UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

Aplicação de Algoritmos de Busca na Otimização de Sistemas Digitais

Xenia Kely Amorim
Autora

Prof. Dr. Luiz Cláudio Villar dos Santos
Orientador

Banca Examinadora:

Olinto José Varela Furtado

Luís Fernando Friedrich

Florianópolis, Fevereiro de 2004.

Agradeço primeiramente a Deus e à minha família pela por todo o amor e carinho. À equipe do projeto OASIS (Modelagem, Síntese e **O**timização de **A**rquiteturas de **Si**stemas Digitais) por compartilharem seu conhecimento, ajudando assim no desenvolvimento deste trabalho. Ao professor Luiz Cláudio Villar dos Santos pela orientação e pela excelente oportunidade de aprimoramento da minha formação acadêmica.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

1 Introdução

1.1 Contexto

Devido à necessidade da indústria de acompanhar o desenvolvimento tecnológico e, assim, permanecer em condições de competir no mercado, ferramentas automatizadas que implementam técnicas de CAD (*Computer-Aided Design*) foram adotadas. Estes programas auxiliam os projetistas no desenvolvimento de projetos de circuitos eletrônicos, no sentido de reduzir os custos e o tempo dispensado no desenvolvimento, bem como fornecer mais recursos. Por solucionarem problemas complexos e evitarem trabalhos repetitivos, as ferramentas de EDA (*Electronic Design Automation* – como são também conhecidas) diminuem o número de erros humanos no projeto dos circuitos [DEMI94].

O projeto de um circuito exige várias etapas de desenvolvimento, que podem ser reunidas em três grupos: *síntese*, *validação* e *teste*. Na primeira etapa, busca-se encontrar um circuito otimizado que possa ser fabricado. Em seguida, a validação é realizada para se que viabilize sua fabricação. O teste de algumas unidades é executado antes que se possa começar a produção em grande escala.

Conforme mostrado na figura abaixo, extraída de [AZAM02], a síntese subdivide-se em três etapas: *Síntese de Alto Nível, Síntese Lógica* e *Síntese Física*.

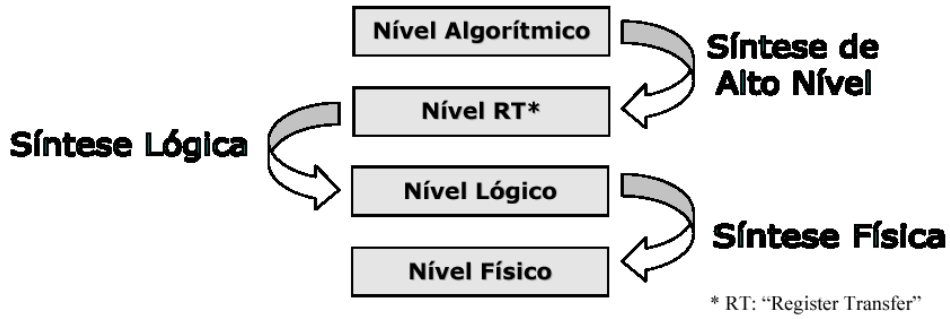

Figura 1.1: Níveis de Abstração do Projeto de Sistemas

A *Síntese de Alto Nível* é o processo de obtenção automática da estrutura arquitetural do circuito (registradores, multiplicadores, somadores, etc.) a partir de uma especificação algorítmica de seu comportamento [CAMP91]. A estrutura e o comportamento do circuito são comumente descritos com a utilização de uma linguagem de descrição de hardware (HDL – *Hardware Description Language*).

Na *Síntese Lógica* há a transformação de especificações lógicas num nível RT (Register-transfer), tais como unidades funcionais (somadores, multiplicadores, etc), elementos de memória (registradores, bancos de memória, etc) e de interconexão (barramentos, etc) para criar modelos mais básicos (portas lógicas e flip-flops).

Finalmente, na *Síntese Física* é gerado o layout geométrico, sendo que suas principais tarefas são: o *posicionamento*, que procura minimizar o tamanho do chip, e o *roteamento*, cujo objetivo é determinar as interconexões no circuito integrado.

O problema do escalonamento constitui-se de uma etapa da *Síntese de Alto Nível*, entre outras como a *seleção de módulos* (que identifica os tipos de operadores a serem utilizados), *alocação* (que define a quantidade de recursos de cada espécie) e, por fim, a *ligação* (que determina em que módulo cada operação será executada).

A *Síntese de Alto Nível* começa com a *descrição comportamental* do sistema, formada pelo conjunto de suas atribuição, estruturas condicionais e laços de repetição. Com as características específicas de uma HDL, é possível representar informações estruturais, temporizações e restrições de projeto.

O comportamento do sistema, representado no nível algorítmico, é formado pelo conjunto de *operações* e *dependências* do sistema. As dependências caracterizam as *restrições de precedência*, e são modeladas através de um *Grafo de Fluxo de Dados* ou DFG (*Data Flow Graph*). Nele, os vértices representam as operações e as arestas, as dependências.

Os resultados da Síntese de Alto Nível, ou seja, a unidade operativa (*Datapath*) e a unidade de controle (*control unit*), podem ser representadas por outros dois tipos de grafos: o DPG (*Datapath Graph*) e o SMG (*State Machine Graph*). O DPG, circuito

descrito no nível RT (*Register Transfer*), contém os tipos e as quantidades dos operadores. O SMG mostra, através de uma máquina de estados finitos, os estados necessários para se completar a execução de todas as operações.

A figura 1.2 mostra a arquitetura de um circuito digital como resultado da Síntese de Alto Nível [AZAM02]:

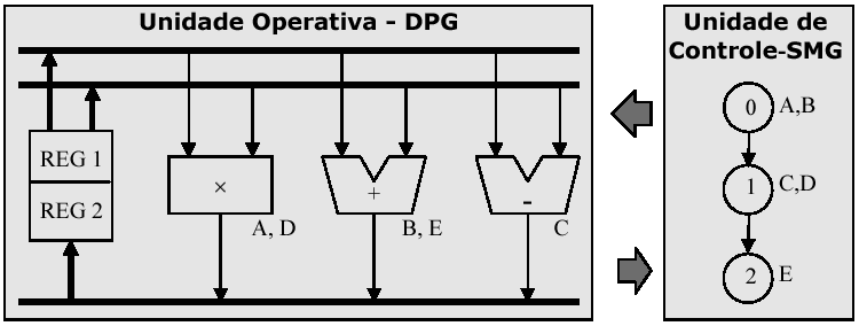

Figura 1.2 – Resultado da Síntese de Alto Nível

O presente trabalho foi desenvolvido no âmbito do projeto DESERT (Desenvolvimento de Ferramentas para Sistemas Embutidos sob Restrições de Tempo Real), que faz parte do grupo de trabalho OASIS (Modelagem, **O**timização e Síntese de **A**rquiteturas de **SIS**temas Digitais). O paradigma adotado no projeto baseia-se essencialmente nos três grafos citados (DFG, DPG e SMG) e permite a implementação de técnicas desenvolvidas para compiladores avançados.

1.2 Escalonamento

O escalonamento com restrição de recursos é um problema de otimização combinatória bem conhecido [DEMI94], pertencente à classe dos problemas NP-completos (*Non-deterministic Polynomial*).

Informalmente, o problema consiste em se encontrar uma ordenação das operações, que obedeça tanto restrições de recurso (somente uma operação pode ocupar um recurso em um dado instante) como restrições de precedência (produção e consumo de dados), de forma a minimizar o tempo total de execução.

O número de restrições (de qualquer tipo) é definido segundo requisitos de projeto do circuito em questão. Por exemplo, um circuito com um tamanho pré-definido deve ter, pelo menos, uma unidade ponto-flutuante para multiplicação/divisão [DEMI94].

1.3 Contribuições do Trabalho

A contribuição deste trabalho está em propor uma alternativa para melhorar a obtenção de diferentes codificações de prioridade, com o uso do algoritmo meta-heurístico Simulated Annealing. Sua utilização proporcionará a pesquisa de novas codificações, que por sua vez permitirão uma melhor convergência em direção à solução ótima.

1.4 Organização do Texto

O restante do trabalho está organizado como segue: o capítulo 2 apresenta a definição formal dos conceitos utilizados na abordagem do problema. O capítulo 3 descreve a interação dos blocos construtor e explorador e apresenta o algoritmo Simulated Annealing como alternativa para a geração de prioridades. No capítulo 4 os resultados são apreciados e no capítulo 5, as conclusões e as perspectivas de trabalhos futuros são apresentados.

2 Modelagem

Este capítulo apresenta as definições das entidades matemáticas e parâmetros, bem como os conceitos comumente utilizados na definição do problema computacional.

2.1 Definições Elementares

Nesta seção serão definidas formalmente as entidades matemáticas que são utilizados na abordagem do problema de escalonamento de recursos. Conforme mencionado no Capítulo 1, o produto resultante do escalonamento pode ser representado na forma de grafos.

Definição 2.1 – Um *grafo polar de fluxo de dados* DFG (V, E) é um grafo orientado, no qual cada vértice $v_i \in V$ corresponde a uma operação e cada aresta $(v_i, v_j) \in E$ representa uma dependência de dados entre as operações dos vértices v_i e v_j. Os vértices v_o e v_n dos pólos do grafo são denominados *fonte* e *sumidouro*, respectivamente.

Definição 2.2 – Um *grafo polar de máquina de estados* SMG (S, T) é um grafo orientado, no qual cada vértice $s_i \in S$ corresponde a um estado e cada aresta $(t_i, t_j) \in T$ representa a transição entre os estados s_i e s_j. Os vértices s_o e s_n dos pólos do grafo são denominados *fonte* e *sumidouro*, respectivamente.

Definição 2.3 – Um *grafo polar da unidade operativa* DPG (C, W) é um grafo orientado, no qual cada vértice $c_i \in C$ corresponde a um componente e cada aresta $(w_i, w_j) \in W$ representa um interconexão entre os componentes c_i e c_j. Os vértices c_o e c_n dos pólos do grafo são denominados *fonte* e *sumidouro*.

Em seguida, serão formalizados alguns conceitos que definem a restrição de recursos. A uma operação está relacionado um tipo, sendo possível a adição, subtração, multiplicação, etc. Estas operações são executadas em unidades funcionais, conhecidas como recursos físicos. Da mesma forma, a estes recursos estão associados tipos como somador, multiplicador, etc.

Definição 2.4 – Um vetor de *restrição de recursos* **v** é um vetor no qual cada componente indica a quantidade de recursos do tipo **k** disponíveis, sendo que $k \in \{1, 2, 3, 4, ..., n\}$.

Definição 2.5 – Uma função que mapeia o conjunto recursos para um determinado tipo de recurso **k** é definida $\tau : V \rightarrow \{1, 2, ..., n\}$, sendo $k \in \{1, 2, ..., n\}$.

Depois de algum tempo, uma operação consome seus operandos e produz um novo resultado. A figura 2.1, em referência à [AZAM02], permite identificar dois ciclos de relógio, representados pelas linhas pontilhadas, nos quais são executadas as operações A, B e C. O atraso de execução de A e B é de um ciclo de relógio, enquanto C requer dois ciclos para ser executada.

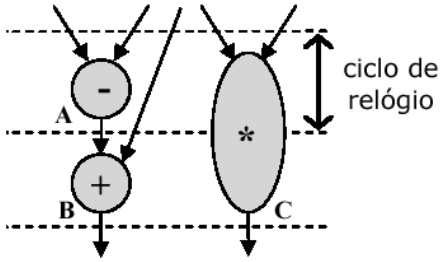

Figura 2.1 – Atraso de execução

Definição 2.6 – O *atraso de execução* d_i indica o número de ciclos de relógio que a operação v_i levou para ser executada, no recurso físico $\tau(v_i)$.

A partir das definições anteriores já é possível definir formalmente o escalonamento.

Definição 2.7 – A função $\varphi : V \rightarrow S$ que define o *escalonamento* mapeia cada vértice v_i do DFG para um estado $s_k = \varphi(v_i)$, tal que:

- $\forall(v_i, v_j) \in E$: $(s_k = \varphi(v_i)$ e $s_r = \varphi(v_j)) \Rightarrow r \geq k + d_i$ (restrição de precedência); e
- $|\{v_i: \tau(v_i)\} = p$ e $k \leq m < k + d_i| \leq a_p$, para cada tipo de recurso $p = 1, 2, ..., n$ e estado s_m, sendo $m = 1, 2, ..., n$ (restrição de recursos).

Se visto como uma seqüência de operações, a performance do modelo da arquitetura de um circuito pode ser definida através da *latência*, isto é, do tempo necessário para executar suas operações, medido em ciclos de relógio.

Definição 2.8 – O início de execução (*start time*) para as operações é denotado por $T = \{t_i; i = 0, 1, ..., n\}$, ou seja, o ciclo no qual a operação começa a ser executada.

Definição 2.9 – A latência λ do escalonamento é definida pelo número de ciclos de relógio necessários para executá-lo, ou seja, a diferença entre o ciclo em que os vértices fonte e sumidouro foram executados: $\lambda = t_n - t_0$.

Quando o escalonamento está sendo executado, as operações escalonadas vão sendo associadas a estados do SMG. As operações que já podem ser escalonadas em um dado estado são aquelas que possuem todos os seus predecessores já escalonados e que ainda não tenham sido escalonadas, isto é, aquelas que obedecem a restrições de precedência.

Definição 2.10 – Dado um SMG S e um estado arbitrário $s_k \in S$, o conjunto de operações v_j disponíveis para o escalonamento no estado s_k é chamado A_k.

No entanto, nem todas as operações de A_k podem ser executadas no estado s_k simultaneamente, devido às restrições de recursos. Desta forma, um subconjunto de A_k deve ser selecionado. A atribuição de um estado s_k do SMG depende da ordem na qual as operações do conjunto A_k são selecionadas. Como conseqüência, uma solução potencial de um problema pode ser induzida atribuindo-se uma *prioridade* para cada operação. Por esta razão, a interação entre os blocos construtor e explorador está em produzir diferentes soluções na forma de codificações de prioridade diferentes [SANT98].

Definição 2.11 – Dado um DFG=(V, E), uma codificação de prioridade é essencialmente uma permutação Π de operações de V. A noção de prioridade está associada com a posição relativa das operações em Π. Desta forma, a operação v_j tem maior prioridade do que $v_{i,}$ vi $<_\Pi v_j$, se $\Pi(v_i) < \Pi(v_j)$.

A geração de soluções idênticas durante a exploração de SMGs alternativos pode ser evitada usando-se a condensação ε da codificação de prioridade Π, como sugerido em [SANT98]: como um mecanismo de seleção baseado em uma lista de escalonamento é usado no Bloco Construtor, se duas codificações de prioridade possuírem a mesma condensação, as soluções por elas induzidas serão idênticas.

Como exemplo de sua utilização, assume-se que antes de enviar uma codificação de prioridade Π_j ao Construtor, o bloco Explorador verifica se existe algum ε_i na tabela, tal que $\varepsilon_j = \varepsilon_i$. Em caso afirmativo, outra codificação Π_{j+1} é gerada e o processo descrito repete-se. Caso a igualdade não se verifique, a codificação Π_j é enviada ao Construtor.

3 Descrição da Abordagem

3.1 Construção e Exploração de Soluções

A idéia-chave da exploração de soluções é a seguinte: diferentes codificações de prioridade resultam em soluções diferentes, com custos possivelmente distintos. Maiores detalhes sobre a abordagem utilizada podem ser encontrados em [AZAM02] e [SANT98].

A abordagem utilizada para resolver os problemas de otimização combinatória pode ser vista como a interação entre dois grandes blocos, conhecidos como *explorador* e *construtor* [AZAM02].

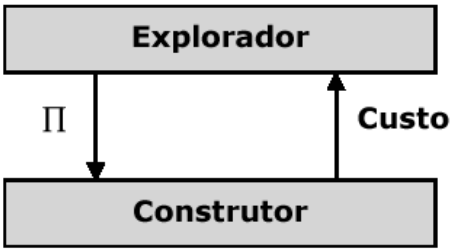

Figura 3.1 – Visão Geral da Interação dos Blocos

Uma das funções do *explorador* é definir a codificação de prioridade das operações a serem escalonadas. Esta codificação é definida por uma permutação Π das operações de um grafo DFG. Outra importante função deste segmento da abordagem é comparar o custo da última solução encontrada com os resultados encontrados anteriormente, e decidir se a solução atual, fornecida pelo construtor, é satisfatória ou não.

É no explorador que decisões são tomadas, com base no custo calculado pelo construtor. A função do construtor é produzir uma solução e avaliar seu custo, baseado somente na codificação de prioridade Π gerada pelo explorador e obedecendo a restrições de recurso e precedência. Na abordagem atual, a criação das prioridades é gerada pelo algoritmo *Random Search* (RS), que atribui, aleatoriamente, prioridade às operações [AZAM02].

3.2 Aplicação do Algoritmo de Simulação de Têmpera

O Simulated Annealing (SA) é um algoritmo meta-heurístico, ou seja, um método heurístico genérico, que pode ser aplicado em qualquer problema de otimização. Trata-se de um algoritmo poderoso que não exige que um problema apresente uma estrutura especial para sua utilização [CHAN94].

3.2.1 Estrutura do Algoritmo de Simulação de Têmpera

O algoritmo Simulated Annealing é uma generalização do método de Monte Carlo, que examina equações de estado de sistemas termodinâmicos [METR53]. O esquema original do método consistia em escolher um estado inicial para o sistema em questão, com uma certa energia E e temperatura T, perturbar T e calcular a mudança de energia dE.

Assim, o algoritmo objeto deste estudo é semelhante ao processo de cristalização dos metais. Neste processo, inicialmente, aquece-se o metal até uma alta temperatura, que vai sendo reduzida paulatinamente. Enquanto estiver quente, a alta temperatura permite diversas configurações das moléculas dispersas no seu meio, não obedecendo a nenhuma regra de organização.

Durante o resfriamento, as configurações tentarão se concentrar naquelas que possuírem menor valor energético, visto que baixas temperaturas dificultam a movimentação das moléculas [NAJI02].

A figura 3.2, extraída de [HENT02], torna possível a visualização do processo.

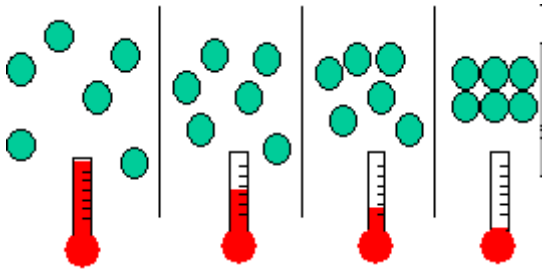

Figura 3.2 – Processo simulado pelo algoritmo Simulated Annealing

A analogia do método de Monte Carlo para problemas combinatórios foi feita primeiramente por [KIRK83]. O estado atual do sistema termodinâmico é análogo à solução atual do problema, sendo que para os problemas de otimização as seguintes analogias também podem ser observadas [MAZI03]:

- A equação de energia do sistema assemelha-se à função de custo;
- Os estados físicos do sistema são equivalentes às soluções do problema;
- A perturbação de um estado do sistema é equivalente a seleção de uma solução vizinha do problema de otimização;
- O resfriamento rápido de um sistema é equivalente a um ótimo local do problema;
- O estado ideal de posicionamento das partículas é equivalente ao ótimo global do um problema combinatório.

Na prática, são necessários três ingredientes básicos para aplicar o SA: uma representação concisa do problema em questão, uma função de vizinhança e uma função de resfriamento. Para o resfriamento existem algumas funções gerais que podem ser utilizadas. No entanto, não são conhecidas regras gerais que guiem a escolha dos outros ingredientes [NAJI02]. A forma como ela são implementadas depende da experiência e da habilidade de utilização do algoritmo.

Uma das dificuldades na implementação do algoritmo é que não existe uma analogia óbvia para a temperatura. Além disso, a enorme quantidade de parâmetros necessários, que controlam a velocidade e a qualidade da busca, requer especial atenção e uma considerável experimentação [BROW92].

Já a grande vantagem frente a outros métodos é a habilidade do algoritmo de evitar ficar preso em um mínimo local, pois emprega uma busca randômica, que não aceita somente as mudanças que diminuem o valor da função de custo.

A variável temperatura controla o "grau de agitação" do algoritmo, sendo que a temperatura está relacionada com a tolerância do algoritmo a perturbações. Quanto mais

alta for a temperatura, maior é a aceitação do algoritmo a modificações, mesmo se as mudanças não forem tão boas. Conforme a temperatura do algoritmo for diminuindo, apenas soluções melhores do que a solução atual serão aceitas. A função de custo é responsável por julgar a solução.

A solução inicial é, geralmente, aleatória. A cada iteração, é calculada uma pequena perturbação no estado atual. No algoritmo SA admite-se uma solução pior que a anterior, segundo uma probabilidade que varia ao longo do algoritmo. Esta probabilidade é calculada em função da temperatura.

3.2.2 Adaptação do Algoritmo Simulated Annealing para o Problema De Escalonamento

O algoritmo Simulated Annealing trabalha com uma solução temporária, que é alterada por uma função de perturbação. A aceitação das soluções geradas a partir das perturbações é controlada pela variável temperatura. Ela é inicializada com um valor geralmente alto, com o objetivo de evitar mínimos locais. Para cada valor de temperatura são realizados uns números de iterações, para que movimentos na vizinhança da solução sejam executados.

A probabilidade de uma solução ser aceita é representada pela expressão $e^{(-\Delta/TkB)}$, onde Δ é a variação no custo da solução atual e da nova solução, T a temperatura corrente e k_B, a constante de Boltzmann[1].

Caso Δ seja menor que zero, a solução perturbada é aceita como a nova solução corrente. Do contrário, ela só será aceita se $e^{(-\Delta/TkB)} > random(0,1)$. Números randômicos uniformemente distribuídos entre 0 e 1 são, segundo [KIRK83], uma forma conveniente de implementar a porção randômica do algoritmo.

O algoritmo 3.1, em referência a [HENT02], e o fluxograma da figura 3.3, mostram o algoritmo Simulated Annealing.

[1] A constante de Boltzmann é utilizada em processos físicos

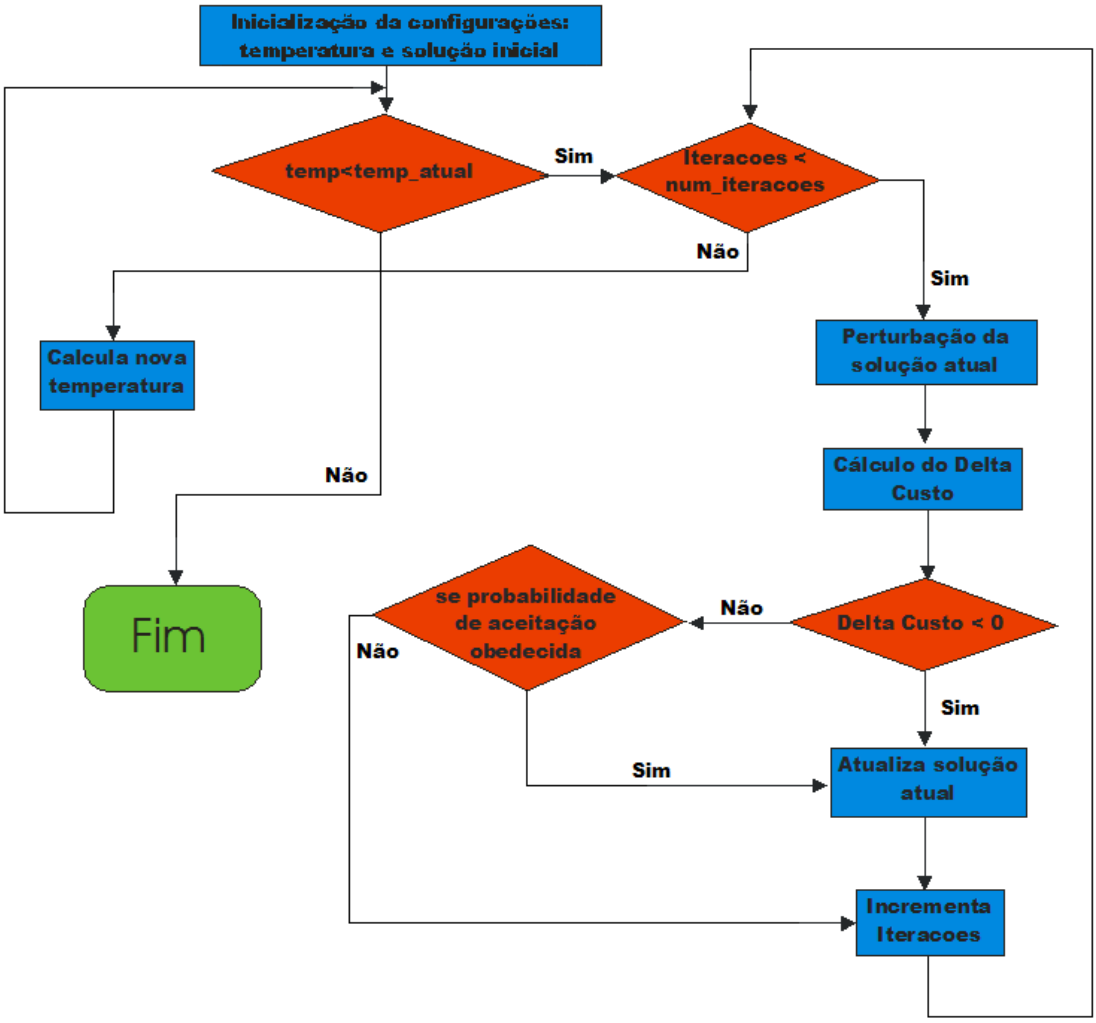

Figura 3.3 – Fluxograma do Algoritmo Simulated Annealing

```
1   Temperatura = temperatura_inicial;
2   Solução = Solução Aleatória
3   Enquanto Temperatura>temperatura_final
4       Para cada iteração na mesma temperatura
5               NovaSolução = Perturbação da Solução;
6               ΔCusto = Custo(NovaSolução) - Custo(Solução);
7               Se (ΔCusto < 0)
8                  Solução = NovaSolução;
9               Senão Se AceitaSolução(NovaSolução)
10                 Solução = NovaSolução;
11              Fim Senão;
12      Fim Para;
12       Temperatura = Scheduling(Temperatura);
13   Fim Enquanto;
```

Algoritmo 3.1 – Algoritmo Simulated Annealing

3.2.2.1 Função de *Scheduling*

A Função de *Scheduling* é responsável por calcular o valor da temperatura no decorrer do algoritmo. É baseando-se na temperatura que se decide se uma solução é aceita ou não, ou seja, a temperatura define a aceitação do algoritmo. Como mencionado anteriormente (3.2.1), o SA permite também soluções ruins, o que lhe permite evitar os mínimos locais.

A função para cálculo da temperatura escolhida é a adotada em [KIRK83]. O resfriamento é exponencial, sendo calculado pela expressão $T_n = (T_1/T_0)^n T_0$, tendo a razão T_1/T_0 valor 0,9. Esta variação deverá ser suave, a fim de possibilitar a aceitação de soluções de escalonamento ruins, com altas temperaturas, e somente as boas com o "resfriamento". O valor da temperatura inicial é geralmente ajustado experimentalmente [NAJI02]. A menor temperatura aceita, que serve como uns dos critérios de parada para o algoritmo SA, é de 0.1 [JACK03].

3.2.2.2 Função de Perturbação

A função de perturbação deve possuir algumas propriedades, sendo elas [HENT02]:

- Simplicidade: a perturbação deve ser pequena para que, deste modo, qualquer permutação possa ser conseguida;
- Deve ser ainda consistente, de forma a gerar permutações consistentes.

No presente trabalho, será utilizado um processo denominado *randomSwitch*. Esta função consiste em se escolher randomicamente duas operações da codificação de prioridade Π. No entanto, a troca de posições só ocorrerá se duas condições forem satisfeitas. São elas:

1. As operações escolhidas devem mapear um mesmo tipo de recurso [AZAM02];

2. A troca de posições deve levar em conta o fato de que, se B depende do resultado da operação A, então não faz sentido atribuir maior prioridade à operação dependente de dados.

Apesar de normalmente ser aleatória, é aconselhável que a solução inicial, na qual será primeiramente aplicada a função de perturbação, seja escolhida baseada em alguma regra ou heurística de baixa complexidade [NAJI02]. Sua escolha influenciará na vizinhança utilizada no sentido de melhorar a qualidade da permutação Π.

Neste trabalho, optou-se por ordenar a codificação Π segundo o *grau de emissão*[2] de cada operação. Isto baseado no fato de que quanto maior for o número de arestas que partem de um vértice, maior será o número de operações disponíveis após o seu escalonamento.

3.2.2.3 Função de Custo

O custo do escalonamento é calculado no bloco *Construtor* [AZAM02]. Ele é encarregado de criar uma solução para cada permutação Π, através do escalonamento do DFG. O cálculo do custo é baseado na latência λ do escalonamento.

3.2.2.4 Função de Aceitação de uma Solução

A probabilidade de aceitação de uma solução é dada pela equação $exp(-\Delta Custo / T)$. A semelhança com o método proposto por [KIRK83] é grande: $\Delta Custo$ corresponde à variação de energia de um estado para outro de um sistema e o parâmetro de controle T à temperatura. Uma vez que a variável T é imaginária, a constante de Boltzmann K, que aparecia na equação original, pode ser considerada igual a 1 [MAZI03], visto que as características físicas de temperatura de um sistema computacional não precisam ser consideradas!

O algoritmo 3.2 mostra uma adaptação feita para o problema de escalonamento do algoritmo Simulação de Têmpera:

[2] Grau de emissão de um vértice *v* corresponde ao número de arestas que partem de *v*

```
1       temperature = initialTemperature;
2       Πinitial = initialSolution();
3       Cost = Construtor(Π);
4
5       Enquanto temperature > finalTemperature
6       Para cada iteração na mesma temperatura
7             Π' = perturbation(Π);
8             newCost = Constructor(Π');
9             ΔCost = newCost – Cost;
10            Se (ΔCost < 0)
11                  Π = Π';
12            Senão Se acceptSolution(ΔCost, temperature)
13                  Π = Π';
14            Fim Se.
15      Fim Para.
16      Fim Enquanto.
17
18      temperature = scheduling(temperature);
```

Algoritmo 3.2 – Adaptação do Algoritmo Simulação de Têmpera para o problema de Escalonamento

4 Implementação e Resultados Experimentais

Neste capítulo serão descritos os experimentos realizados, os resultados obtidos com a utilização do algoritmo SA, bem como os algoritmos implementados. A plataforma utilizada para o desenvolvimento do trabalho foi um PC Pentium 1.8 GHz, 256 Mb de memória RAM, com o sistema operacional Linux e a linguagem C++ [DEIT94], ambiente de trabalho KDE [KDE03] e ambiente de desenvolvimento KDevelop 2.1 [KDEV03], que é parte integrante do KDE. O compilador utilizado foi o gcc (GNU C Compiler).

4.1 Implementação do Algoritmo Simulated Annealing

As classes utilizadas foram as desenvolvidas e as utilizadas por [KLEI02] e [AZAM02]. À classe `solution.cpp` foi agregado o Algoritmo de Simulação de Têmpera, em substituição ao gerador aleatório de permutações. Os algoritmos a seguir apresentam com mais detalhes a implementação do algoritmo 3.1.

O algoritmo da função que determina o $\Pi_{inicial}$ é mostrado a seguir:

```
initialSolution()
      inicializacao do Π;
      orderNodesByOutDegree(Π);
fim.
```

Algoritmo 4.1: Geração da codificação de prioridades inicial

O procedimento `orderNodesByOutDegree` ordena as operações segundo o número de arestas que partem dos vértices que as representam. Quanto maior for o grau de emissão de um nodo, maior será a prioridade a ele atribuída.

O algoritmo da função de perturbação (Seção 3.2.2.2) é apresentado a seguir:

```
perturbation(∏)
      faca
            randomNumbers(R1, R2);

      enquanto((k∏[R1] ≠ k∏[R2]) E (∏[R2] for sucessor de ∏[R1]))
      fim.

      temp = ∏[R1]
      ∏[R1] = ∏ [R2];
      ∏[R2] = temp;

fim.
```

Algoritmo 4.2 – Algoritmo da Função de Perturbação

A função `randomNumbers` retorna em R1 e R2 números aleatórios, de forma que em R1 esteja sempre o número que representa a maior prioridade. Ou seja, dado um ∏ = (A, B, C, D, E, F, G), a R1 será atribuído um número menor do que à R2. Isto serve para garantir a segunda condição imposta em 3.2.2.2.

A seguir, o algoritmo da função de aceitação do SA:

```
acceptSolution(ΔCost, temperature)
      R = número randômico entre [0,1]
      prob = exp( ((-1) * ΔCost)/(temperature) );

      se (R < prob)
        solução aceita;
      senão
        solução rejeitada;
      fim senão;
fim.
```

Algoritmo 4.3 – Algoritmo da Função de Aceitação

Esta função apresenta a porção randômica do SA: um número aleatório entre 0 e 1 é escolhido e então utilizado para determinar a aceitação ou não da solução atual. A probabilidade está definida em 3.2.2.4.

O algoritmo responsável por controlar a temperatura é mostrado a seguir (vide seção 3.2.2.1):

```
scheduling(initialTemperature, currentIteration)
    ratio = 0.9;
    newTemperature = pow(ratio, currentIteration) * initialTemperature;

    return newTemperature;
fim.
```

Algoritmo 4.4 – Algoritmo da Função de Scheduling

4.2 Resultados Experimentais

Para a realização dos experimentos foram utilizados *benchmarks* clássicos da literatura de Síntese de Alto Nível: *diffeq*, *wdelf* e *fdct*.

O exemplo *diffeq* é baseado em um algoritmo para resolução de equações diferenciais, encontrado em [DEMI94]. O *wdelf* é um algoritmo que implementa um filtro de onda digital de quinta ordem, extraído de [DEWI85]. E finalmente, o exemplo *fdct* é um algoritmo que executa a compactação de imagens através da transformação discreta de co-senos (*fast discrete cosine transform*), encontrado em [MALL90]. A tabela 4.1 apresenta as principais características dos DFGs dos exemplos, no que se relaciona ao número de vértices, arestas e tipo das operações.

Tabela 4.1: Características dos benchmarks utilizados

Exemplo	Vértices	Arestas	Operações
diffeq	13	16	2 adições, 2 sub., 1 comp. e 6 mult.
fdct	44	68	13 adições, 13 sub. e 16 mult..
wdelf	36	66	25 adições, 1 sub. e 8 mult.

Ao número de vértices estão somados os vértices polares (fonte e sumidouro) e às arestas foram somadas as arestas emergentes do vértice fonte e as incidentes no sumidouro. O atraso de execução destas arestas é considerado nulo, não interferindo assim no escalonamento.

Para os experimentos realizados, considerou-se que os multiplicadores necessitam de dois ciclos de relógio para executarem uma operação e a Unidade Lógico-Aritmética (adições, subtrações e comparações), um ciclo de relógio.

As tabelas a seguir apresentam o custo médio das soluções obtidas para os três exemplos utilizados como *benchmarks*, em dois cenários distintos: com a abordagem randômica (Random Search - RS) e com o SA.

Tabela 4.2: Custo médio das soluções para o exemplo *fdct*

MULTs	ALUs	$\lambda_{ótima}$	Custo Médio$_{RS}$	Custo Médio$_{SA}$
8	4	8	9,7	8,8
5	4	10	11,5	10,6
4	3	11	13,2	12,4
4	2	13	15	14,3
3	2	14	16,5	15,8
2	2	18	20,7	19,9
2	1	26	27	26,5
1	1	34	37,5	36,1

Tabela 4.3: Custo médio das soluções para o exemplo *wdelf*

MULTs	ALUs	$\lambda_{ótima}$	Custo Médio$_{RS}$	Custo Médio$_{SA}$
3	3	17	17,9	17,6
2	2	18	19,2	18,8
1	2	21	21,8	21,4
1	1	28	29	28,4

Tabela 4.4: Custo médio das soluções para o exemplo *diffeq*

MULTs	ALUs	$\lambda_{ótima}$	Custo Médio$_{RS}$	Custo Médio$_{SA}$
4	1	6	6	6
2	2	7	8,54	8,02
3	2	6	7,3	6,7
3	1	7	7,5	7,3
2	1	8	8,73	8,4
1	1	13	13,5	13,3

Note que o uso do SA garante um custo médio inferior ou igual ao do RS, para as diferentes restrições de latência, em todos os casos testados.

O impacto da utilização do SA no tempo de resposta pode ser observado nas tabelas a seguir. Elas apresentam o tempo necessário para que os algoritmos RS e SA atinjam a $\lambda_{ótima}$.

Tabela 4.5: Tempo para atingir $\lambda_{ótima}$ para o exemplo *fdct*

	MULTs	ALUs	$\lambda_{ótima}$	Random Search	Simulated Annealing
A	8	4	8	0,025	0,004
B	5	4	10	0,03	0,021
C	4	3	11	0,98	0,65
D	4	2	13	0,82	0,70
E	3	2	14	83,7	6,85
F	2	2	18	13,23	5,94
G	2	1	26	0,016	0,032
H	1	1	34	1,69	3,42

Tabela 4.6: Tempo para atingir $\lambda_{ótima}$ para o exemplo w*delf*

	MULTs	ALUs	$\lambda_{ótima}$	Random Search	Simulated Annealing
A	3	3	17	0,076	0,087
B	2	2	18	0,14	0,11
C	1	2	21	0,02	0,001
D	1	1	28	0,0131	0,01305

Tabela 4.7: Tempo para atingir $\lambda_{ótima}$ para o exemplo *diffeq*

	MULTs	ALUs	$\lambda_{ótima}$	Random Search	Simulated Annealing
A	4	1	6	0,00156	0,0005
B	2	2	7	0,026	0,021
C	3	2	6	0,006	0,005
D	3	1	7	0,003	0,002
E	2	1	8	0,003	0,0019
F	1	1	13	0,0037	0,004

Da análise das tabelas, verifica-se que o SA é pior apenas nos casos *fdct G*, *fdct H*, *wdelf A* e *diffeq F*. Isto pode ser explicado da seguinte forma: nestes casos, exceto no *wdelf A*, que será explicado posteriormente, o número de recursos é bastante escasso, logo a latência é menos sensível às permutações, devido à carência de paralelismo.

Os resultados das tabelas 4.5, 4.6 e 4.7 podem ser graficamente visualizados nas figuras a seguir:

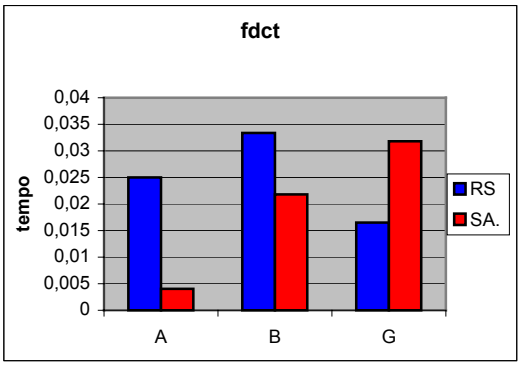

Figura 4.1a: Gráfico da tabela 4.5 (*fdct A,B e G*)

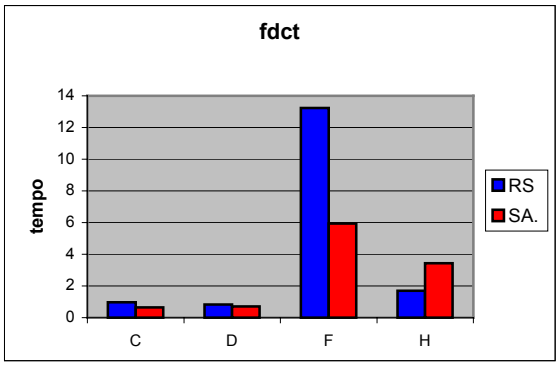

Figura 4.1b: Gráfico da tabela 4.5 (*fdct C,D, F e H*)

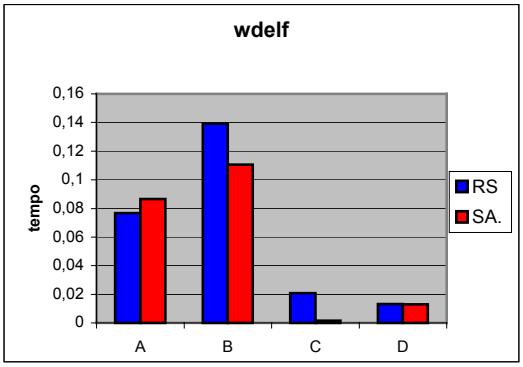

Figura 4.2: Gráfico da tabela 4.6 (*wdelf*)

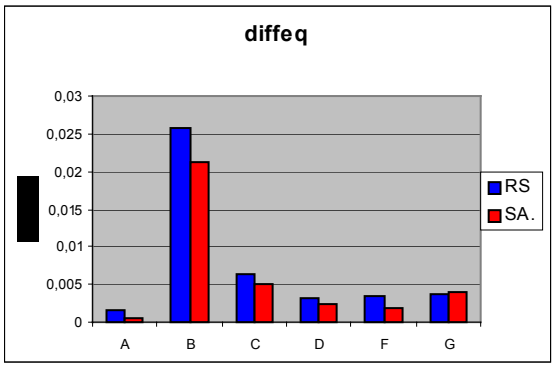

Figura 4.3: Gráfico da tabela 4.7 (*diffeq*)

As tabelas 4.8, 4.9 e 4.10 ilustram o número médio de codificações de prioridade pesquisadas até que a latência ótima seja alcançada, nos exemplos *fdct*, *wdelf* e *diffeq*.

Tabela 4.8: Número médio de soluções para alcançar a $\lambda_{ótima}$ para o exemplo *fdct*

	MULTs	ALUs	$\lambda_{ótima}$	Random Search	Simulated Annealing
A	8	4	8	4,97	2,87
B	5	4	10	6,34	14,5
C	4	3	11	181,3	406,9
D	4	2	13	149,7	413,3
E	3	2	14	10056,2	888,5
F	2	2	18	2241,8	2171,7
G	2	1	26	2,65	12,4
H	1	1	34	238,4	1063,1

Tabela 4.9: Número médio de soluções para alcançar a $\lambda_{ótima}$ para o exemplo *wdelf*

	MULTs	ALUs	$\lambda_{ótima}$	Random Search	Simulated Annealing
A	3	3	17	16,17167	61,27
B	2	2	18	13,92	75,655
C	1	2	21	4,196	1
D	1	1	28	2,375	6,395

Tabela 4.10: Número médio de soluções para alcançar a $\lambda_{ótima}$ para o exemplo *diffeq*

	MULTs	ALUs	$\lambda_{ótima}$	Random Search	Simulated Annealing
A	4	1	6	1	1
B	2	2	7	15	34
C	3	2	6	3,9	8,9
D	3	1	7	1,9	3,8
E	2	1	8	1,9	3,2
F	1	1	13	1,9	4,4

O fato do caso *wdelf A* levar mais tempo, utilizando o SA, para encontrar a solução ótima pode ser explicado observando-se que o Simulated Annealing pesquisa mais soluções (vide tabela 4.9, caso A) e que a diferença entre o custo médio das suas soluções e das soluções do RS, para restrição de latência 17 (vide tabela 4.3), é pequena.

Como o custo para construir soluções é parecido e o SA pesquisa mais soluções, é plausível o fato de o SA necessitar de mais tempo para encontrar uma solução ótima.

Embora o SA necessite de mais soluções, ele resulta em menor tempo de busca. Isto é aparentemente contraditório, entretanto, a explicação é que o SA constrói soluções de menor custo médio (vide tabelas 4.2, 4.3 e 4.4), o que resulta em menor tempo médio para construir uma solução.

5 Conclusões e Perspectivas

Este trabalho teve como objeto de estudos a geração de codificações de prioridade pelo bloco explorador. A principal contribuição foi a extensão deste bloco, conforme sugerido em [AZAM02]: nele foi embutido o algoritmo evolucionário Simulated Annealing, a fim de obter-se uma melhor convergência para as soluções ótimas dos problemas.

Os resultados apresentados indicam que o SA melhorou a busca de codificações de prioridade grande parte dos casos testados, diminuindo o custo médio das soluções, e por conseqüência, diminuindo o tempo médio para construir uma solução.

6 Referências

[AZAM02] AZAMBUJA, Rogério Xavier de. **Escalonamento e Alocação de Registradores sob Execução Condicional**: Universidade Federal de Santa Catarina, Dissertação de Mestrado, Florianópolis, 2002.

[BROW92] BROWN, Donald E et. al. **Rail Network and Scheduling Using Simulated Annealing**. The Institute for Parallel Computation, University of Virginia. IEEE, 1992.

[CAMP91] CAMPOSANO, R. **Path-based Scheduling for Syntesis**: IEEE Trans. On Computer-Aided Desing, Jan. 1991, Vol. 10 nº 1, p. 85-93.

[CHAN94] CHANG-YUN, S.; YOH-HAN, P.; PERCY, Y. **Scheduling Multiple Job Problems with Guided Evolutionary Simulated Annealing Approach**, IEEE, 702-706, 1994.

[DEIT94] DEITEL, H. M.; DEITEL, P. J. **C: How to Program**. Prentice-Hall, New Jersey, 1994.

[DEMI94] DE MICHELI, Giovanni. **Syntesis and Optimization o Digital Circuits**, USA: Mc Graw-Hill, 1994.

[DEWI85] DEWILDE, et. al. **Parallel and Pipelined VLSI Implementation of Signal Processing Algorithms**: in S. Y. Kung, H. J. Whitehouse and T. Kailath, VLSI and Modern Signal Processing. Prentice Hall, 1985.

[HENT02] HENTSCHKE, Renato Fernandes. **Algoritmos para o Posicionamento de Células em Circuitos VLSI**: Universidade Federal do Rio Grande do Sul, Dissertação de Mestrado, Porto Alegre, 2002.

[JACK03] JACKSON, William C.; MCDOWELL, Mark E. **Simulated Annealing with Dynamic Perturbations: A Methodology for Optimization**. Space Systems Division, Los Angeles AFB, CA.

[KDE03] **KDE – The K Desktop Environment**, disponível em: www.kde.org, acesso em: 2003.

[KDEV03] **Kdevelop**, disponível em: www.kdevelop.org, acesso em: 2003.

[KIRK83] KIRKPATRICK, S., GELATT JR, C. D., VECCHI, M. P. **Optimization by Simulated Annealing**, Science, 220, 4598, 671-680, 1983

[KLEI02] KLEIN, Felipe Vieira; **Modelagem de Arquiteturas de Sistemas Digitais: Implementação e Ferramentas de Síntese Automática**. Universidade Federal de Santa Catarina, Trabalho de Conclusão de Curso, 2002.

[MALL90] MALLON, D. J.; DENYER, P. B. **A New Approcah to Pipelined Optmization**: Proc. EDAC'90m OO, 1990.

[MAZI03] MAZIERO, Edélcio Augusto. **Algoritmos Simulated Annealing em Paralelo + Genético Grossover – Uma Abordagem Híbrida**: Universidade Federal de Santa Catarina, Dissertação de Mestrado, Florianópolis, 2003.

[METR53] METROPOLIS, N. et. al. **Equation of State Calculations by Fast Computing Machines**, J. Chem. Phys, 21, 6, 1087-1092, 1953.

[NAJI02] NAJIB, M. Najid; DAUZERE-PÉRÈS, Stephane; ZAIDAT, Ali. **A Modified Simulated Annealing Method for Flexible Job Shop Scheduling Problem**. Institute de Recherche en Communications et Cybernétique de Nantes. IEEE SMC, 2002.

[SANT98] SANTOS, Luiz C. V. dos. **Exploiting instruction-level parallelism: a**

construtive approach, Eindhoven University of Technology, PhD. Thesis, 1998.

7 Anexos

7.1 Código Fonte do arquivo *solution.cpp*

```
/*************************************************************************
                          solution.cpp  -  description
                             -------------------
    begin                : Mon Jan 29 2001
    copyright            : (C) 2001 by Felipe Vieira Klein (extended by Xenia
                           Kely Amorim)
    email                : fvklein, xenia@inf.ufsc.br
 *************************************************************************/

/*************************************************************************
 *                                                                       *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                       *
 *************************************************************************/

#ifndef SOLUTION_CPP
#define SOLUTION_CPP

#include <stdlib.h>
#include <string.h>
#include <math.h> //included by xenia in 13/10/2003

#include "solution.h"
#include "dfn.h"
#include "dfg.h"
#include "list.h"
#include "sta.h"
#include "smg.h"
#include "cmptypes.h"
#include "cmp.h"
#include "debugging.h"
#include "Hash.h"
#include "HashItem.h"
```

```
SOLUTION::SOLUTION( DFG & theDFG, DPG & theDPG ): pi(
theDFG.getNumberOfNodes() - CONTROL_NODES )
{
       // set the references to the data-path and the data-flow graphs
//  computedSolutions = new HashTable( );
       dfg = &theDFG;
       dpg = &theDPG;

       smg = new SMG( NULL );                                    // creates a
new empty state-machine graph
}

SOLUTION::~SOLUTION()
{
       // here the dynamically allocated resources will be
       // destroyed some way to free the memory
       delete smg;
}

void SOLUTION::Undo()
{
    Reservation->~RESERVATION();
    smg->~SMG();
    smg = new SMG( NULL );
    dfg->restore();
}

void SOLUTION::rotateLeft(Vector< DFNPtr> & theSolution)
{
    DFNPtr firstNodePtr = pi[0];

    for (int i = 1; i < pi.Length() -1; i++)
    {
        theSolution[i-1] = pi[i];
    }
    theSolution[pi.Length() -1] = firstNodePtr;
}


void SOLUTION::randomSwitch(Vector<DFNPtr> & theSolution)
{
    int greatest, smallest;
                          // will store a randomly generated integer
    int max = pi.Length();
    theSolution = pi;
    bool found = false;
```

```
    do{
        //Look for 2 random numbers
        greatest = randomNumber( max-1 );
        smallest = randomNumber( max-1 );

        do
        {
            greatest = randomNumber( max-1 );
            do
                smallest = randomNumber( max-1 );
            while ( greatest == smallest );

         } while ( pi[ greatest ]->getType() != pi[smallest]->getType() );

        //Look for 2 nodes to change their positions in PI
          int temp;
          if ( greatest < smallest )
          {
              temp = greatest;
              greatest = smallest;
              smallest = temp;
          }

          List<DFEPtr> *outEdges = pi[smallest]->getOutgoingEdges();
          DFEPtr    currentEdge;
          DFNPtr    sucessor;

          //cout << "Maior prioridade: " << pi[smallest]->getName() << endl;

          found = false;
          for (List<DFEPtr>::iterator itr = outEdges->begin(); itr !=
outEdges->end(); ++itr )
          {
              currentEdge = *itr;
             // gets the pointer to the successor edge
              sucessor = &dfg->getNode( currentEdge->getDestiny() );
              //cout << "Sucessor 1: " << sucessor->getName() << endl;
              //cout << "Sucessor 2: " << pi[greatest]->getName() << endl;

              if (sucessor->getName() == pi[greatest]->getName())
              {
                found = true;
                break;
              }
```

```
            }
    } while ( found == true );

    DFNPtr nodePtr = pi[ greatest ];
    theSolution[greatest] = pi[smallest];
    theSolution[smallest] = nodePtr;
}


String SOLUTION::vectorToString(Vector <DFNPtr> & theVector)
{
    String vectorName = "";
    for ( int i = 0; i < theVector.Length(); i++ )
        vectorName = vectorName + theVector[i]->getName();

    return vectorName;
}


void SOLUTION::initialSolution()
{
    generatePI();
    dfg->orderNodesByOutDegree(pi);

    HashItem<String> HashV = vectorToString(pi);
    oldPis.Insert(HashV);
}

void SOLUTION::perturbation ( Vector< DFNPtr > & theSolution )
{
      String vectorString;
      int cont = 0;

      do {
        randomSwitch(theSolution);
        vectorString = vectorToString(theSolution);
        cont++;

        if (cont > 100)
        {
          Vector <DFNPtr> temp ( pi.Length() / 2 );
          for (int i = 0; i < temp.Length(); i++)
          //Saves the less significant positions of the PI
            temp[i] = pi[i];

          for (int i = 0; i < temp.Length(); i++)
```

```
          {
            pi[i] = pi[ temp.Length() +i];
            pi[ temp.Length() +i] = temp[i];
          }

          theSolution = pi;
          vectorString = vectorToString(theSolution);
          cont = 0;
        }

      } while ( oldPis.IsFound(vectorString) );

      //oldPis.Insert(vectorString);
}

double SOLUTION::scheduling(double initialTemperature, int iteration)
{
    double ratio = 0.9;
    double temp = pow(ratio, iteration) * initialTemperature;

    return temp;
}


int SOLUTION::costOfSolution(Vector < DFNPtr > & theSolution)
{
  int cost = Schedule(theSolution);
  Undo();
  return cost;
}


bool SOLUTION::acceptSolution(double deltaCost, double temperature)
{
    double R = randomDoubleNumber(0);
    double prob = exp( ((-1) * deltaCost)/(temperature) );

    if (R < prob) return true;

    return false;
}


int SOLUTION::simulatedAnnealing(int numberIter, double initialTemp, int
bestLatency, int& ns)
{
```

```
    double finalTemperature = 0.1,
           initialTemperature = initialTemp,
           temperature;
    int deltaCost, costA, costB;

    int numIterations = numberIter,
        temperatureChanges = 1;  //used to evaluate new temperature

    temperature = initialTemperature;

    initialSolution();  //pi is set by this function
    Vector< DFNPtr > newSolution ( pi.Length() );
    oldPis.MakeEmpty();

    int nSolucoes = 1;
    costA = costOfSolution(pi);
    String vectorString = vectorToString(pi);
    oldPis.Insert(vectorString);

   do
    {
        for ( int iter = 0; iter < numIterations; iter++ )
        {
            perturbation( newSolution );

            costB = costOfSolution(newSolution);
            nSolucoes++;

            deltaCost = costB - costA;

            if ( deltaCost < 0 )
            {
                pi = newSolution;
                costA = costB;
                vectorString = vectorToString(pi);
                oldPis.Insert(vectorString);
            } else if ( (acceptSolution(deltaCost, temperature)) && (deltaCost
!= 0) )
            {
                pi = newSolution;
                costA = costB;
                vectorString = vectorToString(pi);
                oldPis.Insert(vectorString);
            }
        }
        temperature = scheduling(initialTemperature, temperatureChanges);
```

```
        temperatureChanges++;

    }
    while (temperature > finalTemperature);
    ns = nSolucoes;
    return costA;
}



/** generates the pi vector ( the priority key vector)
given the data-flow graph */
void SOLUTION::generatePI()
{
      int rnd;
            // will store a randomly generated integer
      Vector< DFNPtr > & Template = dfg->getNodes();                    // the
template vector with the source nodes
      int max = dfg->getNumberOfNodes() - CONTROL_NODES;
      Vector< DFNPtr > tmp( max );
      // a temporary vector with only the necessary allocated space

      #if DEBUG
//  cout << "The Template vector:" << endl;
//  printVector( Template );
      #endif

      int position = 0;
      for ( int i = 0; i < Template.Length(); i++ ) {
            if ( Template[ i ] == NULL || Template[ i ] == &dfg->getUpNOP()
             || Template[ i ] == &dfg->getBottomNOP() )
                  continue;
            tmp[ position++ ] = Template[ i ];
      }

      #if DEBUG
//    cout << "The tmp vector:" << endl;
//    printVector( tmp );
      #endif

      for ( int i = 0; i < tmp.Length(); i++ ) {
            rnd = randomNumber( max-1 );
            pi[ i ] = tmp[ rnd ];                                       //
select a node
            tmp[ rnd ]   = tmp[ max-1 ];                          // to put the
selected node
```

```
            tmp[ max-1 ] = pi[ i ];                                    //
near the end
            max--;                                                     //
shrink the window in template
      }

      #if DEFAULTMODEL
      String name = "A";
      DFNPtr node = &dfg->getNode( name );
      pi[ 0 ] = node;
      name = "B";
      node = &dfg->getNode( name );
      pi[ 1 ] = node;
      name = "C";
      node = &dfg->getNode( name );
      pi[ 2 ] = node;
      name = "D";
      node = &dfg->getNode( name );
      pi[ 3 ] = node;
      name = "E";
      node = &dfg->getNode( name );
      pi[ 4 ] = node;
      name = "F";
      node = &dfg->getNode( name );
      pi[ 5 ] = node;
      name = "G";
      node = &dfg->getNode( name );
      pi[ 6 ] = node;
      name = "H";
      node = &dfg->getNode( name );
      pi[ 7 ] = node;
      name = "I";
      node = &dfg->getNode( name );
      pi[ 8 ] = node;
      name = "J";
      node = &dfg->getNode( name );
      pi[ 9 ] = node;
      name = "K";
      node = &dfg->getNode( name );
      pi[ 10 ] = node;
      #endif

      #if DEBUG
//    cout << "The PI vector:" << endl;
//    printVector( pi );
      #endif
```

```
}

//int SOLUTION::evaluateCost()
int SOLUTION::evaluateCost( int numberInteractions, double initialTemperature,
int bestLatency)
{
  ofstream outdfg("_DFG.daVinci");
  dfg->daVinciTerm(outdfg);
  outdfg.close();

  FILE       *Ptr;
  Ptr = fopen("fdct3.result", "a");

  if(!Ptr) {
    cerr << "Error opening " << "fdct1.result" << endl;
  }

  int ns;

  int ni = numberInteractions;
  double it = initialTemperature;
  for (int i = 0; i < 10; i++)
  {
    fprintf(Ptr, "Iteracao %d\n", i );
    fprintf(Ptr, "To       Io \n");
    fprintf(Ptr, "%f %d \n", initialTemperature, numberInteractions);

    fprintf(Ptr, "Tf       If It Cost\n");
    do
    {
      cost = simulatedAnnealing(ni, it, bestLatency, ns);
      it -= 0.1;
      //ni -= 5;

      fprintf(Ptr, "%f %d  %d   %d \n", it,ni,ns,cost);

    } while ( (it >= 0.1) && (ni > 0) );
    fprintf(Ptr, "%s\n","----------------------------------------");

    ni = numberInteractions;
    it = initialTemperature;
  }

  fclose(Ptr);
```

```
/*  generatePI();
  cost = Schedule();*/

       dpg->setNumOfRegisters(1);

       ofstream saida("_OUT.daVinci");
       smg->daVinciTerm(saida);
       saida.close();
       return( cost );
}

int SOLUTION::Schedule()
{
       List< STAPtr > todo = List< STAPtr >();              // the states to be
scheduled
       DFN & SourceNOP = dfg->getUpNOP();                          // the up NOP

       if ( SourceNOP.numberOfSuccessors() == 0 )
              return( EMPTY_SCHEDULING );                                  //
there are no operations available for scheduling

       dfg->annotateKeys( pi );                                    // marks the
nodes with their keys

       STA & sta = smg->addNode( smg->StateName() ); // creates the first state
to be scheduled
       sta.start( SourceNOP, smg->getNumberOfNodes() );// adds the successors
of the up NOP, it is,

       // the first candidates to be scheduled

       todo.push_front( &sta );                              // inserts in the
"to do list" the first state

       // to be scheduled

       // schedules while there is a state to be scheduled
       while ( todo.size() != 0 ) {
              STA & theState = *todo.front();                      // always get
the most recently added state to work
              Reservation = new RESERVATION( dpg->getConstraints() );
              theState.setReservation( *Reservation );      // the copy of the
model reservation table
              theState.schedule( smg->getNumberOfNodes() );// schedules one
state
```

```
            UpdateDataPath( theState.getReservation() );// updates the
datapath with the used resources
            theState.next( todo, *smg );                          // go to the
next state or finish scheduling
      }

      sta.start( dfg->getBottomNOP(), smg->getNumberOfNodes());

      return( smg->getNumberOfNodes() );
}

//Included by Xenia (03-12-2003)
int SOLUTION::Schedule(Vector < DFNPtr > & solution)
{
      List< STAPtr > todo = List< STAPtr >();             // the states to be
scheduled
      DFN & SourceNOP = dfg->getUpNOP();                  // the up NOP

      if ( SourceNOP.numberOfSuccessors() == 0 )
            return( EMPTY_SCHEDULING );                           //
there are no operations available for scheduling

      dfg->annotateKeys( solution );                        //
marks the nodes with their keys

      STA & sta = smg->addNode( smg->StateName() ); // creates the first state
to be scheduled
      sta.start( SourceNOP, smg->getNumberOfNodes() );// adds the successors
of the up NOP, it is,

      // the first candidates to be scheduled

      todo.push_front( &sta );                        // inserts in the
"to do list" the first state

      // to be scheduled

      // schedules while there is a state to be scheduled
      while ( todo.size() != 0 ) {
            STA & theState = *todo.front();               // always get
the most recently added state to work
            Reservation = new RESERVATION( dpg->getConstraints() );
            theState.setReservation( *Reservation );      // the copy of the
model reservation table
            theState.schedule( smg->getNumberOfNodes() );// schedules one
state
```

```
            UpdateDataPath( theState.getReservation() );// updates the
datapath with the used resources
            theState.next( todo, *smg );                          // go to the
next state or finish scheduling
      }

      sta.start( dfg->getBottomNOP(), smg->getNumberOfNodes());

      return( smg->getNumberOfNodes() );
}




int SOLUTION::randomNumber( float range )
{
      return ( (int)( ( range + 1.0 ) * rand() / RAND_MAX ) );
}

//Included by Xenia (02-12-2003)
double SOLUTION::randomDoubleNumber( float range )
{
      return ( ( ( range + 1.0 ) * rand() / RAND_MAX ) );
}

void SOLUTION::UpdateDataPath( RESERVATION & R )
{
      char   name[10];
      String cmpName, newCMP;

      for ( int i = 0; i < R.getNumberOfTypes(); i++ ) {
            int Occupied = R.getNumberOfOccupiedResources( i );
            for ( int j = 0; j < Occupied; j++ ) {
                  sprintf( name, "%s%d", ComponentNames[ i ], j );
                  cmpName = name;
                  CMP & cmp = dpg->getNode( cmpName );
                  newCMP = R.getOperation( j, i ).getName();
                  dpg->addNode( newCMP );
                  dpg->addEdge( cmp.getName(), newCMP, DEFAULT_COST );
            }
      }
}

void SOLUTION::printVector( Vector< DFNPtr> & theVector )
{
      cout << "|";
```

```
	for ( int i = 0; i < theVector.Length(); i++ ) {
		if ( theVector[ i ] == NULL )
			continue;
		if ( theVector[ i ]->getName() == BOTTOM_NOP )
			cout << "-" << "|";
		else if ( theVector[ i ]->getName() == UP_NOP )
			cout << "+" << "|";
		else
			cout << theVector[ i ]->getName() << "|";
	}
	cout << "\t" << theVector.Length() << endl;
}

#endif
```