300	54829.8	34785.6	0.42	183.02	1.576	0.525
400	129961	86979.4	0.67	48.24	1.494	0.498
500	253897.2	151716	6.75	85.21	1.674	0.558
600	410572.2	184469.4	14.87	6249.82	2.226	0.742

Table 3. Tempos de execução dos algoritmos de multiplicação de matrizes.

De acordo com os resultados obtidos(Figura 4), a variação do *speedup* e da eficiência não é crescente de modo contínuo para o aumento de N. Entretanto, pode ser notado que esses valores são maiores para valores maiores de N. Quando N é 600, o valor de *speedup* é 2.226 e a eficiência é 0.742, denotando um maior ganho e um maior aproveitamento dos processadores em relação aos resultados alcançados com o algoritmo

de ordenação paralela.

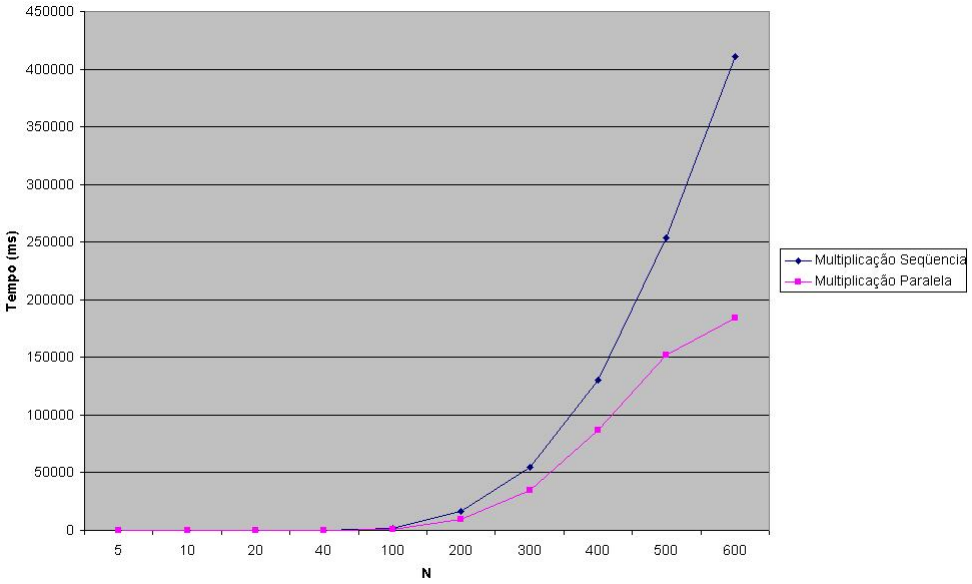

Figure 4. Tempos de execução dos algoritmos de multiplicação de matrizes Seqüencial e Paralela.

6. Conclusão

Este artigo apresentou a descrição da implementação de suporte a SMP no sistema operacional eCos para o processador Nios II. Este trabalho foi motivado, entre outros fatores, pela possibilidade de utilizar a capacidade da FPGA para comportar diversos processadores executando tarefas paralelamente, especificamente algoritmos inerentes à robótica, os quais necessitam de mais de um processador Nios II para alcançarem um desempenho aceitável [Gates 2007]. Em virtude disso, utiliza-se a arquitetura paralela SMP, a qual apresentou melhores condições para implementação entre os modelos analisados.

Para validação do projeto, foram implementadas algumas aplicações paralelas, dentre as quais está o algoritmo de ordenação paralela PSRS e o algoritmo de multiplicação de matrizes. De acordo com os resultados obtidos, o modelo SMP, com os três processadores gerenciados pelo sistema operacional eCos, obteve um rendimento compatível com o número de elementos utilizados nos algoritmos. No PSRS, quanto maior o número de elementos utilizados, maiores os índices de *speedup* e eficiência alcançados. Os melhores resultados foram 0.628 para eficiência e 1.885 para *speedup*, na ordenação de 700000 elementos. Na multiplicação de matrizes os melhores resultados alcançados foram 0.742 para eficiência e 2.226 para *speedup* com N igual a 600, os quais demonstram a capacidade de processamento da arquitetura implementada.

Desta forma, este projeto contribui para a utilização da flexibilidade do software unida ao desempenho alcançado no paralelismo entre diversos processadores implementados em hardware na FPGA. Tal implementação pode ser utilizada para finalidades diversas, entre as quais podem ser citadas sistemas de robótica embarcados, cujos algoritmos carecem de grande capacidade de processamento e de concorrência no processamento de sensores.

References

(2006). *DE2 Development and Education Board: User Manual*. ALTERA Corporation, San Jose, CA.

Altera (2004). The industry s fastest fpgas. Disponível em: http://www.altera.com/-products/devices/stratix2/features/performance/st2-performance.html. Acesso em março de 2005.

Altera (2005). Fpga, cpld, and structured asic devices; altera, the leader in programmable logic. Disponível em http://www.altera.com. Acesso em fevereiro de 2005.

Altera (2006). Stratix iii fpgas. http://www.altera.com/products/devices/stratix3/st3-index.jsp. Acesso em fevereiro de 2007.

Garnett, N., Larmour, J., Lunn, A., Thomas, G., and Veer, B. (2003). ecos reference manual. Disponível em: http://ecos.sourceware.org/docs-2.0/pdf/ecos-2.0-ref-a4.pdf. Acesso em fevereiro de 2005.

Gates, B. (2007). A robot in every home. *Scientific American*. http://www.sciam.com/article.cfm?chanID=sa006&colID=1&articleID=9312A198-E7F2-99DF-31DA639D6C4BA567. Acesso em fevereiro de 2007.

Hat, R. (2003). ecos user guide. Disponível em: http://ecos.sourceware.org/docs-2.0/pdf/ecos-2.0-user-guide-a4.pdf. Acesso em fevereiro de 2005.

Li, X., Lu, P., Schaeffer, J., Shillington, J., Wong, P. S., and Shi, H. (1993). On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10):1079–1103.

Massa, A. J. (2002). *Embedded Software Development with eCos*. Prentice Hall, 1th edition edition.

Quinn, M. J. (1994). *Parallel Computing*. McGraw Hill, New York, 2 edition.

Shi, H. and Schaeffer, J. (1992). Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372.

Veer, B. and Dallaway, J. (2001). The ecos component writer´s guide. Disponível em: http://ecos.sourceware.org/docs-2.0/pdf/ecos-2.0-cdl-guide-a4.pdf. Acesso em fevereiro de 2005.

Xilinx (2005). Xilinx: Programmable logic devices, fpga and cpld. Disponível em http://www.xilinx.com. Acesso em fevereiro de 2005.