SEMPRE: Uma Arquitetura SuperEscalar com Múltiplos PRocessos em Execução.

Ronaldo A. L. Gonçalves[1]
Departamento de Informática / UEM / Brasil
e-mail: ronaldo@din.uem.br

Philippe O. A. Navaux
Instituto de Informática / UFRGS / Brasil
e-mail: navaux@inf.ufrgs.br

Resumo

Este trabalho apresenta a arquitetura SEMPRE, projetada para executar múltiplos processos, ao invés de múltiplas *threads*, aproveitando assim a grande quantidade de processos existentes nas estações de trabalho compartilhadas e servidores de rede. Esta arquitetura emprega mecanismos inovadores que possibilitam a antecipação da troca de contexto entre processos, bem como a rápida remoção de instruções inconvenientes. Além disso, ela provê facilidades para o sistema operacional gerenciar processos com mínimo esforço.

palavras-chaves: superescalar, *multi-threading*, gerenciamento de processos

Abstract

This work presents the SEMPRE architecture, designed to running multiple processes rather than multiple threads taking advantage of the great number of existing processes on shared workstations and network servers. This architecture uses novel mechanisms that allow anticipation of contextual change between processes as well as the fast removal of undesirable instructions. Moreover, it provides facilities for the operating system to manage processes with minimal effort.

key-words: superscalar, multi-threading, processes management

1. Introdução.

Com o passar dos tempos, as aplicações ficam maiores e mais complexas, exigindo que os processadores fiquem mais rápidos. Para satisfazer estas exigências, os processadores de última geração têm sido projetados como arquiteturas superescalares, onde podem ser destacados o *Pentium* [ANDE95], o *PowerPC* [CHAK94] e o MIPS R10000 [MIPS95]. Estas arquiteturas possuem recursos para executar várias instruções por ciclo (ipc) das aplicações convencionais (*single-thread*), muito embora, diversos trabalhos têm mostrado que o paralelismo disponível nestas aplicações possui um limite que não pode ser vencido pelas arquiteturas superescalares tradicionais: o limite do fluxo de dados.

Jouppi e Wall [JOUP89] mediram um paralelismo variando de 1.6 até 3.2 ipc. Butler, Yeh e Patt [BUTL91] concluíram que se o *hardware* estiver corretamente balanceado, o paralelismo pode atingir de 2 até 5.8 ipc. Tran e Wu [TRAN92] não conseguiram obter paralelismo maior que 2 ipc, para *benchmarks* reais. O trabalho mais completo e com resultados mais otimistas foi desenvolvido por Wall [WALL93] através de simulações sobre 18 diferentes programas compilados para código MIPS, com 375 modelos diferentes de análise de paralelismo disponível. Para a maioria dos programas simulados por Wall, o paralelismo médio variou de 4 a 10 ipc, usando técnicas conhecidas de extração de paralelismo, mas não-triviais.

[1] Doutorando no CPGCC (Instituto de Informática) da Universidade Federal do Rio Grande do Sul
e-mail: ronaldog@inf.ufrgs.br

Sabe-se que, o paralelismo ao nível de instrução é limitado basicamente por 3 fatores: dependência de controle, dependência de dados verdadeira (fluxo de dados) e dependência de recursos de *hardware* (falsas dependências, conflitos de portas, barramento etc.). Destes 3 fatores, a dependência causada pelo fluxo de dados não consegue ser resolvida por nenhuma arquitetura atual. Em uma atitude de extremo exagero, as dependências de recursos e de controles podem ser eliminadas se a arquitetura for projetada com recursos de *hardware* infinitos e se todos os caminhos dos desvios, tomados e não tomados, forem executados simultaneamente. Obviamente, isto não é viável e nem ao menos possível, mas mesmo assim, as dependências de controles podem ser quase eliminadas usando boas técnicas de previsão de desvios, e as dependências de recursos podem ser satisfatoriamente reduzidas com um bom balanceamento da arquitetura e uso de renomeação de registradores. Mas as dependências verdadeiras sempre continuarão.

Preocupados com o futuro, onde os limites do paralelismo poderão ser causados somente pelo fluxo de dados, Lipasti e Shen [LIPA96] estão desenvolvendo trabalhos que envolvem a especulação de dados. Eles foram motivados à estes estudos após terem concluído que os processadores superescalares modernos, apesar de toda a tecnologia envolvida, são capazes de executar somente de 0.5 até 1.5 instruções por ciclo. Lamentavelmente, a especulação de dados é útil somente para dados altamente previsíveis, tais como constantes e valores incrementais, servindo apenas para aliviar a problemática.

A teoria de Park [PARK91] é mais pessimista: "A performance além do limite de 2.5 ipc, para a maioria das aplicações não-científicas e de propósito geral, não pode ser alcançada com um único fluxo de instruções". Obviamente, isto sugere o uso de arquiteturas *multi-threading*, onde o paralelismo pode ser extraído de múltiplos fluxos de instruções.

2. Arquiteturas *Multi-Threading*: O Estado da Arte.

As arquiteturas *multi-threading* alcançam alto grau de ipc, escalonando instruções provenientes de diferentes *threads*, e possuem a habilidade de esconder as latências de acesso à memória tão bem quanto as latências causadas pelas dependências entre as instruções. Estas arquiteturas podem ser estruturadas de diferentes formas [TULL95].

Laudon, Gupta e Horowitz [LAUD94] propuseram uma técnica de execução *multi-threading*, chamada *Interleaving*, que pode ser aplicada em processadores superescalares tradicionais, para permitir que os mesmos executem aplicações tanto *single-thread* quanto *multi-threading*, sem a adição de *hardware*. No extremo oposto, Govindarajan e Nemawarkar [GOVI92] projetaram um multiprocessador chamado SMALL, composto de várias unidades de processamento independentes, para executar *multi-threading*. Mas a maioria das propostas de arquiteturas *multi-threading* se baseia na replicação das estruturas de armazenamento existentes em uma arquitetura superescalar tradicional, para poder suportar múltiplas *threads*. Desta forma, Hirata e outros [HIRA92] desenvolveram uma arquitetura *multi-threading* que utiliza bancos de registradores, filas de instruções e contadores de programa, todos específicos para cada *thread* em execução. Também, Wallace, Calder e Tullsen [WALL98] projetaram uma arquitetura *multi-threading* com várias tabelas de renomeação de registradores, para executar ambos os caminhos dos desvios condicionais, quando houver mais *hardware* do que *threads* disponíveis.

Apesar de todos estes trabalhos, o desenvolvimento de arquiteturas *multi-threading* é desmotivado pela falta de aplicações com múltiplas *threads*, pois a maioria esmagadora das aplicações existentes é seqüencial. Por outro lado, grande parte dos processadores atuais é utilizada em estações de trabalho ou servidores de rede, com seus recursos compartilhados por

diferentes aplicações. Estas aplicações em conjunto com o próprio sistema operacional formam um conjunto de processos em execução, que pode ser chamado de sistema multi-processos, onde muito do tempo de execução destes processadores é gasto com o gerenciamento destes processos, principalmente com o escalonamento e as trocas de contexto.

Visando desenvolver processadores voltados para a execução de sistemas multi-processos, o presente trabalho apresenta a arquitetura SEMPRE. Esta arquitetura não deixa de ser *multi-threading* e nem tão pouco superescalar, mas caracteriza-se pela habilidade adicional de extrair o paralelismo existente entre processos, além de executar diretamente pelo *hardware*, muitas das operações (onerosas em consumo de tempo de cpu) normalmente executadas pelo sistema operacional. A arquitetura SEMPRE é detalhada a seguir.

3. A Arquitetura SEMPRE

A arquitetura SEMPRE (SuperEscalar com Múltiplos PRocessos em Execução) é vista na figura 1, que mostra seus principais componentes funcionais e estruturais. Seu *pipeline* superescalar contém 5 estágios funcionais (busca, decodificação, execução, término e conclusão), providos de técnicas usuais de previsão de desvios, execução especulativa, renomeação simplificada de registradores e execução fora-de-ordem. Os estágios desta arquitetura são discutidos nas próximas sub-seções.

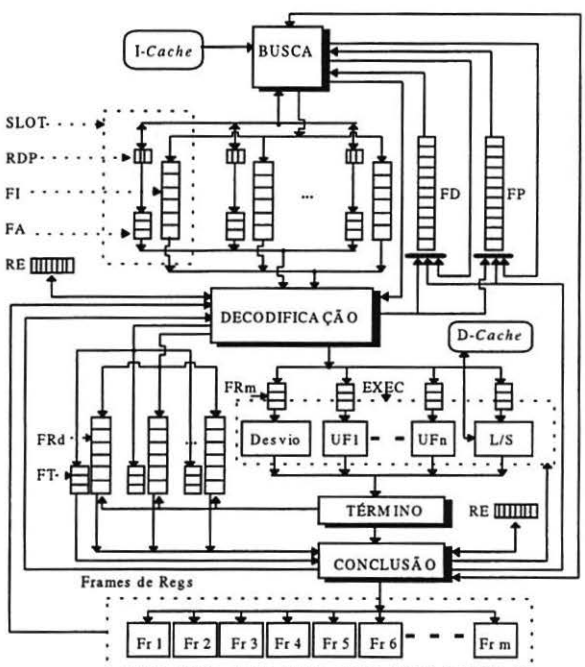

Figura 1: Arquitetura SuperEscalar com Múltiplos PRocessos em Execução

3.1. Estágio de Busca.

As principais funções do estágio de busca são: busca de instruções da *cache*, previsão de desvios, troca de contexto e detecção e execução de instruções privilegiadas. Durante a busca de instruções na *cache*, este estágio busca um bloco de instruções por vez, para cada *slot* da arquitetura, no estilo *round-robin*. Este bloco deve conter pelo menos N*T instruções, onde N indica o número de *slots* existentes e T o tempo médio necessário para buscar o bloco na *cache*, assim como sugerido por Hirata em [HIRA92]. Isto tende a favorecer que cada *slot* permaneça com instruções até a próxima busca. Uma vez selecionado o *slot*, a busca é feita a partir do endereço contido no campo CP (contador de programa) do registrador RDP (registrador descritor de processo), e as instruções buscadas são inseridas na respectiva fila de instruções (FI). A figura 2 mostra o estágio de busca e alguns dos seus relacionamentos.

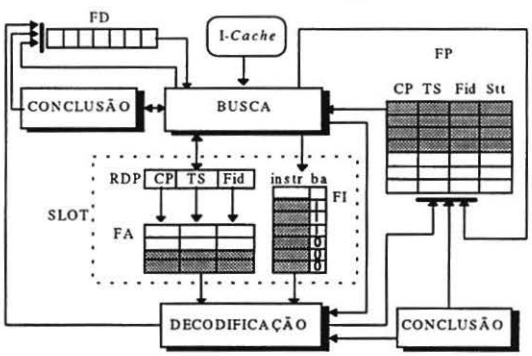

Figura 2: Esquema do Estágio de Busca

Para suportar execução especulativa, o estágio de busca prevê desvios condicionais utilizando um mecanismo baseado em um *bit* de história, que obtém 90,23% de acerto nas previsões segundo Smith [SMIT81]. A principal vantagem deste mecanismo, em arquiteturas *multi-threading*, é que o *bit* de história é associado diretamente nas instruções de desvios, dentro da *cache*, eliminando assim a utilização de tabelas de previsão específicas para cada *thread*, o que causaria um aumento considerável nas lógica e estruturas de armazenamento.

Quando o estágio de busca detecta uma instrução de desvio condicional, ele consulta o *bit* de história na *cache* e faz a previsão. Se a previsão for de desvio tomado, somente as instruções até a instrução de desvio são inseridas na fila de instruções e a busca é redirecionada para o caminho previsto. A previsão feita (tomado ou não tomado) é armazenada juntamente com a instrução na fila de instruções. O campo CP do RDP também é atualizado para o endereço previsto. O estágio de conclusão verifica a corretude da previsão.

A troca de contexto ocorre quando o *slot* corrente ou está vazio ou quando a busca das instruções é interrompida. Esta operação não requer o esvaziamento do *slot* para o posterior preenchimento com outro contexto, como normalmente tem sido utilizado nas arquiteturas *multi-threading* propostas. Na arquitetura SEMPRE, as instruções do novo contexto são concatenadas, na seqüência, com as instruções do contexto antigo, ainda remanescentes no *slot*. Isto antecipa a busca do próximo contexto, e por conseqüência, maximiza a utilização das filas de instruções. O controle de quais contextos estão inseridos no *slot* é feito através da fila de ativos (FA), que contém uma entrada para cada contexto ativo do *slot*.

Sempre que é necessária a troca de contexto, o estágio de busca escalona um processo da fila de prontos (FP) (retira o primeiro da fila) e verifica o seu campo de *status* (Stt). Se o *status* indicar "processo morto", ele é descartado e seu *frame* é inserido na fila de *frames* disponíveis (FD). Se o *status* indicar "processo suspenso", ele é reinserido na fila de prontos. Mas se o *status* indicar "processo pronto", ele é inserido na fila de ativos e uma cópia do mesmo descritor é mantido no RDP, para as futuras verificações do estágio de busca.

O contador de programa do processo anterior é atualizado com o CP contido no RDP, antes do seu contexto ser trocado. Este processo ainda permanece na arquitetura, passando pelas fila de ativos e fila de processos em trânsito (FT), até que sua última instrução seja concluída, quando então ele volta para a fila de prontos. As novas instruções são inseridas na fila de instruções com o *bit* alternador inverso ao *bit* alternador do contexto anterior (veja figura 2). De uma forma geral, a troca de contexto pode ocorrer durante a ocorrência de i-*cache miss*, durante a ocorrência de instruções privilegiadas, mediante solicitação do estágio de conclusão e com o término do *time-slice* do processo.

Na arquitetura SEMPRE o estágio de busca executa algumas funções, antes executadas pelo sistema operacional. O sistema operacional gerencia os processos e o *hardware* apenas provê facilidades para tal. A figura 3 mostra os estados e transições de um processo ao nível de arquitetura. As instruções privilegiadas definidas inicialmente são.

- ***create* op1 op2 rde :** cria um novo processo - retira o Fid (identificador de *frame*) da fila de *frames* disponíveis e insere um novo processo na fila de prontos, contendo: o Fid, o contador de programa (em op1) e o *time-slice* (TS) (em op2). O Fid é retornado em rde (como identificador do processo).
- ***kill* op1 rde :** mata um processo - O processo (cujo Fid deve estar em op1) é eliminado da arquitetura. O rde retorna o sucesso da operação.
- ***suspend* op1 op2 rde :** suspende um processo - O processo (cujo identificador deve estar em op1) tem seu *status* marcado como "suspenso" e quando ele for escalonado é reinserido na fila de prontos. O rde retorna o sucesso da operação.
- ***resume* op1 rde :** reassume um processo suspenso - O processo (cujo identificador deve estar em op1) que está marcado como "suspenso" é simplesmente remarcado como "pronto". O rde retorna o sucesso da operação.

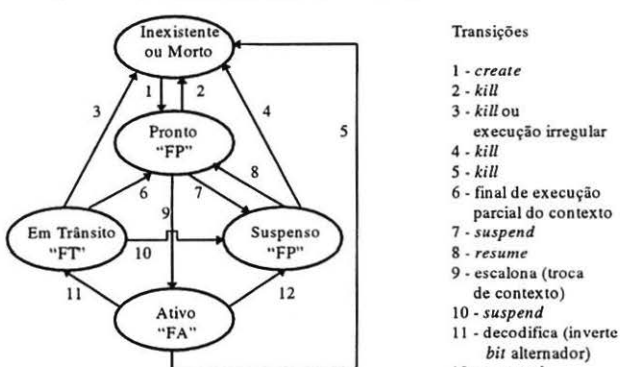

Figura 3: Diagrama de Estados dos Processos na Arquitetura SEMPRE

3.2. Estágio de Decodificação.

De uma forma geral, o estágio de decodificação executa três principais atividades em conjunto: escalonamento das instruções dos *slots*, decodificação (incluindo renomeação simplificada e leitura dos operandos) e despacho para as filas de remessa (FRm) das unidades funcionais. A figura 4 mostra os relacionamentos existentes no mecanismo de decodificação.

As instruções contidas nas primeiras entradas das filas de instruções dos *slots* são decodificadas e somente aquelas que estão prontas são despachadas. Isto evita a saturação das filas de remessa com instruções que não podem ser remetidas, aproveitando a quantidade de instruções prontas que é maior do que em arquiteturas superescalares tradicionais.

Quando uma instrução é despachada, a fila de instruções associada é deslocada e a próxima instrução é visualizada no início da fila de instruções. Quando o estágio de decodificação detecta uma inversão do *bit* alternador (ba), a fila de ativos também é deslocada, e a primeira entrada é então inserida no final da *fila de processos em trânsito* associada ao *slot* em questão. Deve-se observar que a primeira instrução de um novo contexto é despachada juntamente com o descritor do processo associado que está na fila de ativos.

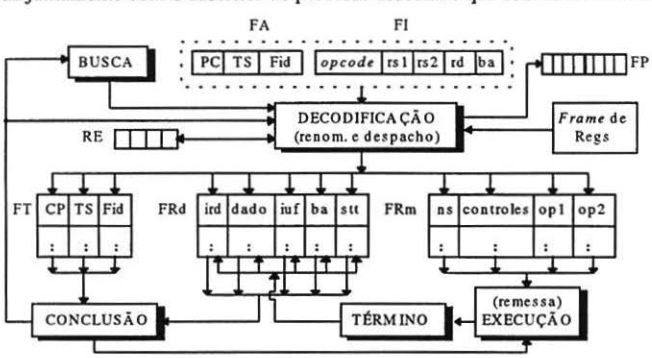

Figura 4: Esquema do Estágio de Decodificação

O algoritmo de escalonamento, chamado aqui de "*round-robin* ponderado", utiliza um registrador RE contendo um identificador para cada *slot* da arquitetura, juntamente com um *bit* de despacho, que informa se alguma instrução daquele *slot* já foi despachada alguma vez. Inicialmente, todos os *bits* de despacho contém 0 (zero), indicando que não houve nenhum despacho de nenhum *slot*. O algoritmo percorre o RE circularmente, e para cada *slot* ele verifica se existe instrução pronta, verificando os *bits* de ocupação dos registradores fontes. Também é verificado se existe entrada livre na fila de remessa apropriada.

As instruções selecionadas têm seus operandos lidos e são então despachadas, juntamente com o número do *slot* e os controles decodificados. Para cada *slot* de onde uma ou mais instruções são despachadas, o *bit* de despacho é marcado com 1. Após o despacho, o RE é reorganizado de forma que os *slots* que já tiveram instruções despachadas são colocados no final do RE, para permitir que os outros *slots* sejam os próximos a serem escalonados. Este algoritmo favorece os processos com maior número de instruções prontas, ao mesmo tempo que cede a todos os *slots* a oportunidade de despachar. Quando todos os *bits* de despacho do registrador RE estão com 1, eles são todos reinicializados novamente com 0.

Para cada instrução despachada, o registrador destino (rd) tem o seu *busy-bit* marcado como "ocupado". Além disso, é inserido uma entrada na fila de reordenação (FRd) associada ao *slot* da onde a instrução foi despachada.

Para eliminar as falsas dependências que existem entre as instruções prontas, é utilizado um mecanismo de renomeação simplificada, que mapeia os registradores destinos nas entradas das filas de reordenação (campo "dado"), semelhante a renomeação via *reorder buffer* descrito em [SMIT95]. As filas de reordenação servem somente para conter os dados gerados pelas instruções e garantir o controle das falsas dependências, pois isto elimina os conflitos entre os registradores destinos. Posteriormente, no estágio de conclusão, o dado da fila de reordenação é retirado e gravado no registrador destino.

3.3 Estágio de Execução.

Neste estágio, cada unidade funcional executa as instruções de sua respectiva fila de remessa, retirando-as em ordem através de deslocamento. Não existe nenhum outro tipo de escalonamento dinâmico, aqui neste estágio, pois as instruções já foram escalonadas durante o despacho. Se uma instrução causa uma exceção, tal como divisão por zero ou acesso não permitido, o resultado da instrução é enviado para o estágio de término como um código de erro. Este estágio também pode ser solicitado pelo estágio de conclusão para remover instruções inconvenientes.

3.4. Estágio de Término.

Este estágio atualiza as entradas das filas de reordenação para cada instrução que termina a sua execução, segundo o esquema de renomeação simplificada, da seguinte maneira: para cada unidade funcional que conclui a execução de uma instrução, a unidade de término procura (na ordem inversa a da inserção) na fila de reordenação associada ao *slot* da instrução concluída, a primeira entrada cujo campo "iuf" (identificador da unidade funcional) corresponde ao da unidade funcional em questão. Nesta entrada, ela grava o resultado da operação no campo "dado" e altera o campo de *status* "stt" para "instrução terminada".

Deve-se observar que as instruções despachadas para a mesma fila de remessa são inseridas e retiradas em ordem. Também, as instruções dentro de uma fila de reordenação estão na mesma ordem em que estavam dentro do *slot*. Assim, para uma instrução pertencente à um contexto posterior terminar a sua execução antes de outra instrução pertencente à um contexto anterior elas devem ser executadas por unidades funcionais diferentes. Como as instruções dentro de uma fila de reordenação são pesquisadas pelo identificador da unidade funcional executora "iuf", jamais haverá coincidência entre dois registradores, com os mesmos identificadores, utilizados por duas instruções de contextos distintos. Isto garante a eliminação das falsas dependências.

Em caso de exceção, o campo "dado" é preenchido com código de erro, a ser tratado na conclusão. Também, em caso de instrução de desvio, o resultado obtido (tomado ou não tomado) é comparado com a previsão inicial. Se os valores diferem, o campo "stt" da fila de reordenação é marcado com "previsão incorreta" para ser tratado pelo estagio de conclusão. Caso contrário, o campo "stt" é marcado de forma normal, com "instrução terminada".

3.5. Estágio de Conclusão.

Para cada fila de reordenação, este estágio verifica se as primeiras instruções estão "terminadas", analisando o campo "stt". Aqui também é utilizado o escalonamento "*round*

robin ponderado" apresentado na seção 3.2., para saber de quais filas de reordenação as instruções prontas devem ser retiradas primeiro, utilizando para isto um outro RE. Para cada instrução escolhida, a mesma é retirada da fila de reordenação, através de deslocamento, e o resultado (campo "dado") da instrução é gravado no registrador destino ("ird") do *frame* correspondente (campo "Fid" da primeira entrada da fila de processos em trânsito). Este mesmo registrador tem seu *busy-bit* marcado como "livre" e pode ser lido por outra instrução.

Quando termina a execução de um contexto (através da verificação da inversão do *bit* alternador na fila de reordenação correspondente), o processo é retirado da fila de processos em trânsito (deslocamento). Se por acaso, a unidade de busca solicitou que o processo fosse suspenso, seu campo de *status* é marcado para tal. Então, o processo é reinserido na fila de prontos. Neste estágio também são controladas a execução especulativa, ocorrência de exceções e mortes de processos.

3.6. Remoção de Instruções Inconvenientes.

Instruções inconvenientes são instruções que cujos resultados devem ser desconsiderados. Como estas instruções podem estar ocupando muitos dos recursos do *hardware* (que podem ser utilizados por instruções úteis), elas devem ser removidas o mais rapidamente possível. Estas instruções surgem em três situações:

1. Instruções especuladas incorretamente: são detectadas durante o estágio de conclusão, quando o resultado do desvio não confere com o resultado previsto.
2. Instruções seguintes à uma exceção: são detectadas no estágio de execução, quando a execução de uma instrução causa uma exceção.
3. Instruções de um processo morto pelo sistema operacional: são detectadas no estágio de busca, quando uma instrução *kill* é encontrada.

Deve-se considerar que as instruções inconvenientes estão misturadas pelo *pipeline* juntamente com outras instruções úteis, o que dificulta o processo de remoção. Mas a arquitetura SEMPRE possibilita a localização destas instruções através das diferentes filas de processos. Estas instruções podem estar em 3 regiões prováveis:

1. Quando o processo estiver na fila de prontos: neste caso as instruções estão dentro do *frame*, ou na memória (para alguns processos suspensos).
2. Quando o processo estiver na fila de ativos: neste caso as instruções estão todas em ordem dentro da fila de instruções, e podem ainda estar sendo buscadas pelo estágio de busca caso o RDP contenha também o descritor do processo em questão.
3. Quando o processo estiver na fila de processos em trânsito: neste caso muitas instruções podem estar nas filas de remessa e muitas já podem ter sido terminadas e seus resultados gravados na fila de reordenação associada. Além disso, muitas instruções podem ainda estar sendo buscadas e decodificadas.

A remoção das instruções destas filas é feita rapidamente deslocando-se todas as entradas até o aparecimento de um *bit* alternador invertido na instrução situada no início da fila, ou então, até o esvaziamento da fila. A comparação de um único *bit* é o principal motivo da rapidez desta operação.

3.7. Execução dos Múltiplos Processos.

A execução dos múltiplos processos na arquitetura SEMPRE começa quando ela entra em funcionamento. Neste momento, o *hardware* insere automaticamente, como primeira entrada na fila de prontos, o descritor do carregador do *Boot* do sistema operacional, que já está em memória ROM, e executa uma instrução *create* para iniciar sua execução. A partir daí, com o sistema operacional em execução, os demais processos podem ser criados.

Deve-se observar que ao nível de arquitetura, todos os processos criados pelo sistema operacional estão prontos para serem executados. Para algum processo ser suspenso, o sistema operacional deve executar explicitamente uma instrução *suspend*, e posteriormente uma instrução *resume*, se desejar que o processo seja novamente escalonado. Para a arquitetura, não interessa o motivo de uma suspensão, mas para o sistema operacional, ela pode indicar várias situações tais como: sincronização entre processos, troca de mensagens, espera de recursos diversos etc.

4. Conclusões e Trabalhos Futuros.

O presente trabalho apresenta uma arquitetura *multi-threading*, voltada para a execução de múltiplos processos, existentes em grande quantidade nas estações de trabalho compartilhadas e servidores de rede. Esta arquitetura é capaz de explorar o paralelismo dentro de cada processo, executando instruções prontas fora-de-ordem, tão bem quanto explorar o paralelismo entre vários processos, escondendo latências e aumentando o desempenho final do sistema.

As principais contribuições desta arquitetura são:

- A arquitetura é baseada em filas do tipo FIFO e algoritmos de escalonamento baseados em *round-robin*, provendo rapidez na execução e simplicidade da lógica de manipulação.
- A arquitetura contém um conjunto de instruções específicas para facilitar o desenvolvimento do sistema operacional. Estas instruções são simples e de fácil decodificação, e são resolvidas diretamente pelo estágio de busca de instruções. Isto permite ganho de tempo substancial, que tradicionalmente é perdido com gerenciamento de processos e troca de contexto ao nível de sistema operacional.
- A arquitetura suporta rápida remoção de instruções inconvenientes, quando solicitadas pelo sistema operacional, ou quando provindas de caminhos incorretos dos desvios ou de exceções. Estas características são suportadas graças ao uso de um *bit* alternador nas entradas das filas de instruções e de reordenação.
- A arquitetura suporta troca antecipada de contexto, tão bem quanto múltiplos processos compartilhando o mesmo *slot*, que são gerenciados através das filas de processos ativos, maximizando a utilização do *hardware*, que tradicionalmente abriga uma única *thread* por *slot*.

Como trabalhos futuros, os próximos grandes passos serão a análise e desempenho da arquitetura (utilizando diferentes parâmetros e algoritmos), o desenvolvimento de um sistema operacional e a implementação da arquitetura em *chip*. Os compiladores bem como as aplicações existentes não precisam ser alterados.

5. Referências Bibliográficas.

[ANDE95] Anderson, D. & Shanley, T., Pentium Processor System Architecture, MindShare, Inc., Addison-Wesley, Massachusetts, 433p., February, 1995.

[BUTL91] Butler, M., et all, Single Instruction Stream Parallelism Is Greater Than Two, Proceedings of the 18th Annual International Symposium on Computer Architecture, Toronto, Canada, May, 1991.

[CHAK94] Chakravarty, D. & Cannon, C., PowerPC: Concepts, Architecture, and Design, J. Ranade Workstations Series, McGraw-Hill, USA, Inc.,p.363, 1994.

[GOVI92] Govindarajan, R. & Nemawarkar, S. S., SMALL: A Scalable Multithreaded Architecture to Exploit Large Locality, Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing, Dallas, TX, Dec, 1992.

[HIRA92] Hirata, H. et all, An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads, Proceedings of the 19th Annual International Symposium on Computer Architecture, ACM & IEEE-CS, 1992.

[JOUP89] Jouppi, N. P. & Wall, D. W., Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines, Research Report, Digital, Western Research Laboratory, Palo Alto, California, July, 1989.

[LAUD94] Laudon, J., et all, Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations, Proceedings of the International Conference on ASPLOS, Oct, 1994.

[LIPA96] Lipasti, M.H. & Shen, J.P., Exceeding the Dataflow Limit via Value Prediction, 29th Micro, Paris, France, December, 1996.

[MIPS95] MIPS R10000 Microprocessor User's Manual, Version 1.0, MIPS Technologies, Inc. North Shoreline, Mountain View, California, June, 1995.

[PARK91] Park, W. W., et al, Performance Advantages of Multithreaded Processors, Proceedings of the International Conference on Parallel Processing, 1991.

[SMIT81] Smith, J. E., A Study of Branch Prediction Strategies, Proceedings of the 8th International Symposium on Computer Architecture - ISCA'81, Minneapolis, Minnesota, May, 1981.

[SMIT95] Smith, J.E & Sohi, G.S., The Microarchitecture of SuperScalar Processors, Proceedings of the IEEE, 83(12), pp.1609-1624, December, 1995.

[TRAN92] Tran, T. & Wu, C., Limitation of Superscalar Microprocessor Performance, Proceedings of the 25th Annual International Symposium on Microarchitecture, Portland, Oregon, Dec, 1992.

[TULL95] Tullsen, D. M., et all, Simultaneous Multithreading: Maximizing On-Chip Parallelism, Proceedings of the ISCA'95, Santa Margherita Ligure, Italy, Computer Architetcure News, n.2, v.23, 1995.

[WALL93] Wall, D. W., Limits of Instruction-Level Parallelism, Research Report, Digital, Western Research Laboratory, Palo Alto, California, June, 1993.

[WALL98] Wallace, S., Calder, B., Tullsen, D. M., Threaded Multiple Path Execution, Proceedings of the 25th International Symposium on Computer Architecture, June, 1998.