UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

Bernardo Cardoso Rodrigues

Algoritmos Genéticos como ferramenta de otimização multiobjetivo no contexto de Co-Projeto Hardware/Software

São Carlos

2018

Bernardo Cardoso Rodrigues

Algoritmos Genéticos como ferramenta de otimização multiobjetivo no contexto de Co-Projeto Hardware/Software

Monografia apresentada ao Curso de Engenharia de Computação, da Escola de Engenharia de São Carlos e Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo, como parte dos requisitos para obtenção do título de Engenheiro de Computação.

Orientador: Prof. Dr. Alexandre Cláudio Botazzo Delbem

São Carlos

2018

AUTORIZO A REPRODUÇÃO TOTAL OU PARCIAL DESTE TRABALHO, POR QUALQUER MEIO CONVENCIONAL OU ELETRÔNICO, PARA FINS DE ESTUDO E PESQUISA, DESDE QUE CITADA A FONTE.

Ficha catalográfica elaborada pela Biblioteca Prof. Dr. Sérgio Rodrigues Fontes da EESC/USP com os dados inseridos pelo(a) autor(a).

CR696a

Cardoso Rodrigues, Bernardo
Algoritmos Genéticos como ferramenta de otimização multiobjetivo no contexto de Co-Projeto Hardware/Software / Bernardo Cardoso Rodrigues; orientador Alexandre Cláudio Botazzo Delbem. São Carlos, 2018.

Monografia (Graduação em Engenharia de Computação) -- Escola de Engenharia de São Carlos e Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo, 2018.

1. Co-Projeto Hardware Software. 2. Otimização Multiobjetivo. 3. Dominância de Pareto. 4. Hipervolume. 5. Algoritmos Genéticos. 6. NSGA-II. 7. Nios II. 8. Algoritmos Genéticos Paralelos. I. Título.

Eduardo Graziosi Silva - CRB - 8/8907

FOLHA DE APROVAÇÃO

Nome: Bernardo Cardoso Rodrigues

Título: "Algoritmos genéticos como ferramenta de otimização multiobjetivo no contexto de Co-Projeto Hardware/Software"

Trabalho de Conclusão de Curso defendido em 23/11/2018.

Comissão Julgadora:	**Resultado:**
Prof. Titular Alexandre Cláudio Botazzo Delbem (Orientador) - SSC/ICMC/USP	APROVADO
Prof. Associado Vanderlei Bonato SSC/ICMC/USP	APROVADO
Prof. Dr. João Navarro Soares Júnior SEL/EESC/USP	APROVADO

Coordenador do Curso Interunidades Engenharia de Computação:

Prof. Dr. Maximiliam Luppe

AGRADECIMENTOS

Gostaria de agradecer a todos que me apoiaram e acreditaram em minha jornada. Em especial, gostaria de agradecer aos meus pais, Marcondes e Nívia, por sempre estarem presentes e proporcionarem o melhor para o meu crescimento pessoal, acadêmico e profissional.

"Nullius in verba"

The Royal Society of London

RESUMO

RODRIGUES, B. **Algoritmos Genéticos como ferramenta de otimização multiobjetivo no contexto de Co-Projeto Hardware/Software**. 2018. 103p. Monografia (Trabalho de Conclusão de Curso) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2018.

O objetivo desse trabalho é realizar uma prova de conceito utilizando Algoritmos Genéticos como ferramenta de otimização multiobjetivo no contexto de Co-Projeto *Hardware/-Software*. Assume-se um particionamento e a partir dele, uma otimização de *hardware* é realizada. É proposta uma ferramenta baseada no MOGA NSGA-II com avaliação paralela que testa suas soluções em uma FPGA Intel/Altera a fim de realizar uma exploração orientada em um espaço de *design*. A intenção foi otimizar o *soft-core* Nios II para se adaptar a algoritmos, ou *softwares*, de referência. Foram utilizados a performance e o uso de recurso da FPGA como critérios de otimização. Cinco algoritmos foram referência para determinar o objetivo de performance. Os resultados preliminares demonstraram que a ferramenta proposta obteve respostas de mesma qualidade estatística que a referencia usada, um estimador da fronteira Pareto-Ótima. No entanto conclui-se não ser possível precisar a eficácia da ferramenta pois o número de experimentos não foi suficiente devido a limitações computacionais e de tempo.

Palavras-chave: Co-Projeto *Hardware Software*, Otimização multiobjetivo, Dominância de Pareto, Hipervolume, Algoritmos Genéticos, NSGA-II, Algoritmos Genéticos Paralelos, *Bootstrap*, FPGA, Nios II

ABSTRACT

RODRIGUES, B. **Genetic Algorithms as a multi objective optimization Hardware/Software Codesign tool**. 2018. 103p. Monografia (Trabalho de Conclusão de Curso) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2018.

The goal of this thesis is to implement a proof of concept using Genetic Algorithms as a Hardware/Software Codesign optimization tool. A partitioning is assumed and the hardware is evolved acording to a reference software. The proposed aplication is based on the MOGA NSGA-II and performs a design space exploration. The tested solutions are configured and benchmarked on a FPGA. The idea is to evolve the Nios II softcore to best suit a software benchmark using the least resources as possible. Five different benchmarks were used as reference software. The preliminary results show that the proposed tool reached a solution set statistically equivalent to the used reference, an estimtimator of the Pareto-Optimal frontier. However, it was not possible to assert the overall performance of the tool since more runs and experiments would be needed. This was not possible because of limited time and computational resources.

Keywords:Hardware/Software Codesign, Multiobjective optimization, Pareto dominance, Hipervolume, Genetic Algorithms, NSGA-II, Parallel Genetic Algorithms, Bootstrap, FPGA, Nios II

LISTA DE FIGURAS

Figura 1 – *Workflow* de desenvolvimento tradicional de um sistema de *Hardware*(HW) e *Software*(SW). 30
Figura 2 – *Workflow* de uma abordagem de Co-Projeto *Hardware/Software* salientando uma possível economia de tempo de desenvolvimento. 30
Figura 3 – Possível implementação do Algoritmo 3.1 em hardware 33
Figura 4 – Arquitetura de um computador de Von Neumann 34
Figura 5 – Comparativo entre paradigmas de computação: Flexilidade vs. Performance 35
Figura 6 – Organização de uma FPGA 36
Figura 7 – Uma porta XOR implementada em uma LUT de 2 entradas 37
Figura 8 – Um somador completo implementado em duas LUTs de 3 entradas . . 37
Figura 9 – Unidade base de uma FPGA Xilinx 37
Figura 10 – Fronteira de Pareto. Pontos verdes formam a fronteira. Pontos dominados em marrom. Região azul é a porção dominada pelo ponto x. 41
Figura 11 – Esboço de três metodologias de busca. 43
Figura 12 – Etapas de um Algorítimo Evolutivo 44
Figura 13 – Fluxo de um Algorítimo Genético simples 45
Figura 14 – Exemplo de uma representação cromossômica de uma população e seus indivíduos. 46
Figura 15 – Exemplos de recombinação: um corte, dois cortes e uniforme 48
Figura 16 – Comparativo entre a implementação de um GA mestre-escravo e um GA sequencial. Na abscissa têm-se tempo de cálculo da função de aptidão em segundos e na ordenada tem-se o tempo total de computação para uma geração. 50
Figura 17 – Fronteiras de dominância de uma população. 51
Figura 18 – Funcionamento do operador de seleção no NSGA-II 52
Figura 19 – Cálculo do *Crowding distance* 52
Figura 20 – Hipervolume $I_H(f(x), r)$, área grifada, do conjunto de soluções A descrito pela função $f(x)$ em relação ao ponto r 55
Figura 21 – Exemplo de uma computação de Hipervolume $I_H(f(x), r)$. 56
Figura 22 – Contribuição H_i do ponto $(x_i^\mu, f(x_i^\mu))$ no valor do Hipervolume I_H. . . 56
Figura 23 – Diagrama de blocos de um Nios II. Blocos azuis são necessários, já os cinzas são opcionais 60
Figura 24 – Diagrama de blocos de um sistema que usa um Nios II. 61
Figura 25 – Estrutura de um projeto Quartus Prime com sistema Nios II. 64
Figura 26 – Camadas de abstração de um sistema baseado no HAL 66

Figura 27 – Estrutura de uma aplicação de software para um sistema Nios II. . . . 67
Figura 28 – Um sistema completo. 71
Figura 29 – Arquitetura em alto nível da implementação 73
Figura 30 – Bloco *Gerador de Indivíduos* 74
Figura 31 – Bloco *Avaliador* 75
Figura 32 – Passos 1 a 4 do *Avaliador* 77
Figura 33 – Passos 5 a 7 do *Avaliador* 77
Figura 34 – Passos 8 a 10 do *Avaliador* 78
Figura 35 – Passos 8 a 10 do *Avaliador* 79
Figura 36 – Agrupamento de algoritmos com perfis próximos 81
Figura 37 – Espaço de objetivos para otimização $f_{execução}$ e f_{ALM} com kernel *adpcm*. Valores normalizados. 84
Figura 38 – Histograma dos valores de hipervolume da fronteira de Pareto obtidos após 100000 iterações de Boostrap sobre da amostra obtida no experimento *Mil Cores* para o *kernel adpcm*. Valores normalizados. . . 85
Figura 39 – Evolução do hipervolume das fronteiras de Pareto para 5 execuções. Valores normalizados. 86
Figura 40 – Evolução do hipervolume das fronteiras de Pareto para 5 execuções (detalhe). Valores normalizados. 86
Figura 41 – Fronteira de Pareto: Mil cores vs. NSGA Execução 1. Valores normalizados. 87
Figura 42 – Fronteira de Pareto: Mil cores vs. NSGA Execução 2. Valores normalizados. 87
Figura 43 – Histograma das ocorrências do hipervolume da fronteira de Pareto após 100000 iterações de Bootstrap na última geração das 5 execuções. Valores normalizados. 89
Figura 44 – Boxplot das dispersões de hipervolume do experimento *Mil Cores* e execuções de NSGA-II. Valores normalizados. 90
Figura 45 – Boxplot das dispersões de hipervolume do experimento *Mil Cores* e execuções de NSGA-II (Detalhe). Valores normalizados. 90
Figura 46 – Espaço de objetivos (visão 1). x = $f_{execução}$, y = f_{ALM}, z = $f_{memória}$. Valores normalizados. 91
Figura 47 – Espaço de objetivos (visão 2). x = $f_{execução}$, y = f_{ALM}, z = $f_{memória}$. Valores normalizados. Valores normalizados. 91
Figura 48 – Fronteiras de Pareto *Mil Cores 3 Obj.* vs. Fronteiras de Pareto NSGA-II para kernel *adpcm* (visão 1). x = $f_{execução}$, y = f_{ALM}, z = $f_{memória}$. Valores normalizados. Valores normalizados. 91
Figura 49 – Fronteiras de Pareto *Mil Cores 3 Obj.* vs. Fronteiras de Pareto NSGA-II para kernel *adpcm* (visão 2). x = $f_{execução}$, y = f_{ALM}, z = $f_{memória}$. Valores normalizados. Valores normalizados. 91

Figura 50 – Evolução do hipervolume para 5 otimizações com *kernels* diferentes. Valores normalizados. 92
Figura 51 – Histograma dos valores do hipervolume obtidos após 100000 iterações de *Bootstrap* sobre das fronteiras de Pareto encontradas pelo *Mil Cores 3 Obj.* e pelo NSGA. *Kernel* usado *adpcm*. Valores normalizados. 95
Figura 52 – Histograma dos valores do hipervolume obtidos após 100000 iterações de Bootstrap sobre das fronteiras de Pareto encontradas pelo *Mil Cores 3 Obj.* e pelo NSGA. Kernel usado *sobel*. Valores normalizados. 95
Figura 53 – Histograma dos valores do hipervolume obtidos após 100000 iterações de Bootstrap sobre das fronteiras de Pareto encontradas pelo *Mil Cores 3 Obj.* e pelo NSGA. Kernel usado *dotprod*. Valores normalizados. . . . 96
Figura 54 – Histograma dos valores do hipervolume obtidos após 100000 iterações de Bootstrap sobre das fronteiras de Pareto encontradas pelo *Mil Cores 3 Obj.* e pelo NSGA. Kernel usado *vecsum*. Valores normalizados. . . . 97
Figura 55 – Histograma dos valores do hipervolume obtidos após 100000 iterações de Bootstrap sobre das fronteiras de Pareto encontradas pelo *Mil Cores 3 Obj.* e pelo NSGA. Kernel usado *quicksort*. Valores normalizados. . . 97
Figura 56 – Boxplot das dispersões de hipervolume *Mil Cores 3Obj.* e NSGA-II para os 5 kernels diferentes. Valores normalizados. 98

LISTA DE TABELAS

Tabela 1 – Cromossomo correspondete a um Nios II 70

LISTA DE QUADROS

Quadro 1 – Variações da ULA do Nios II/f 63
Quadro 2 – Dados estatísticos da dispersão do hipervolume apresentada na Figura 38 85
Quadro 3 – Evolução dos objetivos para 1ª execução: Primeira fronteira vs. última fronteira 88
Quadro 4 – Evolução dos objetivos para 2ª execução: Primeira fronteira vs. última fronteira 88
Quadro 5 – Evolução dos objetivos para 3ª execução: Primeira fronteira vs. última fronteira 88
Quadro 6 – Evolução dos objetivos para 4ª execução: Primeira fronteira vs. última fronteira 88
Quadro 7 – Evolução dos objetivos para 5ª execução: Primeira fronteira vs. última fronteira 89
Quadro 8 – Dados estatísticos da dispersão apresentada na Figura 38 e Figura 43 89
Quadro 9 – Evolução dos objetivos para o *kernel adpcm*: Primeira fronteira vs. última fronteira 93
Quadro 10 – Evolução dos objetivos para o *kernel vecsum*: Primeira fronteira vs. última fronteira 93
Quadro 11 – Evolução dos objetivos para o *kernel sobel*: Primeira fronteira vs. última fronteira 93
Quadro 12 – Evolução dos objetivos para o *kernel dotprod*: Primeira fronteira vs. última fronteira 93
Quadro 13 – Evolução dos objetivos para o *kernel quicksort*: Primeira fronteira vs. última fronteira 94
Quadro 14 – Dados estatísticos da dispersão apresentada na Figura 51 95
Quadro 15 – Dados estatísticos da dispersão apresentada na Figura 52 96
Quadro 16 – Dados estatísticos da dispersão apresentada na Figura 53 96
Quadro 17 – Dados estatísticos da dispersão apresentada na Figura 54 97
Quadro 18 – Dados estatísticos da dispersão apresentada na Figura 55 98

LISTA DE ALGORITMOS

3.1 Exemplo de um algoritmo 33
5.1 Algoritmo Genético canônico 45
8.1 Implementação da prova de conceito 74
8.2 Pseudocódigo da bloco gerador de indivíduos, equivalente a chamada de função gera_população_inicial() 75
8.3 Pseudocódigo da base para *benchmark* 80

LISTA DE ABREVIATURAS E SIGLAS

HW	Hardware
SW	Software
CPU	Central Processing Unit
ASIC	Application Specific Integrated Cirtuit
DSE	Design Space Exploration
FPGA	Field Programable Gate Array
DSP	Domain Specific Processor
LUT	Lookup Table
HDL	Hardware Description Language
SoC	System On a Chip
AE	Algoritmo Evolutivo
AG	Algoritmo Genético
EA	Evolutionary Algorithm
GA	Genetic Algorithm
MOEA	Multi-Objective Evolutionary Algorithm
MOGA	Multi-Objective Genetic Algorithm
IP	Intelectual Property
RISC	Reduced Instruction Set Computer
HAL	Hardware Abstraction Layer
BSP	Board Suport Package
SBT	Software Build Tools
ULA	Unidade Lógico e Aritmetica
ECC	Error-Correcting Code
MMU	Memory Management Unit

GNU	GNU's Not Unix
NSGA-II	Non-dominated Sorting Genetic Algorithm II

SUMÁRIO

1 INTRODUÇÃO . . . 27
1.1 Motivação . . . 27
1.2 Objetivos . . . 27
1.3 Organização do Trabalho . . . 28

2 CO-PROJETO DE HARDWARE/SOFTWARE . . . 29
2.1 Particionamento . . . 30
2.2 Exploração do Espaço de *Design* (DSE) . . . 31
2.2.1 Espaço de *Design* . . . 32
2.2.2 Espaço de Objetivos . . . 32

3 COMPUTAÇÃO RECONFIGURÁVEL . . . 33
3.1 Field Programable Gate Array . . . 36
3.2 Processadores Soft-Core . . . 38

4 OTIMIZAÇÃO . . . 39
4.1 Otimização Multiobjetivo . . . 39
4.1.1 Dominância de Pareto . . . 40

5 MÉTODOS DE EXPLORAÇÃO DE ESPAÇO DE *DESIGN* . . . 43
5.1 Busca Aleatória . . . 43
5.2 Algoritmos Evolutivos . . . 43
5.3 Algoritmos Genéticos . . . 44
5.3.1 Genótipo . . . 46
5.3.2 População inicial . . . 46
5.3.3 Seleção . . . 47
5.3.4 Recombinação . . . 47
5.3.5 Mutação . . . 48
5.3.6 Inserção . . . 49
5.3.7 Algoritmos Genéticos Paralelos . . . 49
5.4 Algoritmos Evolutivos Multi-objetivo . . . 50
5.4.1 NSGA-II: Non-dominated Sorting Genetic Algorithm II . . . 51

6 AVALIAÇÃO DE DESEMPENHO . . . 55
6.1 Hipervolume . . . 55
6.2 Amostragem . . . 57
6.2.1 Amostragem aleatória simples . . . 57

6.2.2 Bootstrap 57

7 *SOFT-CORE* NIOS II 59
7.1 Arquitetura 59
7.2 Nios II/f 61
7.2.1 *Cache* de Instruções e de Dados 62
7.2.2 Implementações aritméticas 62
7.2.3 Predição de condicional 63
7.3 Desenvolvimento de Hardware de um sistema Nios II 63
7.3.1 Estrutura de um projeto 64
7.4 Desenvolvimento de *software* no Nios II 65
7.4.1 HAL : *Hardware Abstraction Layer* 66
7.4.2 Estrutura de arquivos de um Projeto de software 67

8 IMPLEMENTAÇÃO DA PROVA DE CONCEITO 69
8.1 Modelagem dos indivíduos 69
8.2 Definição do problema de otimização 71
8.3 Arquitetura da implementação 72
8.4 Gerador de indivíduos 74
8.5 Avaliador 75
8.5.1 Implementação Paralelizada 78
8.6 Operadores do AG 79
8.7 Seleção do AG 80
8.8 Benchmarking 80
8.8.1 *Kernels* referência 81

9 RESULTADOS 83
9.1 Ambiente de teste 84
9.2 Otimização de dois objetivos: $f_{execução}$ e f_{ALM} 84
9.2.1 Estimador da fronteira Pareto-Ótima para 2 objetivos 84
9.2.2 Otimização com NSGA-II 86
9.3 Otimização de três objetivos: $f_{execução}$, f_{ALM} e $f_{memória}$ 90

10 CONCLUSÕES 99
10.1 Trabalhos futuros 100

REFERÊNCIAS 101

1 INTRODUÇÃO

1.1 Motivação

Teich (2012) descreve como técnicas de co-projeto *Hardware/Software* estão cada vez mais cotadas no contexto de *design* de *hardware* por conta da crescente demanda por sistemas computacionais mais rápidos e eficientes. A questão é que muitas vezes essa demanda não é simplesmente suprida. Sistemas computacionais são incrivelmente complexos e sensíveis, com incontáveis variáveis associadas. Encontrar o *design* ideal para uma aplicação é uma tarefa laboriosa, de elevado custo, e até mesmo impossível, sem as ferramentas adequadas.

Até recentemente, o problema de desempenho de sistemas computacionais era primariamente solucionado com o aumento da velocidade de chaveamento das portas lógicas por meio de tecnologias de transistores mais avançadas. No entanto, cada vez mais projetistas de *hardware* flertam com os limites físicos dos componentes eletrônicos. Por isso, arquiteturas de *hardware* disruptivas têm resultado em melhores ganhos de performance.

Mishra et al. (2012), Teich (2012), Koudil et al. (2007), Catania, Malgeri e Russo (1997) e Silva (2016) expõem como o uso de Inteligência Artificial e Aprendizado de Máquina apresenta-se como uma excelente ferramenta no estado da arte do co-projeto *Hardware/Software*. Essas técnicas propõe uma nova forma de contemplar e modelar as incontáveis variáveis mencionadas e, a partir delas, mostram o melhor caminho a ser trilhado. Isso é importante para uma indústria que pouco perdoa erros de projeto. Uma escolha equivocada no contexto da microeletrônica pode resultar em um grande prejuízo financeiro, meses de trabalho perdido, uma mancha na reputação de uma multinacional e até *recalls* em escalas globais.

1.2 Objetivos

O objetivo desse trabalho é realizar uma prova de conceito: verificar a viabilidade do uso da meta-heurística dos Algoritmos Genéticos para uma otimização multiobjetivo no contexto de Co-Projeto *Hardware/Software*. Em outras palavras, deseja-se propor uma ferramenta de auxílio ao design de *hardware* que seja capaz de evoluir núcleos de processamento a fim de obter as melhores soluções dado um *software*, ou particionamento, de referência. Frank et al. (2018) mostra o uso da mesma meta-heurística para otimização de designs de veículos aéreos.

Gostaria-se de trazer resultados e implementações mais palpáveis para um contexto real de engenharia. Ou seja, quer-se atender uma possível demanda existente com o trabalho proposto. Por conta disso, as ferramentas e conceitos utilizados terão de ser

os mais próximos possíveis da realidade. Por fim, deseja-se determinar se Algoritmos Genéticos apresentam uma boa alternativa para solucionar o problema de otimização apresentado.

1.3 Organização do Trabalho

Este trabalho está dividido da seguinte maneira:

- **Capítulo 1:** Atual capitulo.
- **Capítulo 2:** Apresenta os conceitos do Co-Projeto *Hardware/Software*. Embasamento teórico sobre Exploração do Espaço de *Design*. Definição dos espaços de *Design* e de Objetivos.
- **Capítulo 3:** Trata dos conceitos de computação reconfigurável. Traz uma revisão sobre outros paradigmas de *design* de *hardware* como o ASIC e o de Von Neumann. Definição de um processador *soft-core*.
- **Capítulo 4:** Define problemas de otimização, otimização multiobjetivo e dominância de Pareto.
- **Capítulo 5:** Embasamento teórico em métodos de exploração de *design*. Explanação sobre Algoritmos Genéticos e seus operadores. Apresenta conceitos de algoritmos evolutivos multiobjetivos. Apresenta o algoritmo NSGA-II e suas particularidades.
- **Capítulo 6:** Conceitos de hipervolume, re-amostragem e *Bootstrap*.
- **Capítulo 7:** Definição da arquitetura do Nios II e do ambiente de desenvolvimento HW/SW da Intel/Altera.
- **Capítulo 8:** Apresenta e define a ferramenta proposta pelo trabalho bem como seus blocos constituintes.
- **Capítulo 9:** Resultados experimentais da ferramenta proposta.
- **Capítulo 10:** Conclusões acerca do trabalho e seus resultados.

2 CO-PROJETO DE HARDWARE/SOFTWARE

O conceito do Co-Projeto *Hardware/Software* prega o *design* concorrente e casado de elementos de *software* e de *hardware* em sistemas eletrônicos a fim de satisfazer da melhor maneira possível os requisitos de um projeto (performance, custo, robustez, tempo de desenvolvimento, etc.). Isso é feito por meio da sinergia existente entre o *hardware* e o *software* identificada pelos seus projetistas. (MICHELI; GUPTA, 1997)

Espera-se que com a proximidade dos limites da *lei de Moore*[1], a prática do Co-Projeto seja cada vez mais importante na busca por sistemas computacionais melhores.

As idéias e propósitos do Co-Projeto HW/SW podem ser resumidas da seguinte maneira (TEICH, 2012):

- **Coordenação:** Técnicas de Co-Projeto são usadas para organizar as iterações de um *workflow* interdisciplinar onde *hardware* e *software* são desenvolvidos por times de especialistas separados, mas que dialogam entre si para atingir um objetivo comum. Isso implica em um desenvolvimento casado do *hardware* (*designers* de sistemas eletrônicos, *chips*, SoCs) e *software* (desenvolvedores de *firmware*, sistema operacional e aplicação).

- **Concorrência:** Tanto o *hardware* quanto o *software* são desenvolvidos paralelamente a fim de minimizar o *tempo-até-mercado*[2]. Isso quer dizer que a equipe de desenvolvedores de *software* deve realizar seu trabalho antes mesmo que a plataforma de *hardware* esteja pronta. Técnicas de prototipação virtual são usadas para solucionar essa questão.

- **Corretude:** Mesmo que os projetos de *hardware* e *software* aconteçam separadamente, é imperativo que ambos funcionem como foram especificados (individualmente e após integração).

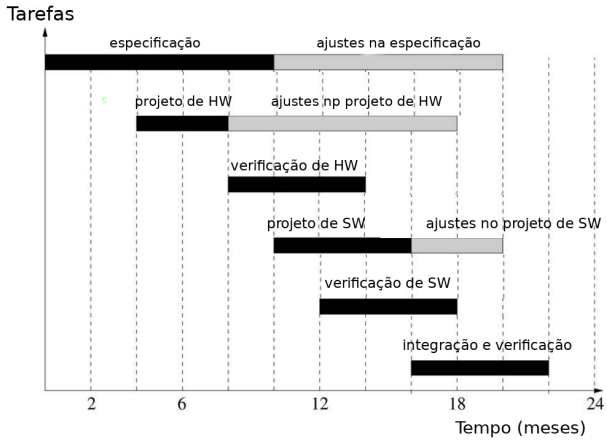
- **Complexidade:** A necessidade de sistemas cada vez mais complexos é uma motivação para o uso do Co-Projeto. Técnicas modernas auxiliam na implementação desses sistemas em tempos reduzidos e com qualidade.

---

[1] Observação feita por Gordon Moore, co-fundador da Intel e da Fairchild, ao notar que a densidade de transistores em circuitos integrados em geral dobra a cada 2 anos.

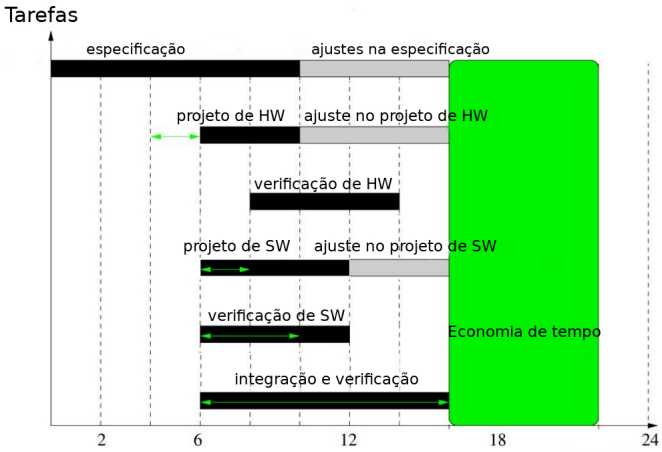
[2] Tempo desde a concepção de um produto até ele estar disponível no mercado

Figura 1: *Workflow* de desenvolvimento tradicional de um sistema de *Hardware*(HW) e *Software*(SW).



Adaptado de: Teich (2012, p. 1415)

Figura 2: *Workflow* de uma abordagem de Co-Projeto *Hardware/Software* salientando uma possível economia de tempo de desenvolvimento.



Adaptado de: Teich (2012, p. 1415)

Nas figuras 1 e 2, podemos ver uma comparação entre um *workflow* tradicional de desenvolvimento e um que utiliza o Co-Projeto *Hardware/Software*. Nessa comparação é salientada a melhora na rapidez com que os resultados são entregues. Apesar desse ser um ponto importante a favor do Co-Projeto, ele não é o único. Sistemas mais eficientes e de melhor performance são viabilizados com essa metodologia. Isso se deve pois há um esforço ativo na otimização tanto do hardware, como no *software* e na integração desses elementos.

Tradicionalmente, as técnicas de Co-Projeto focavam em especificar uma bipartição de uma aplicação segundo uma especificação. Um parte seria implementada em *software* e executaria em uma CPU, enquanto a outra parte seria implementada em *hardware*, um ASIC (TEICH, 2012). Atualmente, o campo de pesquisa do co-projeto *Hardware/Software* está mais diversificado e interdisciplinar. Conceitos e ferramentas modernos de otimização, prototipação, programação, *design* e organização de projetos foram incorporados às ferramentas. A *Exploração do Espaço de Projeto* (seção 2.2) e as *FPGAS* (Capítulo 3) são exemplos da adição mencionada.

2.1 Particionamento

Uma das principais tarefas realizadas no processo de co-projeto é o particionamento. Essa atividade consiste na identificação dos subproblemas, também conhecidos como *kernels*, que compõem o sistema a ser desenvolvido. Com esses blocos funcionais em

mãos, é decidido quais devem ser implementados em *software* ou em *hardware* segundo os requisitos estabelecidos. (HIDALGO; LANCHARES, 1997)

O particionamento deve ser feito com muita prudência já que é um dos passos iniciais do processo de desenvolvimento e irá impactar diretamente todas as etapas seguintes. A escolha de quais elementos serão implementados em *hardware* ou em *software* é pautada pela especificação do sistema, levando em consideração performance, custo, consumo de energia etc.

2.2 Exploração do Espaço de *Design* (DSE)

Design Space Exploration, ou Exploração do Espaço de *Design*, consiste em explorar as possibilidades de implementação de *design* da aplicação em questão a fim de encontrar soluções não triviais que de alguma forma se destaquem sobre as demais. A DSE é uma ferramenta muito utilizada nas tarefas de: (KANG; JACKSON; SCHULTE, 2010)

- **Prototipagem:** A DSE é capaz de gerar conjuntos de protótipos além de realizar suas simulações e *profiling*[3]. Isso permite aos projetistas ter uma melhor noção de quais decisões podem impactar o projeto tanto de maneira positiva como negativa.
- **Otimização:** Quando a DSE é aliada às métricas de qualidade das soluções que são exploradas, é possível realizar otimizações automatizadas no espaço de buscas para obter um conjunto de soluções interessantes que serão analisadas em detalhe posteriormente.
- **Integração de Sistema:** Como a tarefa de integração e configuração dos vários componentes de um sistema pode se tornar extensa, a DSE pode ser utilizada para gerar combinações válidas dessas configurações segundo dadas restrições.

O uso da DSE deve ser cuidadoso pois dependendo de como um sistema é modelado, pode haver milhões de possíveis soluções a serem exploradas. Isso pode inviabilizar uma abordagem não otimizada ou não inteligente. Por isso, uma boa DSE deve levar em consideração os seguintes pontos: (KANG; JACKSON; SCHULTE, 2010)

- **Representação:** A representação do espaço de busca deve ser expressiva e formal. Dessa forma, técnicas de análise e exploração automática podem ser usadas.
- **Análise:** A busca deve garantir que todas as soluções encontradas sejam factíveis e atendam as restrições de design impostas.

[3] Uma forma de análise dinâmica da execução de um algoritmo em um hardware que pode incluir, por exemplo, uso de memória, tempo de execução, uso de instruções, frequência e duração de chamadas de funções, etc.

- **Método de exploração:** O sistema deve proporcionar uma navegação das soluções encontradas agradável aos projetistas. Cabe à DSE garantir que soluções muito parecidas sejam resumidas e que o conjunto de soluções interessantes seja o mais expressivo e diversificado possível.

2.2.1 Espaço de *Design*

Esse espaço se refere ao conjunto de todas soluções, ou decisões, possíveis para o problema proposto. Ele é definido pelos parâmetros variáveis que compõe as soluções (MARTINS, 2015b). No contexto do Co-Projeto HW/SW, exemplos de parâmetros vão desde quais configurações de compilador utilizar até o tipo de implementação de unidade de ponto flutuante da CPU. O número de variáveis, discretas ou contínuas, define a cardinalidade desse espaço.

2.2.2 Espaço de Objetivos

O espaço de objetivos é definido pelos valores das métricas utilizadas para quantizar a qualidade das soluções testadas. O número de objetivos usados define a cardinalidade desse desse espaço (MARTINS, 2015b).

3 COMPUTAÇÃO RECONFIGURÁVEL

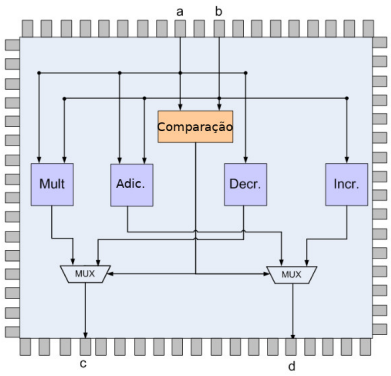
A implementação de um algoritmo pode ser feita tanto via *software* como *hardware*. (COMPTON; HAUCK, 2000) É evidente que cada paradigma tem suas particularidades e que o uso mais indicado de um ou de outro é dependente de cada caso. Com o passar dos anos surgiram híbridos dos dois, alguns flertando mais com um dos paradigmas do que com o outro. Dessas tecnologias, pode-se destacar os ASICs, os DSPs, a computação reconfigurável e a computação de Von Neumann.

Implementações de *Application Specific Integrated Circuits* (ASICs) realizam todas suas operações em *hardware* e por isso têm um desempenho superior aos demais paradigmas de implementação. No entanto, são inflexíveis pois o *hardware* é projetado para uma única aplicação e tendem a ter um desenvolvimento de custo mais elevado (COMPTON; HAUCK, 2000). Em um ASIC, as instruções são implementadas direto no *hardware*. Na Figura 3 pode-se ver uma implementação em *hardware*, a nível de unidades lógicas, do Algoritmo 3.1

Algoritmo 3.1: Exemplo de um algoritmo

```
1   se a < b então
2       d = a + b
3       c = a * b
4   senão
5       d = b + 1
6       c = a - 1
7   fim se
```

Figura 3: Possível implementação do Algoritmo 3.1 em hardware



Adaptado de: BOBDA (2007, p. 7)

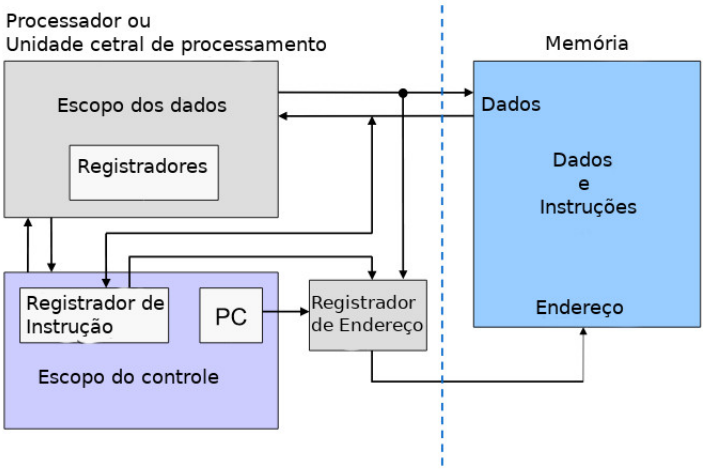
Do outro lado do espectro, tem-se a implementação em *software* que se entende por uma aplicação projetada segundo o paradigma de Von Neumann. John Von Neumann foi um matemático que concebeu a ideia de computação de propósito geral. Ele mostrou que era possível ter um *hardware* genérico, com implementações de operações diversificadas, que, se controlado corretamente, poderia realizar qualquer tipo de computação (BOBDA, 2007). Essa arquitetura se mostrou extremamente bem sucedida pois foi um dos pilares para quase todos os computadores digitais que viriam a seguir. Da ideia de um *hardware* de propósito geral surgiram os microprocessadores e da ideia do seu controle ordenado surgiram os programas de computador.

Em um computador de Von Neumann, um grupo de instruções (o programa) é executado sequencialmente a partir de uma unidade de memória. Os dados, também coletados de uma memória, são processados conforme descrito nas instruções e os resultados, então são escritos de volta à memória.

Na Figura 4, vê-se a estrutura geral de um computador de Von Neumann:

- Uma unidade de memória onde estão guardadas as instruções e os dados
- Uma unidade de controle (escopo de controle) que contém a instrução atual e o endereço para a próxima.
- Uma unidade de lógica e aritmética que executa as instruções.

Figura 4: Arquitetura de um computador de Von Neumann



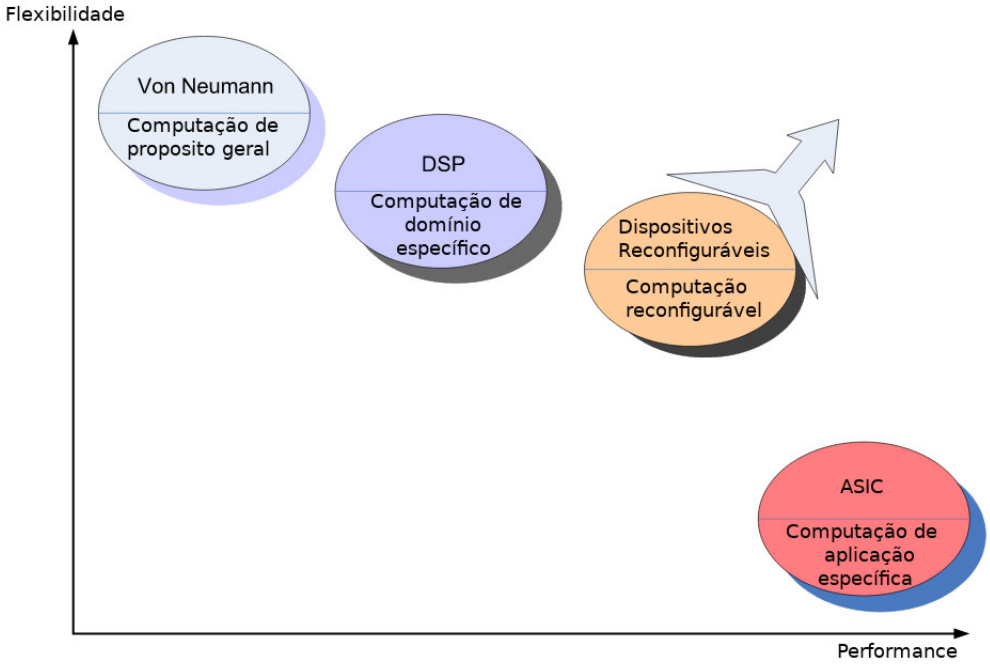
Adaptado de : BOBDA (2007, p. 2)

Os DSPs, ou *Domain Specific Processor*, são processadores adaptados para uma classe de algoritmos ou aplicações. O escopo de dados é feito de forma otimizada para um grupo específico de operações e além disso, tenta-se reduzir acessos à memória. Os processadores digitais de sinais talvez sejam os DSPs mais difundidos mundialmente. Como muitas das aplicações de um DSP envolvem computação em repetição, existe um suporte especial a nível de *hardware* para *loops* eficientes.

Pode-se ver a partir da discussão dos parágrafos anteriores que há duas maneiras de se classificar um processador: flexibilidade e performance (BOBDA, 2007). Vê-se também que há um equilíbrio entre as duas; não existindo uma tecnologia que seja a melhor dos dois mundos. Idealmente, gostaríamos de ter a flexibilidade de um computador de Von Neumann e o desempenho de um ASIC.

A computação reconfigurável vem como uma outra opção que se propõe a ser um bom balanço entre flexibilidade e performance. É chamado de um dispositivo reconfigurável aquele que consiga modificar sua estrutura lógica em tempo de inicialização e/ou execução. A intenção é que o dispositivo se adapte logicamente para melhor se aproximar da aplicação em questão. Na Figura 5, vê-se uma ilustração de como cada um dos paradigmas supracitados se relacionam quando se observa a flexibilidade e a performance de cada um.

Figura 5: Comparativo entre paradigmas de computação: Flexilidade vs. Performance



Adaptado de : BOBDA (2007, p. 8)

Prototipação rápida é, com certeza, um dos destaques atribuídos à computação reconfigurável. O desenvolvimento de um ASIC é uma tarefa árdua que demanda uma equipe extremamente capacitada de projetistas e, ao contrário de uma aplicação em *software*, uma aplicação de *hardware* que apresente uma falha após seu lançamento resulta em um prejuízo enorme, tanto de tempo como financeiro (BOBDA, 2007). Como um dispositivo reconfigurável pode ser facilmente alterado, é possível testar várias implementações do projeto antes de sua reprodução em larga escala.

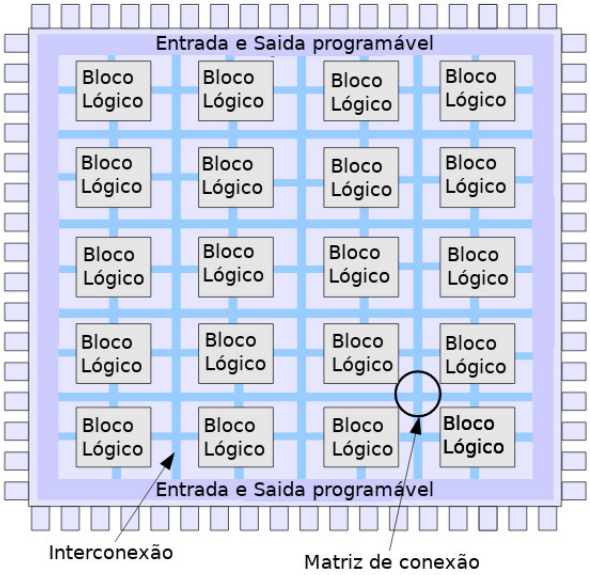
Normalmente, um dispositivo reconfigurável é composto por um conjunto de unidades lógicas programáveis e uma matriz de interconexões. O que caracteriza a aplicação no final das contas é como essas unidades e as conexões são formatadas para gerar um caminho de dados específico que resulte na semântica lógica desejada. Quanto maior a granularidade do dispositivo, ou seja, quanto mais básicas forem as unidades lógicas endereçáveis, mais flexível é o dispositivo (BONDALAPATI; PRASANNA, 2002). No

entanto, uma alta granularidade também implica num maior tempo de configuração.

3.1 Field Programable Gate Array

As FPGAs (*Field Programable Gate Arrays*) são dispositivos reconfiguráveis inicialmente desenvolvidos pela empresa Xilinx em 1985. Elas são construídas primariamente por 3 elementos: células logico-programáveis, uma rede de interconexões e células de entrada/saída. Uma aplicação implementada em uma FPGA é basicamente um arranjo desses elementos configurados de uma maneira particular.

Figura 6: Organização de uma FPGA



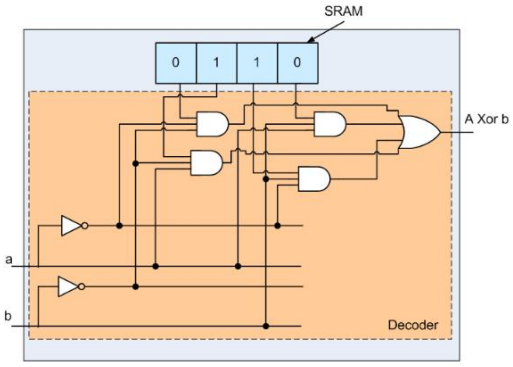
Adaptado de : BOBDA (2007, p. 29)

Na Figura 6, pode-se ver os elementos programáveis que constituem uma *FPGA*. Unidades de memória embarcadas nas FPGAs são usadas para carregar as configurações dos elementos lógicos e das interconexões. A maioria desses dispositivos modernos utiliza a tecnologia de SRAM para isso. Os bits dessa memória são conectados aos pontos de configuração da FPGA e são responsável pelo acionamento dos modos de operação das unidades lógicas e suas interconexão. O dispositivo pode ser facilmente reprogramado quando alterados os valores salvos nessas memórias (BOBDA, 2007).

As *Lookup-tables* (LUTs) são geralmente utilizadas nas FPGAs como estruturas computacionais (COMPTON; HAUCK, 2000) sendo que nelas, pequenas células de memória contêm todos os resultados de uma função dadas todas as possíveis estradas. As LUTs são feitas de forma que dado um conjunto de entradas, a saída esperada seja rapidamente obtida. Uma LUT de n entradas pode ser usada para implementar até 2^{2^n} soluções

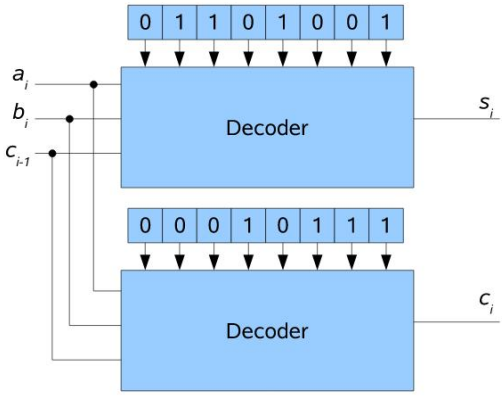
(BOBDA, 2007). Fisicamente, uma LUT é implementada por unidades da memória SRAM e um decodificador como pode ser visto nas figuras 7 e 8. Funções mais complexas são quebradas em elementos fundamentais que então são implementados em diversas LUTs e são conectadas por meio da matriz de interconexões.

Figura 7: Uma porta XOR implementada em uma LUT de 2 entradas



BOBDA (2007, p. 37)

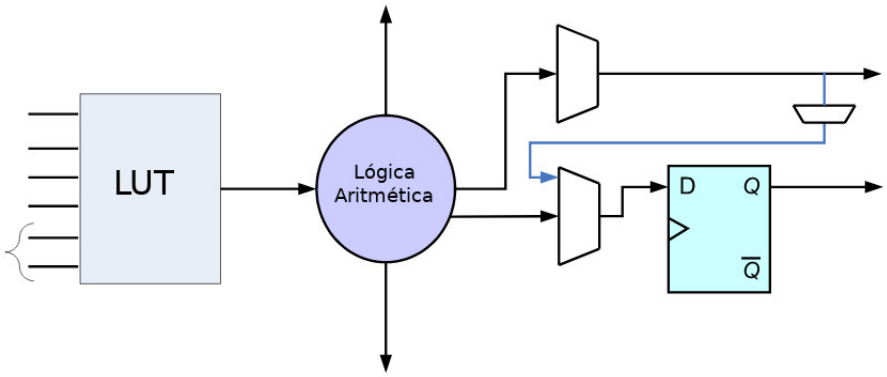
Figura 8: Um somador completo implementado em duas LUTs de 3 entradas



BOBDA (2007, p. 37)

Na Figura 9 pode-se ver como algumas estruturas lógicas podem ser combinadas para formar um elemento base de uma arquitetura FPGA.

Figura 9: Unidade base de uma FPGA Xilinx



Adaptado de : BOBDA (2007, p. 38)

Linguagens de descrição de *hardware* (HDLs) são utilizadas pelos desenvolvedores para a descrição formal dos circuitos e aplicações. Essas linguagens permitem a descrição de toda a lógica e semântica de uma aplicação. Em uma próxima etapa, essa descrição pode ser simulada virtualmente ou mesmo implementada em uma FPGA. O programa

responsável por converter o código de HDL em um *netlist* equivalente é chamado de *Synthesizer* (DIAS, 2011).

3.2 Processadores Soft-Core

Com o aumento do custo e do *tempo até prateleira* de dispositivos ASIC estado da arte, um número crescente de sistemas embarcados tem sido implementados em FPGAs. Como muitas dessas aplicações são embarcadas em conjunto com um microprocessador, os fabricantes já vem oferecendo FPGAs com SoCs que incluem processadores como os da família ARM. No entanto, na maioria dos casos, o processador embarcado ou é sobre ou é sub-dimensionado para a aplicação. Por isso houve esforços para implementar processadores na própria lógica reconfigurável da FPGA. Dessa maneira o *designer* poderia definir como o processador seria dada sua necessidade e requerimentos. Esses processadores em FPGA são chamados de *soft-cores*. Apesar de flexíveis, os *soft-cores* apresentam desempenho e eficiência piores que um *hard-core* análogo, ou processador comum (YIANNACOURAS; ROSE; STEFFAN, 2005).

4 OTIMIZAÇÃO

Um problema de otimização é definido pela intenção de maximizar ou minimizar uma função $f(x_1, \ldots, x_n)$, onde n é numero de variáveis do problema. No entanto, normalmente esses problemas também incluem um número de restrições, que podem ser aplicadas às variáveis ou mesmo a $f(x)$. Deb (2000) define que um problema de otimização com restrições pode ser escrito na forma de um problema de programação não linear como a seguir:

$$\begin{array}{lll} \text{Minimizar} & f(\vec{x}\,) & \\ \text{Sujeito a} & g_j(\vec{x}\,) \leq 0 & j = 1, \ldots, J, \\ & h_k(\vec{x}\,) = 0 & k = 1, \ldots, K, \\ & x_i^l \leq x_i \leq x_i^u & i = 1, \ldots, n. \end{array}$$

Nessa definição, $f(\vec{x}\,)$ é a função objetivo, ou seja, a função a ser otimizada. Nesse caso, quer-se que $f(\vec{x}\,)$ seja minimizada. $\vec{x}$ é uma representação vetorial de todas as n variáveis do problema. $g_j(\vec{x}\,)$ representa a j-ésima restrição por inequação, enquanto $h_k(\vec{x}\,) = 0$ representa a k-ésima restrição por igualdade. O termo $x_i^l \leq x_i \leq x_i^u$ representa a região de factibilidade de cada variável.

Métodos clássicos de otimização podem ser separados pela forma que tratam as restrições de um problema: (i) métodos genéricos que não tiram proveito da estrutura matemática da restrição, e (ii) métodos específicos que só se aplicam a tipos especiais de restrições. Os métodos genéricos englobam várias metodologias como a *função penalidade* que piora um resultado quando alguma restrição é violada.

4.1 Otimização Multiobjetivo

A otimização multiobjetivo consiste na otimização de vários critérios, ou funções, simultaneamente. Normalmente, essas funções apresentam uma relação conflitante; que traz uma complexidade extra. Por exemplo, seguindo novamente a definição de Deb (2000), um problema de otimização multiobjetivo com restrições poderia ser descrito da maneira a seguir:

$$
\begin{array}{lll}
\text{Minimizar} & f_m(\vec{x}\,) & m = 1, \ldots, M, \\
\text{Maximizar} & f_p(\vec{x}\,) & p = 1, \ldots, P, \\
\text{Sujeito a} & g_j(\vec{x}\,) \leq 0 & j = 1, \ldots, J, \\
 & h_k(\vec{x}\,) = 0 & k = 1, \ldots, K, \\
 & x_i^l \leq x_i \leq x_i^u & i = 1, \ldots, n.
\end{array}
$$

O termo $f_m(\vec{x}\,)$ representa todos os M objetivos a serem minimizados enquanto $f_p(\vec{x}\,)$ representa os P objetivos a serem maximizados.

Diferente de um problema mono-objetivo, em uma otimização multiobjetivo não é possível obter um mapeamento totalmente ordenado das soluções. No caso mono-objetivo, a própria ordenação de $f(\vec{x}\,)$ permite a definição do ótimo: o mínimo de $f(\vec{x}\,)$ para a minimização e o máximo de $f(\vec{x}\,)$ para a maximização. Na otimização multiobjetivo passamos a ter um conjunto de soluções ditas ótimas. Esse subconjunto é definido pelas soluções não dominadas, ou seja, não existem outras soluções que sejam melhores, em todos os critérios, que essas. Essa relação é definida como dominância de Pareto. O objetivo de uma otimização multiobjetivo é obter uma aproximação desse grupo de soluções chamado Pareto-Ótimo.

É possível transformar um problema multiobjetivo em mono-objetivo por meio de uma soma ponderada de todos os critérios em questão e então otimizar essa soma. É evidente que esse processo depende fortemente dos pesos empregados e obtém uma visão enviesada e incompleta das possíveis soluções ótimas.

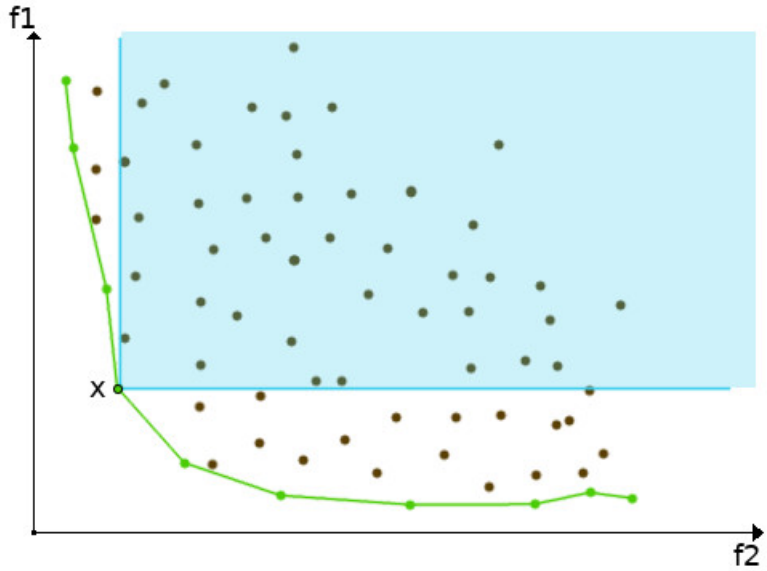
4.1.1 Dominância de Pareto

A dominância de Pareto é formalmente definida em Martins (2015a) como uma relação binária $>_p$ entre os elementos $u, v \in O$ de forma que $u >_p v$ se e somente se $u_i \geq v_i \forall i \in [M] \wedge \exists i \in [M] : ui > vi$ com $[M] = \{1, \ldots, M\}$. Onde M é o conjunto de todos os objetivos e O é o conjunto de todas soluções. A definição de $<_p$ é análoga mas trocando as relações $\geq$ por $\leq$ e $>$ por $<$.

Se dado um conjunto de soluções S, dispersos em um espaço de dois objetivos f_1 e f_2, fossem pegos somente os pontos não dominado segundo a dominância de Pareto $<_p$, obteria-se um subconjunto F, chamado fronteira Pareto-ótima. Na Figura 10 pode-se ver tal subconjunto em verde. As soluções em marrom são os pontos dominados pela fronteira. Existe uma outra fronteira análoga caso a relação $>_p$ tivesse sido empregada. O que determina se $>_p$ ou se $<_p$ será usado é o tipo de otimização, respectivamente, maximização ou minimização. A região azul é um detalhe da porção do espaço dominado pelo ponto arbitrário x, ou seja, todo e qualquer ponto que esteja dentro dessa região é dominando por x. Isso significa que todas as soluções dentro da região azul tem todos seus

objetivos piores que o ponto x. No entanto, vemos outros pontos marrons que não estão na região azul, que não são dominados por x em particular, mas são dominados por pelo menos um outro ponto da fronteira. Em suma, os pontos da fronteira são aqueles que não estão dentro da região de dominância de qualquer outro ponto do conjunto.

Figura 10: Fronteira de Pareto. Pontos verdes formam a fronteira. Pontos dominados em marrom. Região azul é a porção dominada pelo ponto x.



A noção da dominância de Pareto pode ser estendida para um espaço de objetivos de n dimensões. Como foi previamente dito, em um problema de otimização multiobjetivo não existe uma única solução ótima mas sim um conjunto de soluções ótimas. A relação de dominância de Pareto nos fornece uma maneira de formalizar uma definição para esse conjunto de soluções, as Pareto-ótimas.

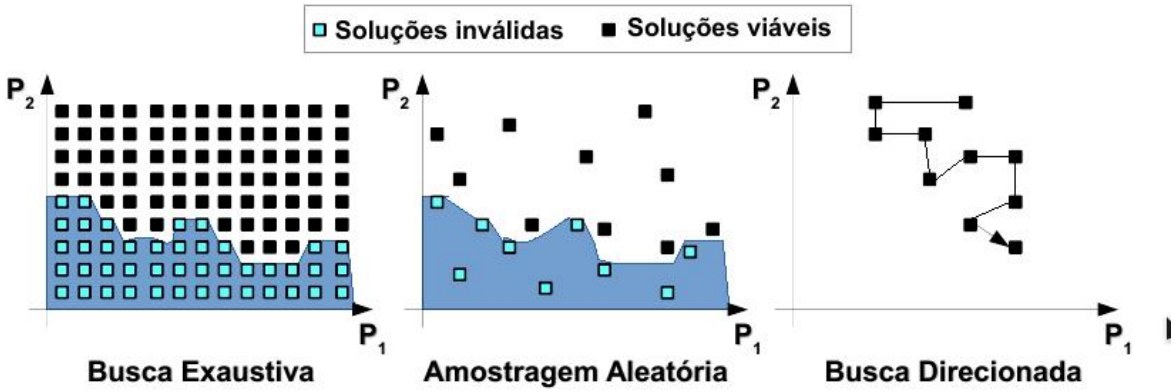
5 MÉTODOS DE EXPLORAÇÃO DE ESPAÇO DE *DESIGN*

A exploração de espaço de design, descrita na seção 2.2, pode ser vista como um problema de otimização. Muitas vezes se quer caminhar pelo espaço de design buscando as soluções que otimizem um ou mais critérios; provavelmente estipulados nos requisitos de projeto. O próprio ato de realizar a busca pode ser modelado como um problema de otimização. Como Martins (2015b) expõe, quer-se caminhar nos espaço de forma a otimizar tanto a cobertura de soluções quanto a qualidade das soluções. A escolha de qual método será utilizado para fazer a exploração é crucial para o sucesso da operação já que a busca exaustiva normalmente é inviável por conta da quantidade de soluções possíveis. Algoritmos evolutivos vem apresentando bons resultados quando utilizados como ferramenta de DSE (TEICH, 2012). Existem outras metodologias como algoritmo de Monte Carlo e a busca aleatória (MARTINS, 2015b).

5.1 Busca Aleatória

Esse é uma metodologia básica que tenta lidar com espaços de design muito extensos por meio de uma amostragem aleatória sucessiva. Nesse caso, uma amostragem aleatória pode ser analisada ou um processo iterativo pode ser empregado. Na Figura 11, vê-se um comparativo entre umas busca exaustiva, uma amostragem aleatória, e uma busca direcionada.

Figura 11: Esboço de três metodologias de busca.



Fonte: (MARTINS, 2015b, p. 31)

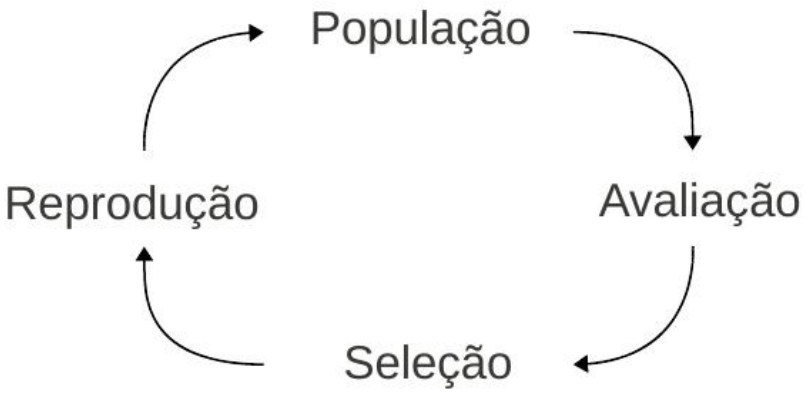
5.2 Algoritmos Evolutivos

Algoritmos evolutivos são meta-heurísticas de busca com aprendizado de máquina que mimetizam algumas das dinâmicas evolutivas encontradas na natureza. Essa ideia foi

inicialmente proposta na década de 1930, mas só pode ser realmente validada a partir da década de 1960 com o aumento dos recursos computacionais disponíveis. Desde então várias metodologias foram propostas, como os Algoritmos Genéticos, Estratégias Evolutivas, Programação Genética e Estimação de Densidade de Probabilidade (MARTINS, 2015a).

Um algoritmo evolutivo define operadores estocásticos que são aplicados às populações P, que modelam conjuntos de soluções, produzindo uma sequência de gerações $P(1), \ldots, P(t), \ldots$ em que $P(t+1) = \phi(P(t))$ e t é a geração atual e ϕ é a combinação de operadores usada. Esses operadores tentam simular os efeitos da mutação, reprodução e seleção natural já bem estudados no contexto da biologia (MARTINS, 2015a). A cada iteração o algoritmo evolutivo define as soluções mais promissoras que serão passadas para população seguinte.

Figura 12: Etapas de um Algorítimo Evolutivo



Fonte: (SOARES, 2014, p. 8)

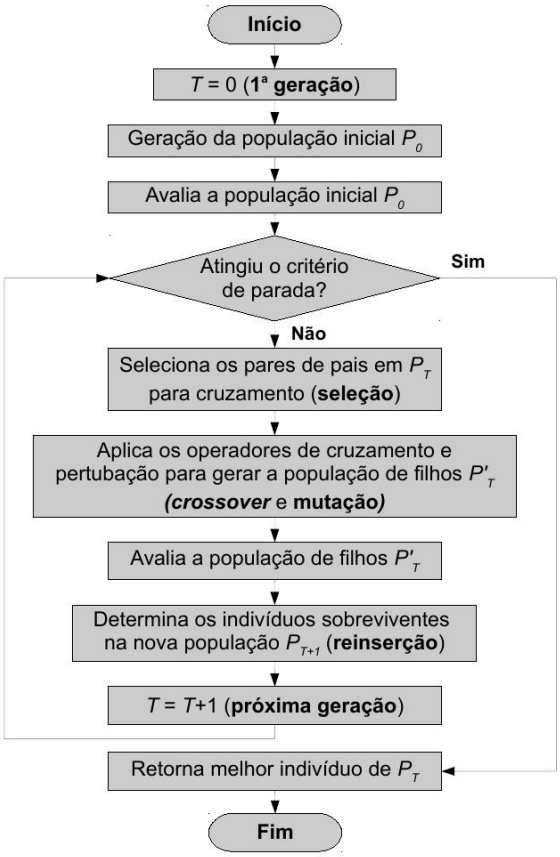
Na Figura 12, vê-se as etapas principais de um Algoritmo Evolutivo. Nessa ilustração, ϕ corresponde aos passos da avaliação, seleção e reprodução.

5.3 Algoritmos Genéticos

Os Algoritmos Genéticos são extensões da heurística proposta pelos algorítimos evolutivos. São uma estratégia de exploração estocástica mas direcionada e inspirada tanto na teoria da evolução como também no estudo da genética. Proposta por Holland (1975), os AGs modelam os conjuntos de soluções como populações de indivíduos. Já as soluções são modeladas como sequencias de genes. Além dos operadores existentes nos AE, os AGs introduzem a recombinação; uma forma de cruzamento entre os indivíduos a fim de encontrar uma nova solução mais promissora. A cada geração os indivíduos, ou soluções,

com menor aptidão segundo o critério de seleção são descartados; restando somente os melhores. Na Figura 13, vê-se um diagrama do fluxo de um suposto AG genérico.

Figura 13: Fluxo de um Algorítimo Genético simples



Fonte: (MARTINS, 2015b, p. 56)

Algoritmo 5.1: Algoritmo Genético canônico, adaptado de: (MARTINS, 2015a)

```
Requer: População inicial P(0), número máximo de gerações
    num_gerações, operadores de seleção S, de mutação M, de
    recombinação R e de inserção I.
inicio
enquanto t < num_gerações faça
    P_selecionados = S(P(t))
    P_filhos = R(P_selecionados)
    P_filhos = M(P_filhos)
    P(t+1) = I(P_filhos, P(t))
retorna P(t)
fim
```

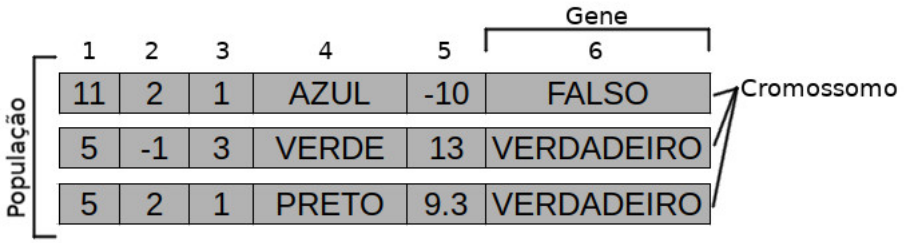
No Algoritmo 5.1, vê-se um algoritmo para um AG canônico. O operador de seleção (subseção 5.3.3) S seleciona os indivíduos que serão usados para a recombinação (subseção 5.3.4) e para a mutação (subseção 5.3.5).

O critério de parada de um AG depende da implementação, mas pode ir desde um número fixo de gerações à análise de convergência.

5.3.1 Genótipo

No paradigma dos AGs, as soluções são modeladas como cromossomos. Cada gene de um cromossomo representa uma variável ou característica dessa solução. Já os valores que cada gene pode assumir são chamados de alelos (SASTRY; GOLDBERG; KENDALL, 2005). Os cromossomos podem ser de tamanho fixo ou variável dependendo do problema. O caráter discreto ou contínuo de um gene depende diretamente da característica que modelam. É comum o uso de genes binários para representar a presença ou não de certo traço. Na Figura 14, pode-se ver um exemplo da representação proposta, onde o conjunto de cromossomos representa uma população.

Figura 14: Exemplo de uma representação cromossômica de uma população e seus indivíduos.



Os genes 1,2 e 3 são de representação inteira, ou seja, discreta. O gene 4 modela uma característica discreta não inteira, mas com valores possíveis finitos. O gene 5 é de caráter contínuo. Por fim, o gene 6 ilustra uma representação binária. A maneira que essas representações são traduzidas para um contexto computacional depende da implementação particular de cada algoritmo e aplicação.

5.3.2 População inicial

Um AG começa com uma população inicial aleatória, P_0 na Figura 13, de tamanho n. Todas as gerações seguintes devem manter esse mesmo tamanho. É importante garantir que os cromossomos aleatórios criados correspondam a soluções factíveis. É possível direcionar a criação desses cromossomos caso alguma informação seja conhecida *a priori*. A cada geração um número de descendentes sera gerado e existirão mais indivíduos que n. Para se retornar ao tamanho da população n, deverá haver algum critério de aptidão. A escolha do valor de n é muito importante. Caso seja pequeno demais, a convergência do algoritmo

será prematura e irá resultar em soluções triviais. Aumente o n demais e a execução não escalará, gastando computação desnecessariamente.

Após a criação da população inicial, cada indivíduo deve ser avaliado e receber seu valor de aptidão, ou *fitness*.

5.3.3 Seleção

A etapa de seleção tenta implementar o conceito da *sobrevivência do mais apto*. A ideia é favorecer a participação das melhores soluções nos próximos passos de do algoritmo: a recombinação e a mutação. Sastry, Goldberg e Kendall (2005) cita algumas maneiras de realizar essa etapa:

- **Roleta:** É um método estocástico onde a probabilidade de um indivíduo ser escolhido é proporcional a sua aptidão. Ou seja, quanto maior a aptidão mais provável é seleção desse indivíduo.
- **Seleção Estocástica:** Método estocástico puro. Todos os indivíduos possuem a mesma probabilidade de serem selecionados.
- **Torneio:** Grupos de indivíduos são formados aleatoriamente. A solução com mais aptidão em cada grupo é selecionada.

O conjunto de indivíduos criado pelo operador de seleção é chamado de *Pool de acasalamento*. É nesse conjunto que os operadores de recombinação e mutação serão aplicados.

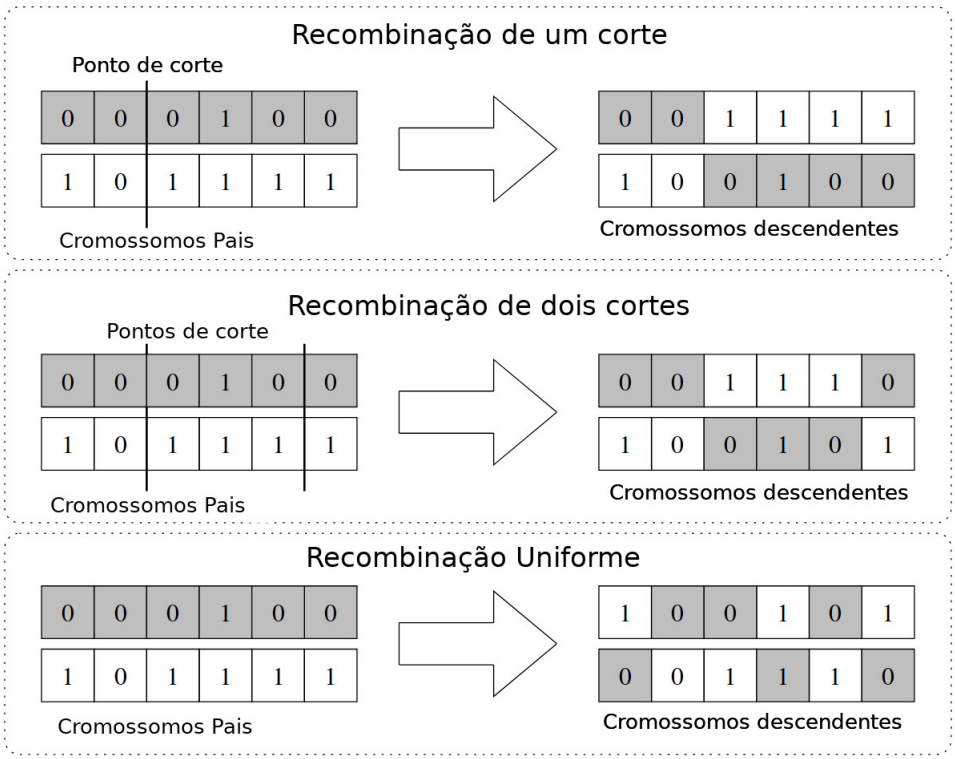
5.3.4 Recombinação

A recombinação, ou *crossover*, é o operador que simula o cruzamento dos indivíduos. Nela, uma troca de material genético é feita entre 2 ou mais pais a fim de gerar 2 ou mais descendentes que sejam potencialmente melhores que os anteriores. A implementação desse operador é muitas vezes dependente das características do problema. Sastry, Goldberg e Kendall (2005) apontam que implementações simples da recombinação não são escaláveis mas que estudos da área mostram implementações de operadores mais eficientes; como aqueles que usam *linkage learning* [1].

Normalmente os pais envolvidos na recombinação são escolhidos aleatoriamente do *Pool de acasalamento*. Dado um número aleatório r, se $r \leq p_c$, onde p_c é a probabilidade de recombinação, a troca de material é bem sucedida.

[1] Análise de agrupamento de genes a fim de identificar potenciais relações entre eles. Dessa maneira os operadores poderão fazer suas análises de forma mais educada e produzir, em geral, melhores descendentes.

Figura 15: Exemplos de recombinação: um corte, dois cortes e uniforme



Fonte: (SASTRY; GOLDBERG; KENDALL, 2005, p. 101)

Sastry, Goldberg e Kendall (2005) cita algumas implementações da recombinação:

- **Recombinação de k cortes:** Essa é a maneira mais simples e mais empregada da recombinação. Na Figura 15, pode-se ver uma recombinação de um e de dois cortes. k cortes são feitos aleatoriamente nos cromossomos dos pais. Com os segmentos formados, os descendentes são gerados. Sempre alternando o pai do segmento correspondente.

- **Recombinação uniforme:** Também ilustrado na Figura 15, consiste em cada alelo ser invertido entre os pais dada uma probabilidade p_e, que normalmente é de $0,5$.

5.3.5 Mutação

O operador da mutação tenta trazer diversidade a busca. Sem ele, as mesmas sequências de alelos seriam trocadas entre si eternamente. Caso um alelo não tenha sido sorteado na criação da população inicial, sem a mutação, ele nunca seria contemplado.

Após a etapa de recombinação cada gene do cromossomo tem uma probabilidade p_m de ter seu valor trocado por outro qualquer. No caso de um gene discreto, o novo alelo

pode seguir uma regra ou ser completamente aleatório. Já no caso de um gene continuo, um valor arbitrário pode ser subtraído ou somado aleatoriamente.

A mutação permite uma caminhada pela vizinhança das soluções pois faz mudanças pontuais. Esse operador normalmente tem um caráter secundário e por isso p_m tende a ser pequena, enquanto a recombinação tem p_c elevado, pois é o operador primário para geração de novos indivíduos.

5.3.6 Inserção

Após as etapas de seleção, recombinação e mutação existirá um conjunto de descendentes D_t além da população de pais P_t onde t é o número da geração atual. Como as populações P_t devem ter tamanho fixo n para simular os recursos finitos que os indivíduos estão disputando, P_t e D_t devem ser resumidos em uma única população P_{t+1} segundo algum critério. Novamente tem-se uma etapa de seleção que busca os indivíduos mais aptos. Alguns métodos de inserção incluem (SASTRY; GOLDBERG; KENDALL, 2005) (MARTINS, 2015b):

- **Substituição total:** Método mais simples que define P_{t+1} como D_t.
- **Substituição uniforme:** realiza roleta ou classificação para selecionar os melhores.
- **Elitismo** Seleciona os melhores de maneira pragmática segundo algum critério, sendo eles provenientes dos descendentes ou pais.

5.3.7 Algoritmos Genéticos Paralelos

Grande parte da computação em um AG está na etapa de avaliação das soluções. Muitas vezes, determinar a aptidão de um indivíduo envolve simulações complexas e demoradas. Por isso existem esforços para tornar esse gargalo menos significante. Uma das maneiras mais bem sucedidas de fazê-lo é tornando a avaliação dos indivíduos paralela. Como a avaliação de um indivíduo é independente dos demais, emprega-se a ideia do *dividir para conquistar*. É possível realizar avaliações concorrentemente, tirando o máximo de proveito das arquiteturas computacionais modernas *multi-core* e distribuídas.

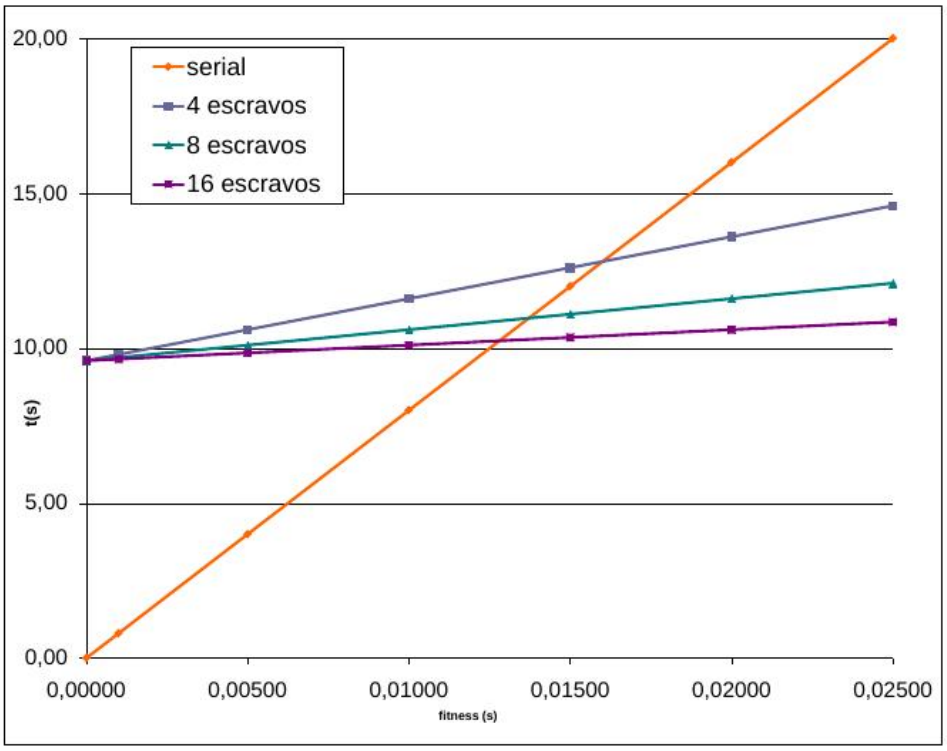
Cantu-Paz (1998) aponta três maneiras de se paralelizar um GA: (i) população única mestre-escravo, (ii) população única fina e (iii) multi-população grosseira. No contexto desse trabalho, somente a implementação *população única mestre-escravo* terá destaque.

A metodologia *população única mestre-escravo* ou *single-population master-slave* é a maneira mais simples de se implementar um GA paralelo. Nela, tem-se uma única população atual P_t, como num GA sequencial, mas a avaliação dessa população é separada em grupos e distribuída entre os processadores disponíveis. Cada grupo de indivíduos é passado a um processo escravo que realiza todas as avaliações e devolve os resultados

ao processo principal, que por sua vez, realiza as tarefas do GA (seleção, recombinação, mutação, etc). Quando o processo principal fica bloqueado esperando os resultados, tem-se uma paralelização síncrona. Essa metodologia é idêntica a um GA sequencial a não ser pela etapa de avaliação paralelizada.

Na Figura 16, Bonetti (2010) mostra o ganho de desempenho de uma implementação paralela de um GA em comparação a uma sequencial.

Figura 16: Comparativo entre a implementação de um GA mestre-escravo e um GA sequencial. Na abscissa têm-se tempo de cálculo da função de aptidão em segundos e na ordenada tem-se o tempo total de computação para uma geração.



Fonte: (BONETTI, 2010, p. 67)

5.4 Algoritmos Evolutivos Multi-objetivo

Os MOEAs, *Multiobjective Evolutionary Algorithms*, ou MOGA, *Multiobjective Genetic Algorithms* tentam solucionar um problema de otimização multiobjetivo conforme o descrito na seção 4.1 como um EA e um GA fariam em problema de otimização mono-objetivo, respectivamente. Como na otimização multiobjetivo, as soluções não podem ser ordenadas de forma absoluta segundo suas avaliações, a principal diferença entre um EA/GA e um MOEA/MOGA é a implementação do operador de seleção.

Alguns MOGA foram propostos usando a ideia de seleção usando dominância de Pareto (subseção 4.1.1), destacando-se os algoritmos NSGA-II (DEB et al., 2002) e o

SPEA-2 (ZITZLER; THIELE, 1999). Deb (1999) cita que para problemas com muitos objetivos, o NSGA-II passa a ter certa dificuldade de obter bons resultados pois converge prematuramente.

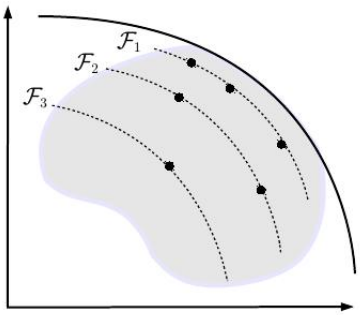
5.4.1 NSGA-II: Non-dominated Sorting Genetic Algorithm II

Proposto em Deb et al. (2002), o NSGA-II é uma evolução do algoritmo NSGA sugerido em Deb e Srinivas (1994). A base da seleção do NSGA-II consiste na ordenação de fronteiras de dominância.

Dada uma população P_t, os indivíduos não-dominados segundo a dominância de Pareto são separados formando a fronteira F_1. O processo é repetido ate que a população inteira seja consumida e k fronteiras de dominância existam, sendo F_1 a não dominada e F_k a mais dominada. Na Figura 18, pode-se ver um exemplo de fronteiras de dominância, onde a linha continua que domina a fronteira F_1 é chamada a fronteira Pareto-Ótima, ou seja, a melhor fronteira existente no problema. Os indivíduos contidos nas fronteiras menos dominadas são os mais aptos e serão selecionados primeiro.

O NSGA-II corrige dois pontos que eram criticados no NSGA original: (i) alto custo na ordenação de não dominância, (ii) falta de elitismo.

Figura 17: Fronteiras de dominância de uma população.

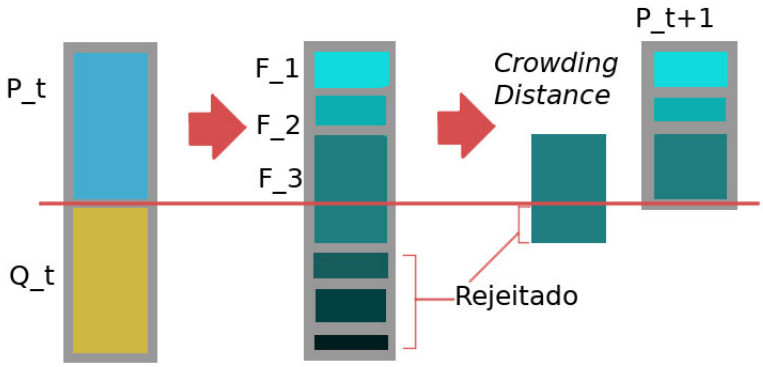


Fonte: (MARTINS, 2015a, p. 28)

Em uma abordagem ingênua dessa ordenação, poderia-se comparar um indivíduo com todos os outros da população em questão para determinar se ele é dominando. Isso requer $O(MN)$ computações onde N é o tamanho da população e M é o número de objetivos. Para realizar isso em todos os N indivíduos, seriam necessárias $O(MN^2)$ computações. Depois disso, o processo pode se repetir por, no pior caso, mais N vezes, resultando numa complexidade final de $O(MN^3)$ altamente não escalável. No entanto, o NSGA-II emprega um método intitulado *Fast Non-dominated Sorting*, que realiza $O(MN^2)$ computações. Para cada solução p analisada, um valor e um conjunto são calculados: (i) n_p, que é igual ao número de soluções que dominam p e (ii) S_p, que é o conjunto de soluções

dominadas por p. Depois disso, todas as soluções da primeira fronteira, ou seja, com $n_p = 0$ são separadas e se tornam o conjunto F_1. Para se determinar as fronteiras seguintes F_t, se percorre o conjunto S_p de cada indivíduo de F_{t-1} e subtraí-se 1 do seu n_p. Caso o n_p se iguale a 0, esse indivíduo é colocado em F_t.

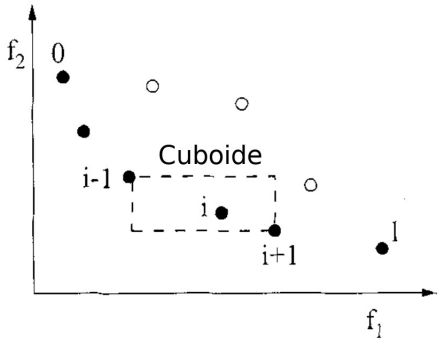
Figura 18: Funcionamento do operador de seleção no NSGA-II



Adaptado de: (DEB et al., 2002, p. 186)

Na Figura 18, vê-se uma ilustração do processo de seleção no NSGA-II. Dados a população de pais P_t e os descendentes Q_t, as fronteiras $F_1, F_2, \ldots, F_k$ são geradas por meio do *Fast Non-dominated Sorting*. Para formar a nova geração P_{t+1}, são selecionadas as fronteiras menos dominadas até que o tamanho N seja atingido (linha vermelha). As fronteiras F_1 e F_2 foram incluídas integralmente. No entanto, a inclusão de F_3 extrapola o tamanho da população. Por isso, somente uma parcela de F_3 será inclusa. Para determinar quais indivíduos de F_3 serão selecionados, o operador *Crowding distance* é utilizado. O objetivo desse operador é fazer o particionamento de fronteiras maximizando a diversidade das soluções. Ou seja, pegar as mais distintas entre si.

Figura 19: Cálculo do *Crowding distance*



Adaptado de: (DEB et al., 2002, p. 185)

O operador *Crowding distance* atribui um valor c_d a cada indivíduo da fronteira baseado no cuboide necessário para contê-lo tangenciando as soluções da sua vizinhança imediata. Na Figura 19 vê-se o cuboide referente a solução i. Como se trata de um espaço de objetivos de duas dimensões o cuboide é na verdade um área retangular. Dois de seus vértices são as soluções vizinhas $i - 1$ e $i + 1$. Essa área é o c_d da solução i. Nota-se que quanto mais agrupadas as soluções, ou seja mais parecidas entre si, menor o c_d. Por isso, depois de determinar o c_d de todas as soluções, retira-se o número de indivíduos necessário para atingir o tamanho esperado de P_t priorizando os de maior c_d.

Essencialmente a única diferença entre um MOGA e um GA é seu operador de seleção por conta das particularidades de um espaço de objetivos de duas ou mais dimensões. Por isso o resto do NSGA-II funciona de maneira similar a um GA tradicional. Os operadores de recombinação e mutação não mudam. Há uma outra particularidade no NSGA-II, pois só há seleção no momento de reinserção. A seleção para determinar os pais que serão recombinados e que sofrerão mutação é feita de maneira aleatória. Mas nada impediria a realização de um segundo *Fast Non-dominated Sorting* nessa etapa, e assim realizar uma seleção baseada na aptidão dos indivíduos.

Como as soluções menos dominadas são sempre selecionadas de maneira pragmática o elitismo é garantido.

6 AVALIAÇÃO DE DESEMPENHO

Para avaliar o desempenho de uma otimização mono-objetivo, basta analisar a convergência dos valores máximos ou mínimos das melhores soluções encontradas. Como os indivíduos são ordenáveis segundo o valor de seu único objetivo essa análise se torna trivial. No entanto, novamente vê-se que para o caso de uma otimização multiobjetivo essa tarefa torna-se bem mais complexa.

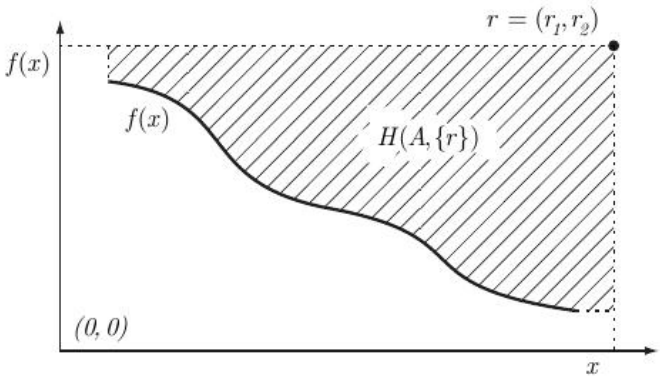
Vários indicadores foram propostos para avaliar a qualidade de uma população e por consequência o desempenho de um algoritmo evolutivo conforme a qualidade das populações evoluísse. Esses indicadores tentam modelar tanto a proximidade das populações da fronteira Pareto-Ótima como a diversidade de soluções da fronteira em si.

Existem métodos de otimização multiobjetivo que funcionam como um mono-objetivo pois tentam otimizar o próprio indicador de desempenho. Muitas vezes, essa abordagem soluciona o problema de convergência prematura para otimizações de muitos objetivos previamente citada (MARTINS, 2015a).

6.1 Hipervolume

O hipervolume $I_H(F, z_{ref})$ é um indicador que computa a união dos hipercubos formados pelas regiões dominadas por cada indivíduo da fronteira F em relação a um ponto z_{ref} referência. O hipervolume é útil pois consegue quantificar tanto a proximidade de F à fronteira Pareto-Ótima, como a distribuição dos indivíduos no espaço de objetivos (AZEVEDO; ARAUJO, 2011). A Figura 20 mostra o hipervolume $I_H(F, z_{ref})$ como a região hachurada.

Figura 20: Hipervolume $I_H(f(x), r)$, área grifada, do conjunto de soluções A descrito pela função $f(x)$ em relação ao ponto r

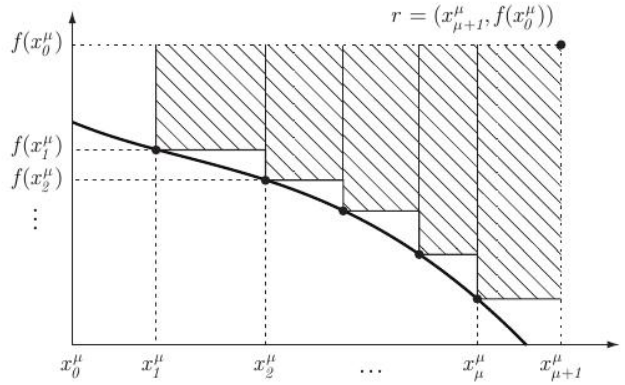


(AUGER et al., 2009)

Com o hipervolume consegue-se quantificar a convergência de um algoritmo de otimização multiobjetivo em relação a fronteira Pareto-Ótima. Dados um espaço de objetivos de duas dimensões, os pontos $(x_1^\mu, \ldots, x_\mu^\mu)$ da fronteira F e sendo $x_i^\mu \leq x_{i+1}^\mu$ pode-se definir I_H abaixo (AUGER et al., 2009). Na Figura 21, vê-se a contribuição de cada ponto $f(x_i^\mu)$ no cálculo do hipervolume.

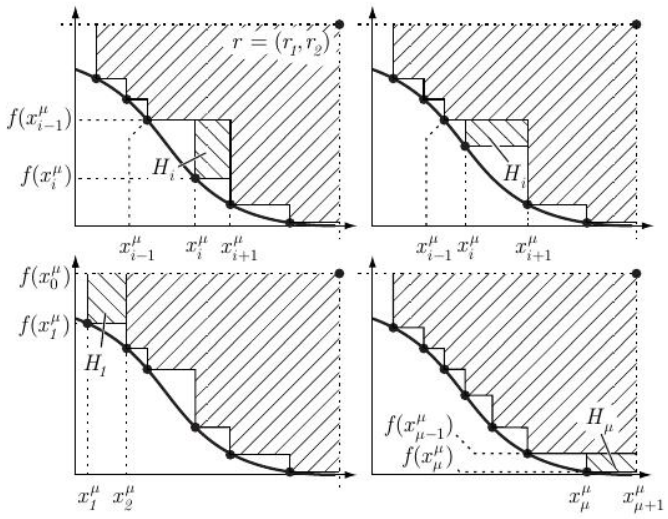
$$I_H^\mu((x_1^\mu, \ldots, x_\mu^\mu)) = \sum_{i=1}^{\mu} (x_{i+1}^\mu - x_i^\mu)(f(x_0^\mu) - f(x_i^\mu))$$

Figura 21: Exemplo de uma computação de Hipervolume $I_H(f(x), r)$.



(AUGER et al., 2009)

Figura 22: Contribuição H_i do ponto $(x_i^\mu, f(x_i^\mu))$ no valor do Hipervolume I_H.



(AUGER et al., 2009)

Na Figura 22 vê-se a contribuição H_i do ponto $(x_i^\mu, f(x_i^\mu))$ e como esse valor varia quando move-se o ponto x_i^μ no intervalo $[x_{i-1}^\mu, x_{i+1}^\mu]$. O hipervolume dos demais pontos não varia com o deslocamento de x_i^μ. Vê-se também H_i é maximizada quando x_i^μ esta equidistante de $[x_{i-1}^\mu$ e $x_{i+1}^\mu]$, favorecendo dispersões menos agrupadas.

6.2 Amostragem

Muitas vezes o espaço de objetivos e de *design* tomam dimensões muito elevadas, impossíveis de se percorrer inteiramente. Por isso, trabalha-se com amostras desses espaços invés do universo do problema. Essas amostras devem ser estimativas suficientemente precisas para o propósito em questão. Ou seja, as amostras devem refletir as características do universo com uma pequena margem de erro e com alto grau de confiança.

6.2.1 Amostragem aleatória simples

A amostragem aleatória simples é um método de seleção de unidades de um universo em que cada amostra distinta tem uma chance igual de ser escolhida. Essa tarefa pode ser feita com um gerador de números aleatórios de densidade uniforme ou com uma tabela de números aleatórios.

6.2.2 Bootstrap

O *Bootstraping* é uma técnica de re-amostragem usada para analisar a confiança de estimadores calculados a partir de amostras (HILLIS; BULL, 1993). Essa técnica usa amostragem aleatória simples com substituição para criar pseudo-amostras a partir da amostra original. Com essas pseudos amostras, o estimador é novamente calculado resultando em uma dispersão de estimadores. As pseudo amostras devem ser do mesmo tamanho que a amostra original.

Essa técnica é utilizada quando é laborioso ou impossível obter uma amostragem completa do um universo. O objetivo é extrair mais informações da amostra colhida e determinar a confiança dos estimadores calculados a partir dela.

7 *SOFT-CORE* NIOS II

O Nios II é a segunda iteração do *soft-core* distribuído atualmente pela Intel. Esse núcleo é voltado para aplicações embarcadas microprocessadas e implementadas em FPGA que não precisam de alto poder computacional. Por se tratar de um *soft-core*, ou seja, parametrizável e configurável, é uma excelente opção para o estudo proposto nesse trabalho. É possível variar diversas características da implementação do núcleo de maneira fácil por meio das ferramentas de *design* distribuídas também pela Intel.

7.1 Arquitetura

O Nios II se trata de um núcleo de processamento genérico de arquitetura RISC com as seguintes funcionalidades (INTEL, 2018a):

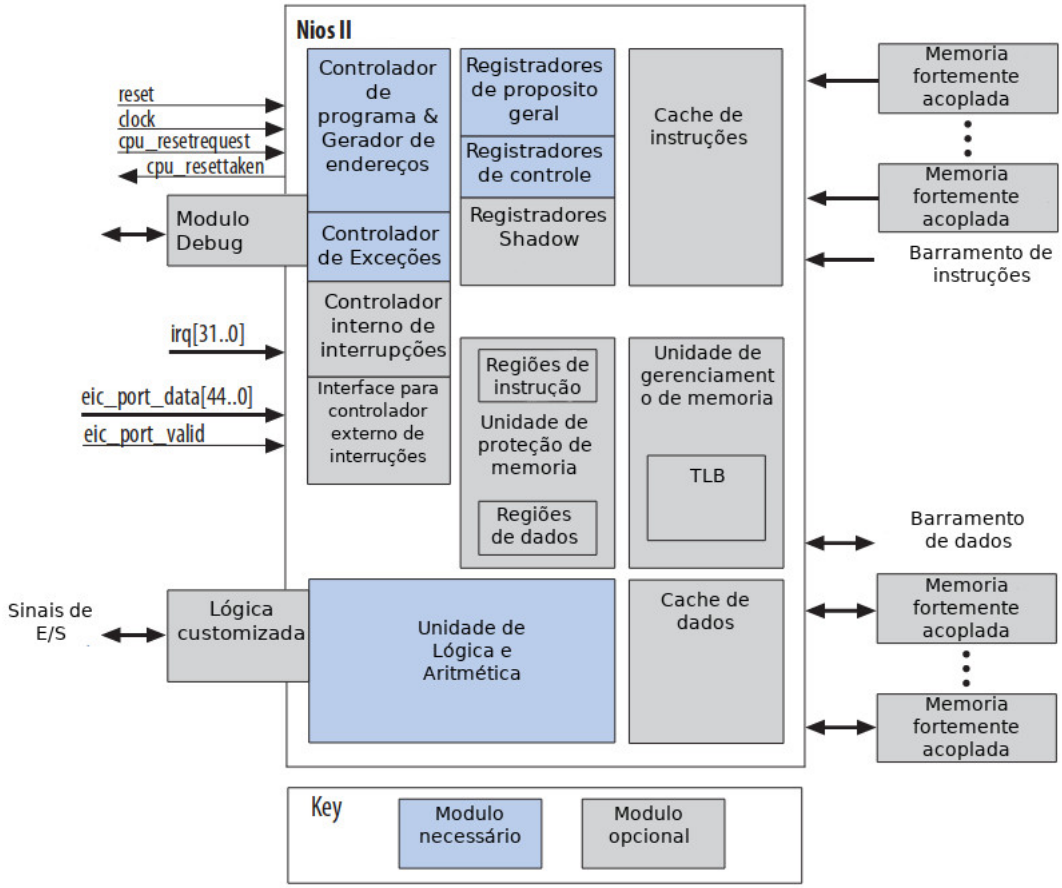
- Conjunto de instruções, caminho de dados e endereçamento de 32-bits
- 32 registradores de propósito geral
- Registradores *shadow* opcional
- 32 fontes de interrupção
- Interface para controlador de interrupções externo
- Multiplicação 32 x 32 e divisão com resultado de 32-bits com única instrução
- Unidade de ponto flutuante opcional
- Suporte para *debug* em hardware
- Ambiente de desenvolvimento de software baseado no GNU C/C++ *tool chain*

Na Figura 23 vê-se um diagrama de blocos do *soft-core* Nios II. Os blocos funcionais necessários estão grifados em azul enquanto os opcionais estão em cinza. É possível ver como o Nios II se trata de um núcleo flexível, que inclusive suporta instruções totalmente customizadas. Há três padrões de variáveis na arquitetura de um Nios II:

- **Calibrar funcionalidade:** Por exemplo, alterar o tamanho da *cache* para otimizar a arquitetura de uma aplicação em questão.
- **Incluir ou retirar funcionalidade:** Por exemplo, tirar o módulo de *debug* em ambiente de produção para reduzir custos.

- **Implementação Hardware/Software:** Determinar se certas instruções terão suporte de *hardware* ou serão emuladas em *software*.

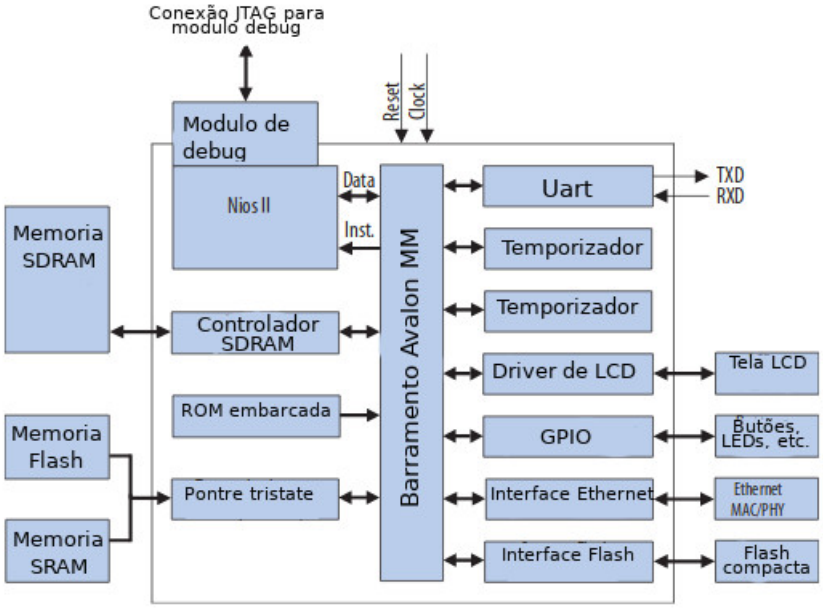
Figura 23: Diagrama de blocos de um Nios II. Blocos azuis são necessários, já os cinzas são opcionais



Adaptado de : Intel (2018a)

A arquiterua Nios utiliza um barramento proprietário chamado *Avalon-MM*, ou *Avalon Memory Mapped*, para se comunicar com qualquer dispositivo externo. Esses dispositivos podem ser desde memórias a interfaces de GPIO. Na Figura 24 vê-se uma ilustração de um sistema completo com base em um núcleo Nios II integrado por um barramento *Avalon-MM*.

Figura 24: Diagrama de blocos de um sistema que usa um Nios II.



Adaptado de : Intel (2018a)

O barramento Avalon MM usa o paradigma de mapeamento de memória, ou seja, cada dispositivo conectado a ele, que não seja o Nios II, recebe um intervalo de endereços de memória únicos que serão acessados de maneira transparente pelos desenvolvedores de software. Além disso, a temporização do barramento é gerenciada inteiramente por um controlador mestre que está no Nios II. Qualquer dispositivo que queira se integrar no sistema deve implementar o controlador escravo da especificação Avalon MM.

7.2 Nios II/f

Existem dois *presets* de Nios II disponíveis: (i) Nios II/e ou *economy* e (ii) Nios II/f ou *fast*. Como o nome já sugere, a versão Nios II/e tem seu foco em economia de recursos. Isso implica que suas opções de configuração são limitadas pela ausência de suporte estrutural. Por isso, somente o Nios II/f será tratado nesse trabalho.

Apesar de ser focado em alto desempenho, o Nios II/f permite uma grande customização possibilitando deixá-lo somente com seus recursos básicos; sem qualquer otimização. Algumas das particularidades de um Nios II/f (INTEL, 2018a):

- *Cache* de dados e/ou instruções.
- Suporte para MMU.
- Espaço de endereçamento de até 4GB.

- Suporte para memórias fortemente acopladas.
- *Pipeline* de 6 estágios.
- Predição dinâmica ou estática de condicionais.
- Implementação opcional em *hardware* de multiplicação, divisão e *shift*.

7.2.1 *Cache* de Instruções e de Dados

As implementações de *cache* do Nios II/f são independentes entre si. Ambos são implementados com mapeamento direto e com 32 bytes, ou 8 palavras, para cada linha da *cache*. Possuem um desempenho de uma leitura para cada ciclo de máquina. Podem ser virtualmente indexadas caso um MMU esteja presente.

É possível a implementação de transferência em rajadas para ambas as *caches* de maneira independente. Isso teoricamente aumenta a vazão de dados ao custo de um hardware mais complexo.

Para o *cache* de dados é possível especificar a tecnologia de sua implementação: RAM ou registradores. Também, somente para *cache* de dados, é possível determinar se a interface mestre irá suportar endereços de 31 ou 32 bits.

7.2.2 Implementações aritméticas

O Nios II permite as seguintes variações para a Unidade de Lógica Aritmética:

- **Ausente:** Multiplicação é emulada em software
- **Elementos Lógicos:** Implementado usando os recursos de elementos lógicos de uma FPGA
- **Multiplicadores embarcados** Usa multiplicadores dedicados disponíveis na FPGA
- **Bloco DSP:** Usa multiplicador de 32 bits disponível na FPGA
- **Divisão em Hardware:** Usa circuito SRT Radix-2 implementado com elementos lógicos.
- **Shift e Rotate:** Implementação em pipeline ou não

O *core* Nios II/f possui outras customizações além das supracitadas como, por exemplo, suporte a ECC, memórias fortemente acopladas, MMU, etc. Essas variações, no entanto, não estão no escopo desse trabalho e por isso não serão abordadas.

Quadro 1: Variações da ULA do Nios II/f

Variação na ULA	**Detalhes**	**Ciclos por instrução**
Elementos lógicos	ULA inclui multiplicador 32 x 4-bit	11
Multiplicador 32-bit	ULA inclui multiplicador 32 x 32-bit	1
Multiplicador 16-bit	ULA inclui 3 multiplicadores 16 x 16-bit	1
Multiplicador 16-bit	ULA inclui 4 multiplicadores 16 x 16-bit	2
Divisor em Hardware	ULA inclui circuito SRT Radix-2	35
Shift/Rotate em pipeline	Circuitaria dedicada	1
Shift/Rotate sem pipeline	Utiliza multiplicador presente	2-11

Fonte: Intel (2018a)

7.2.3 Predição de condicional

É possível escolher a implementação da otimização de predição de condicional:

- **Estática:** implementada usando um *offset* direcional para as condicionais.
- **Dinâmica:** implementada por meio de um histórico das condicionais em uma tabela de 2 bits.

7.3 Desenvolvimento de Hardware de um sistema Nios II

O *soft-core* Nios II é um IP proprietário e, apesar de ser customizável em vários aspectos, é código fechado. Não é possível desenvolver uma aplicação Nios II fora do ambiente de desenvolvimento da Intel. Esse ambiente, atualmente chamado de *Intel® Quartus® Prime Software*, possui uma série de ferramentas de *design* e compilação tanto de *hardware* como *software*. Além disso, esse ambiente é especialmente desenvolvido para trabalhar com FPGAs da família Altera.

No contexto do desenvolvimento de *hardware*, pode-se destacar as seguintes ferramentas:

- **Quartus Prime:** Ferramenta principal no desenvolvimento de *hardware* do ambiente Altera/Intel. Aqui define-se a raiz do projeto de *hardware*. Reúne todos os arquivos e códigos referentes ao *design*. Realiza mapeamento de pinos para as E/S do sistema. Compila o projeto e programa as placas FPGA.
- **Platform Designer:** Ferramenta de integração que permite a construção de um sistema usando uma abstração de blocos funcionais. IPs customizados podem ser incluídos desde que sigam os padrões de *design* estabelecidos na documentação. É a maneira mais simples de se construir um sistema Nios II. Após gerar o arquivo de descrição, basta importá-lo na ferramenta *Quartus Prime* para compilá-lo.

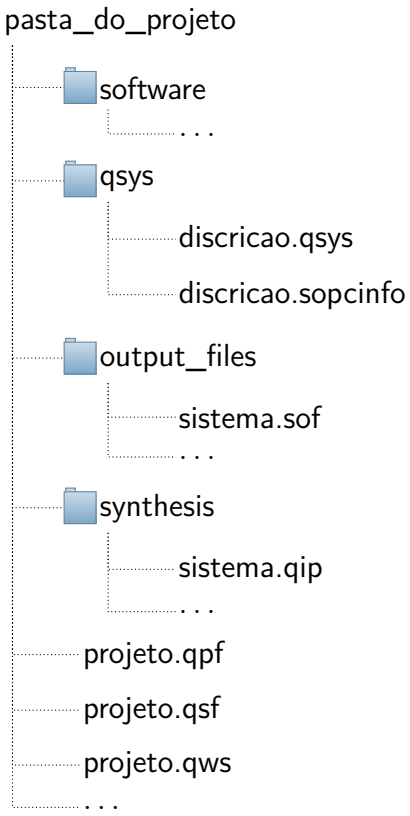
- **ModelSim:** Uma versão especial e licenciada do simulador de sistemas digitais desenvolvido pela empresa Mentor Graphics. Permite a simulação e avaliação de sistemas simples até altamente complexos como microprocessadores.

Todas as ferramentas possuem uma interface gráfica e uma interface de linha de comando. A interface gráfica é mais objetiva e tem uma curva de aprendizado mais acessível. Para realizar tarefas de automação ou *scripting*, a interface de linha de comando é imprescindível.

7.3.1 Estrutura de um projeto

Na Figura 25 vê-se uma estrutura de pastas e arquivos de um projeto Quartus Prime com sistema Nios II. Vale ressaltar que essa estrutura ocorre após uma compilação bem sucedida do projeto. Além disso, muitos arquivos e pastas foram omitidos para manter a clareza; por serem criados e manipulados inteiramente pelas ferramentas, são totalmente transparentes a um desenvolvedor.

Figura 25: Estrutura de um projeto Quartus Prime com sistema Nios II.



A pasta *Software* será tratada com mais detalhes na subseção 7.4.2. As pastas *qsys*, *synthesis* e seus arquivos são criados pela ferramenta *Platform Designer*. O arquivo .qsys é um XML que descreve o sistema a nível de blocos. Nele estão descritos quais blocos

estão presentes no sistema, como eles se conectam, qual mapeamento de memória, qual a configuração dos blocos, etc. Um Nios II é um dos blocos declarado nesse arquivo. Após a compilação do .qsys, a ferramenta gera os arquivos .sopcinfo e o .qip. O .qip é uma extensa lista de diretivas *tcl* que ditam como construir o sistema. Já o .sopcinfo é outro XML que descreve as funcionalidades disponíveis no sistema. Esse arquivo será usado na etapa de desenvolvimento de *software*. Os arquivos .qpf, .qsf e .qws são as definições do projeto de maneira geral. São criados pela ferramenta *Quartus Prime* e contém, por exemplo, quais códigos fazem parte do projeto e qual o modelo de FPGA que esta sendo usado. Após uma compilação do projeto, o arquivo .sof da pasta *output_files* é gerado. Esse arquivo é o *bitstream* que será descarregado na FPGA para configurar corretamente o *hardware* do projeto.

7.4 Desenvolvimento de *software* no Nios II

O desenvolvimento de *software* no Nios II usa um SBT, *Software Build Tools*, proprietário baseado no *tool chain* do GNU-GCC incluido no ambiente de desenvolvimento *Quartus Prime*. Esse SBT pode ser utilizado tanto via linha de comando como na IDE Eclipse.

Toda aplicação de Nios II consiste essencialmente de três elementos (INTEL, 2018b): (i) aplicação do usuário, (ii) bibliotecas opcionais do usuário e um (iii) *Board Suport Package* ou BSP. A aplicação do usuário e as bibliotecas opcionais são os elementos implementados pelos desenvolvedores da aplicação, enquanto o BSP é criado pelo SBT na inicialização do projeto. Um *makefile*, também gerado pelo SBT, é a raiz de toda a compilação do projeto.

O BSP é uma biblioteca especializada contendo *software* de suporte específico para um sistema elaborado com o Nios II e um ambiente Avalon MM. O BSP cria um *runtime* customizado para o sistema alvo baseando-se nas informações disponíveis no arquivo .sopcinfo (ver subseção 7.3.1). Alguns elementos inclusos no BSP (INTEL, 2018b):

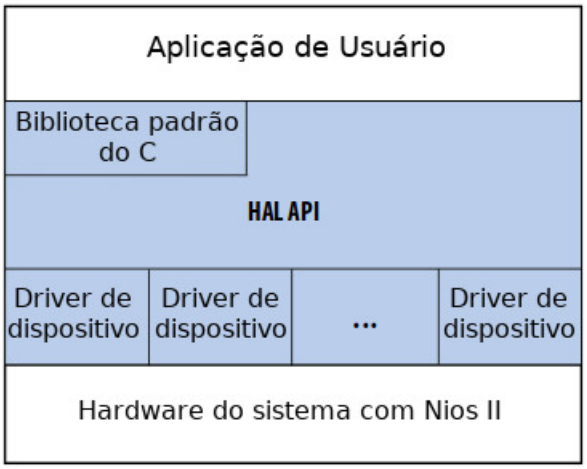
- *Hardware Abstraction Layer*
- Biblioteca opcional do padrão newlib C
- *Drivers*
- Pacotes opcionais
- Sistema operacional de tempo real opcional

7.4.1 HAL : *Hardware Abstraction Layer*

O HAL é um *runtime* leve que possibilita uma interface simples para as aplicação de *software* interagir com os dispositivos e o hardware do sistema. O HAL possui uma *API*[1] integrada a biblioteca padrão ANSI C e por isso permite o uso de funções conhecidas como *printf()*, *fopen()*, *fwrite()*, etc.

A criação de um BSP depende inteiramente do hardware disponível para a aplicação. Na Figura 26, vê-se como os elementos de um sistema baseado em HAL se organizam. O HAL disponibiliza *drivers* genéricos para periféricos comuns como temporizadores, controladores Ethernet, dispositivos de memória Flash, e etc.

Figura 26: Camadas de abstração de um sistema baseado no HAL



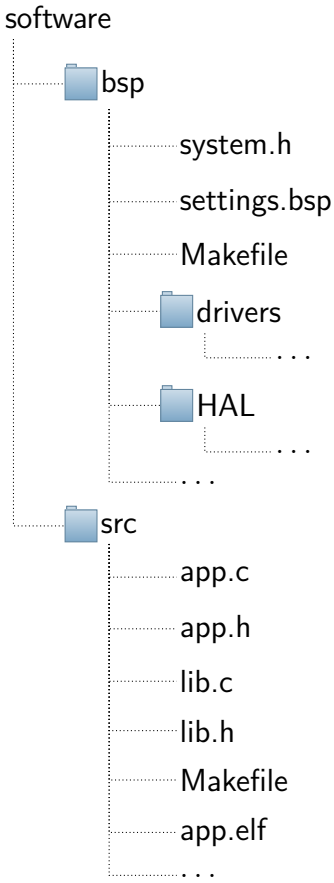
Adaptado de : Intel (2018b)

[1] Conjunto de diretrizes, rotinas e funções usados para determinar camadas de abstrações entre o *software*

7.4.2 Estrutura de arquivos de um Projeto de software

Na Figura 27 vê-se a estrutura de aquivos de um projeto de software para um sistema Nios II.

Figura 27: Estrutura de uma aplicação de software para um sistema Nios II.



A pasta *bsp* contém todos os arquivos correspondentes ao *Board Specific Package*. Eles são gerados a partir do arquivo *.sopcinfo*. Vê-se que uma pasta *drivers* e outra *HAL* são criadas contendo todos arquivos e bibliotecas necessários para que a aplicação funcione corretamente. O arquivo *system.h* contém uma descrição completa, em formato de *defines*, do hardware disponível. Nele estão presentes: nomes simbólicos de periféricos, endereços úteis, configurações de periféricos, etc. O arquivo *settings.bsp* é um XML de configurações do BSP a ser gerado. Essas informações incluem, por exemplo, se o *debug* está ativado, endereços dos periféricos, que tipo de sistema operacional estará disponível, etc. O *Makefile* da pasta *bsp* é responsável pela compilação de todo o BSP.

Na pasta *src* tem-se a aplicação propriamente dita. Não há uma forma específica de como essa pasta deve se organizar. A ferramenta de criação do Makefile da aplicação suporta arvores de projeto totalmente customizada. Nesse caso, a aplicação (*app.c* e *app.h*) e suas bibliotecas(*lib.c* e *lib.h*) são uma sugestão de como estruturar a aplicação. Após uma compilação bem sucedida do BSP e da aplicação por meio dos seus Makefiles

correspondentes, o arquivo *.elf* estará disponível. Esse arquivo corresponde ao binário da aplicação que será carregado na memória do Nios II por meio da interface USB utilizando uma das ferramentas do ambiente *Quartus*.

8 IMPLEMENTAÇÃO DA PROVA DE CONCEITO

O intuito desse trabalho é realizar uma prova de conceito utilizando Algoritmos Genéticos como ferramenta de Co-Projeto HW/SW. A ideia é, por meio da evolução ou otimização do *hardware*, obter núcleos de processamento melhores e mais eficientes segundo um algoritmo de referência. Um dos diferencias desse estudo é que todos os indivíduos contemplados ao longo do processo foram implementados e avaliados em uma FPGA a fim obter-se um resultado mais verossímil.

Koudil et al. (2007), Catania, Malgeri e Russo (1997) e Mishra et al. (2012) tentam resolver o problema do particionamento utilizando algum elemento de inteligência artificial. Já nesse trabalho será assumido um particionamento, assim como em Silva (2016), e partir dele serão otimizadas as características do *hardware* com base no *software* de referência.

A plataforma escolhida para realizar tal tarefa foi a da Intel por disponibilizar uma excelente quantidade de materiais e manuais online, além de ter uma comunidade um tanto ativa. Esse ambiente ainda dispõe do *soft-core* Nios II que mostrou-se uma excelente base para o estudo pois além de ser baseado numa arquitetura RISC/Harvard de 32-bits, e portanto condizente com outras CPUs atuais, apresenta uma boa quantidade de customização sem muitas dificuldades. Em seu 14º ano de vida e na sua 18º revisão, vê-se que esse *soft-core* ainda se mantém relevante na indústria e academia.

Esse estudo propõe uma otimização multiobjetivo dos núcleos Nios II. É recorrente encontrar na prática da engenharia problemas de otimização com metas conflitantes. Esses problemas são justamente os mais difíceis por não serem triviais e requererem ferramentas mais elaboradas. Por isso, irá-se otimizar os núcleos Nios II maximizando performance e minimizando o uso de unidades lógicas e unidades de memória. Ou seja, quer-se mais desempenho usando menos recursos. Mais uma vez vê-se um problema de otimização com critérios, em teoria, conflitantes.

O NSGA-II foi escolhido como o MOGA a ser implementado, por se tratar de um algoritmo maduro que já foi abordado em inúmeros estudos; e seu desempenho, particularidades e limitações são bem documentadas. De certa forma, o NSGA-II é tido como o ponto de partida quando a discussão é Algoritmos Genéticos multiobjetivo. A partir dos estudos a partir dele e de suas limitações, outros algoritmos puderam surgir e amadurecer.

8.1 Modelagem dos indivíduos

Inicialmente deve-se modelar um núcleo Nios II/f conforme o paradigma cromossômico dos Algoritmos Genéticos. Dessa forma cada núcleo poderá ser tratado como um

indivíduo e poderão ser aplicados os operadores necessários para a execução de um AG. A partir das configurações disponíveis em um processador Nios II descritas na seção 7.2 o cromossomo da Tabela 1 foi obtido:

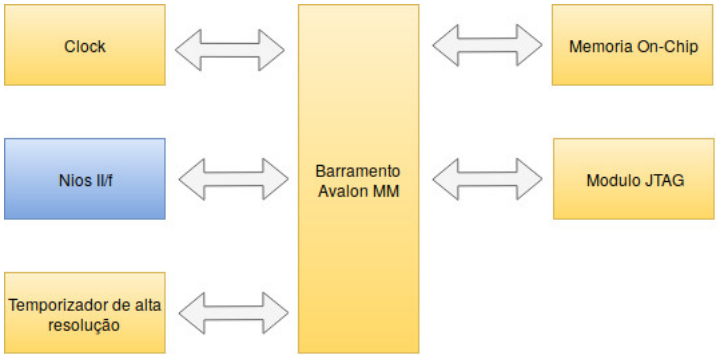
Tabela 1: Cromossomo correspondete a um Nios II

#	Gene	Alelos
1	Tamanho da *cache* de dados	0, 1024, 2048, 4096, 8192 bytes
2	Rajadas na *cache* de dados	Sim, Não
3	Tecnologia da *cache* de dados	Registradores, RAM
4	Tamanho da *cache* de instruções	0, 1024, 2048, 4096, 8192 bytes
5	Rajadas na *cache* de de instruções	Sim, Não
6	Endereçamento 32 bits	Sim, Não
7	Tipo de divisor	Nenhum, SRT 2
8	Implementação do multiplicador	0, 1, 2, 3
9	Implementação do *shift/rotate*	Sim, Não
10	Implementação da extensão multiplicador	Sim, Não
11	Numero de entradas no histórico das condicionais	256, 4096, 8192
12	Tipo de predição de condicionais	Dinâmico, Estático

Essas configurações foram definidas após análise do arquivo *.qsys* descrito na subseção 7.3.1. Algumas combinações de alelos são impossíveis, como por exemplo $gene_8 = 0$ e $gene_{10} = Sim$, e por isso são excluídas quando ocorrem durante os algoritmos. Vale ressaltar que os alelos apresentados na tabela foram renomeados para melhor entendimento; no arquivo *.qsys* outra terminologia é usada. Podemos ver que todos os genes propostos são discretos, além disso os genes $1, 2, 5, 6, 7, 9, 10$ e 12 são genes binários. O par de genes 8 e 10 definem qual tipo de implementação de multiplicador será utilizada. Após uma análise combinatória dos genes propostos, vê-se que o espaço de *design* desse problema de otimização possui 76800 indivíduos em potencial.

O arquivo *.qsys* define o sistema inteiro que seria implementado na FPGA; ilustrado na Figura 28. No entanto os genes escolhidos só contemplam o bloco correspondente ao Nios II que esta em azul. Os demais blocos são necessários para que o sistema possa existir e operar corretamente mas se mantém constantes ao longo de todos os indivíduos. Por isso, essa circuitaria auxiliar ficará implícita a partir daqui.

Figura 28: Um sistema completo.



O bloco *memória On-chip* corresponde a uma memória de programa e dados implementada inteiramente usando elementos da FPGA. O *módulo JTAG* permite o acesso as funcionalidades *debug* do Nios II. O *Barramento Avalon MM* proporciona a integração de todos os blocos. O bloco *temporizador* proporciona funcionalidades extras ao Nios II de captura de *timestamps* com resolução de ciclos de máquina.

8.2 Definição do problema de otimização

Com o modelo cromossômico e os objetivos da otimização em mãos podemos formalizar o problema segundo a especificação descrita na seção 4.1:

$$
\begin{aligned}
&\text{Minimizar} && f_{execução}(\vec{x}) \\
&\text{Minimizar} && f_{ALM}(\vec{x}) \\
&\text{Minimizar} && f_{memória}(\vec{x}) \\
&\text{Sujeito a} && x_1 \in \{0, 1024, 2048, 4096, 8192\} \\
& && x_2 \in \{Sim, Não\} \\
& && x_3 \in \{Registradores, RAM\} \\
& && x_4 \in \{0, 1024, 2048, 4096, 8192\} \\
& && x_5 \in \{Sim, Não\} \\
& && x_6 \in \{Sim, Não\} \\
& && x_7 \in \{Nenhum, SRT2\} \\
& && x_8 \in \{0, 1, 2, 3\} \\
& && x_9 \in \{Sim, Não\} \\
& && x_{10} \in \{Sim, Não\} \\
& && x_{11} \in \{256, 4096, 8192\} \\
& && x_{12} \in \{Dinâmico, Estático\} \\
&\text{Onde} && \vec{x} = (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12})
\end{aligned}
$$

O problema de otimização terá três objetivos: $f_{execução}$ modela a performance do indivíduo $\vec{x}$. Como $f_{execução}$, será medido em ciclos de máquina, minimizá-lo significa executar o programa mais rapidamente e por consequência maximizar a performance. f_{ALM} e $f_{memória}$ modelam o gasto de elementos lógicos e unidades de memória, respectivamente, para implementar o indivíduo $\vec{x}$. Cada cromossomo da Tabela 1 esta modelado em uma variável do vetor $\vec{x}$ como pode-se ver nas restrições do problema.

f_{ALM} e $f_{memória}$ são obtidos após a compilação dos indivíduos, já $f_{execução}$ é obtido após a avaliação, em FPGA, do indivíduo proposto.

8.3 Arquitetura da implementação

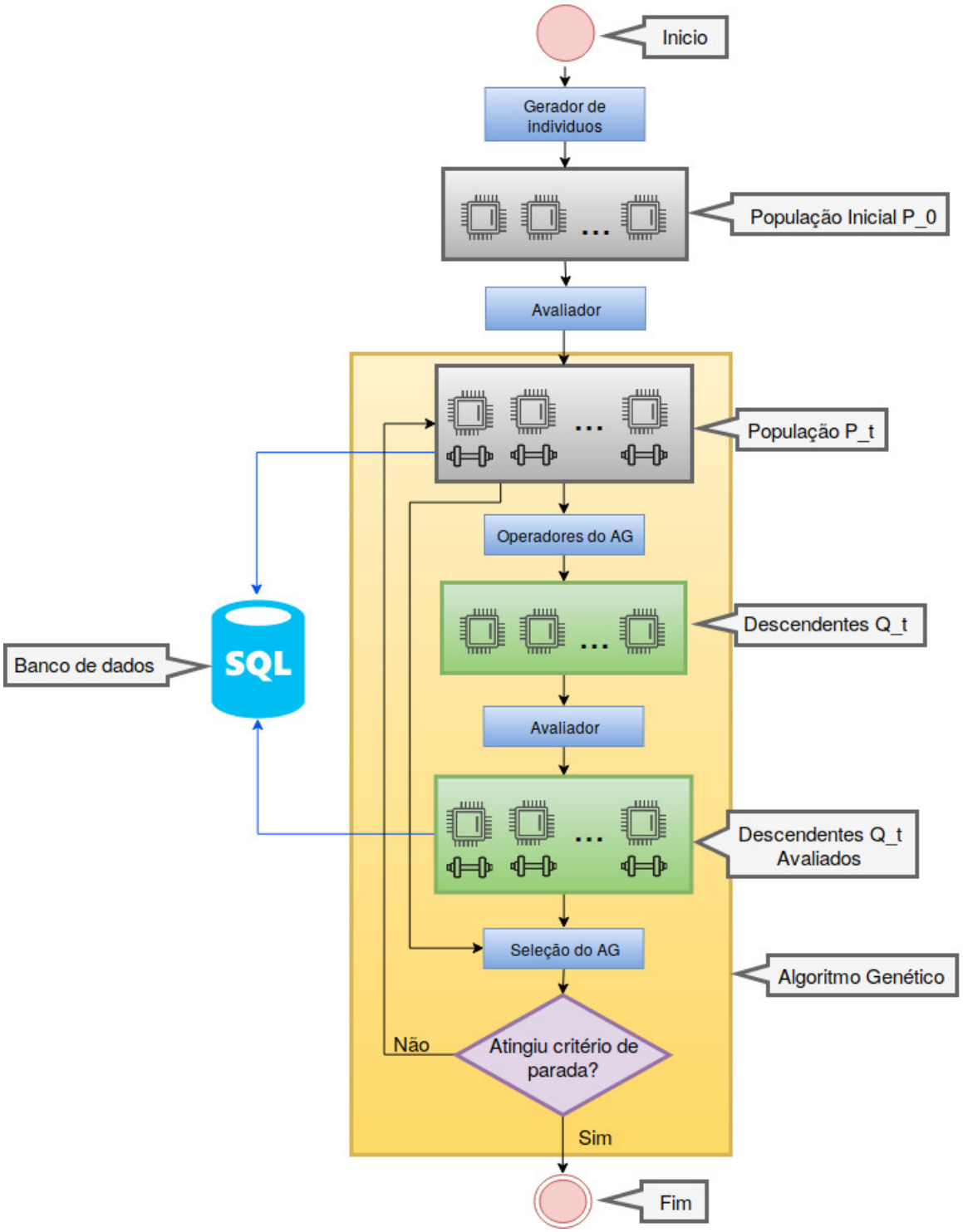
Como a implementação de uma versão do NSGA-II que interaja com o ambiente de desenvolvimento *Quartus Prime* é uma tarefa complexa e interdisciplinar, o problema foi divido em blocos independentes para ser mais facilmente organizado.

Na Figura 29, pode-se ver um diagrama de blocos da arquitetura e um esboço do fluxo do algoritmo proposto. Em azul se tem os blocos funcionais, ou seja, que realizam alguma computação; em cinza e verde se tem as coleções de dados. Cada bloco funcional é independente e foi idealizado para ser uma abstração de alto nível. As setas pretas modelam o fluxo normal do programa. Enquanto as setas azuis modelam o ato de armazenar aquele conjunto em um banco de dados quando estiverem disponíveis.

Essa estrutura poderia ser adaptada para funcionar com outros tipos de MOGAs ou até mesmo MOEAs. O que faz essa implantação se tratar de um NSGA-II é a forma com que os blocos *Operadores do AG* e *Seleção do AG* são desenvolvidos.

Todos os elementos em questão foram implementados em Python 3.6 em um ambiente Linux. O banco de dados foi feito usando SQLite 3. Alguns blocos utilizaram bibliotecas específicas que serão salientadas em suas seções. Vale ressaltar que o ambiente *Quartus Prime* para Linux possui uma usabilidade e comandos particulares descritos em Intel (2013).

Figura 29: Arquitetura em alto nível da implementação



O Algoritmo 8.1 é um pseudocódigo que tenta descrever o mesmo fluxo apresentado na Figura 29. O bloco *Gerador de indivíduos* é equivalente a função *gera_população_inicial()*. A população P(t) equivale a coleção P_t. O bloco *Avaliador* equivale a função *avalia()*. As setas em azul são equivalentes as chamadas de *salva_no_banco_de_dados()*. Por fim *Seleção do AG* é o mesmo que a função *seleciona()*. Enquanto P(t) e Q(t) modelam as

populações, A(t) e A2(t) modelam as coleções de avaliações.

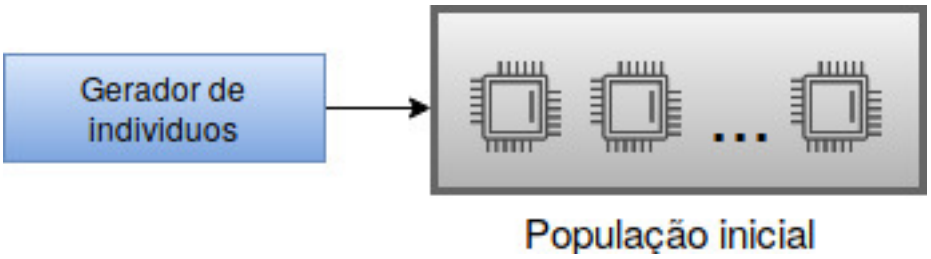
Algoritmo 8.1: Implementação da prova de conceito

```
1  inicio
2  t = 0
3  P(t) = gera_população_inicial()
4  A(t) = avalia(P(t))
5  enquanto não atingiu_criterio_de_parada faça
6      salva_no_banco_de_dados(P(t),A(t))
7      Q(t) = operadores_do_GA(P(t),A(t))
8      A2(t) = avalia(Q(t))
9      salva_no_banco_de_dados(Q(t),A2(t))
10     P(t+1), A_(t+1) = seleciona(P(t),A(t),Q(t),A2(t))
11 retorna P(t)
12 fim
```

8.4 Gerador de indivíduos

O *gerador de indivíduos* é um bloco responsável por criar uma população inicial P_0 formada por n indivíduos diferentes e válidos . A implementação dessa função usa o método *numpy.random.randint* da biblioteca computacional *Numpy* do Python para gerar números aleatórios.

Figura 30: Bloco *Gerador de Indivíduos*



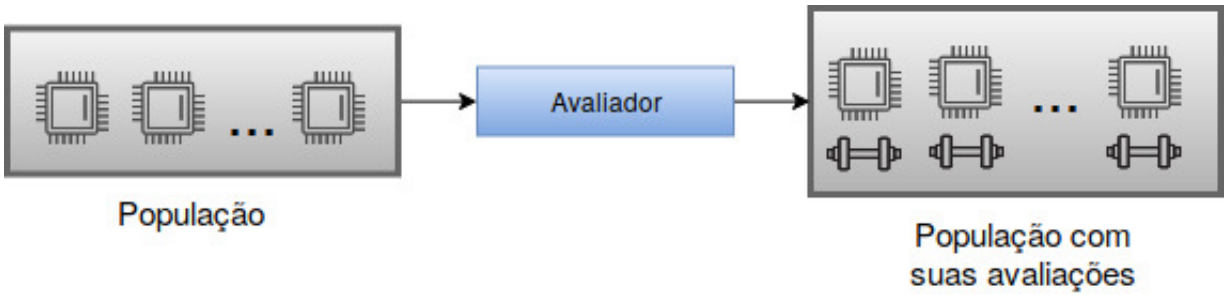
Algoritmo 8.2: Pseudocódigo da bloco gerador de indivíduos, equivalente a chamada de função gera_população_inicial()

```
1  Requer: tamanho da população n
2  inicio
3  população = []
4  enquanto população < n faça
5      cromossomo = []
6      para cada gene:
7          alelo = pega_alelo_aleatoriamente(gene)
8          cromossomo = adiciona(alelo, cromossomo)
9      se não_existe(cromossomo,população) e é_valido(cromossomo):
10         população = adiciona(cromossomo, população)
11 retorna população
12 fim
```

8.5 Avaliador

O bloco *Avaliador* é a função implementada que apresenta o maior grau de complexidade do trabalho. Ele é responsável por toda interação com o ambiente *Quartus Prime*. Além disso, possui uma arquitetura paralelizada a fim de melhorar seu desempenho. Seu objetivo é encapsular toda a complexidade envolvida na obtenção dos valores dos objetivos dos indivíduos de uma coleção de cromossomos.

Figura 31: Bloco *Avaliador*



Na Figura 31, vê-se a entrada e a saída do bloco em questão. O *Avaliador* recebe como entrada uma coleção de cromossomos, uma população. Sua saída, apesar de ser somente uma coleção de avaliações, foi ilustrada conforme a Figura 29, junto com a população.

Para se obter os valores dos objetivos, dados um cromossomo e um software de teste (*benchmark*), as seguintes ações devem ser tomadas:

1. Gerar o arquivo *.qsys* correspondente ao cromossomo
2. Compilar o *.qsys* para obter o *.qip*
3. Instanciar um novo projeto *Quartus Prime*
4. Adicionar o *.qip* ao projeto
5. Compilar o projeto
6. A partir do arquivo *.sopcinfo* gerar o BSP
7. Gerar o *Makefile* do software de referência
8. Compilar o BSP e o *software* de *benchmark*
9. Transferir configuração *.sof* para FPGA
10. Transferir *benchmark* para FPGA
11. Coletar performance e uso de recursos da FPGA

Essencialmente, o bloco *avaliador* é um *script* de automação dos processos de compilação e *benchmark* de cromossomos. Inicialmente será mostrado o processo para um único cromossomo. Depois a ideia será estendida para a arquitetura paralelizada que foi de fato empregada. Como na subseção 7.3.1, arquivos e diretórios auxiliares foram omitidos para simplificação.

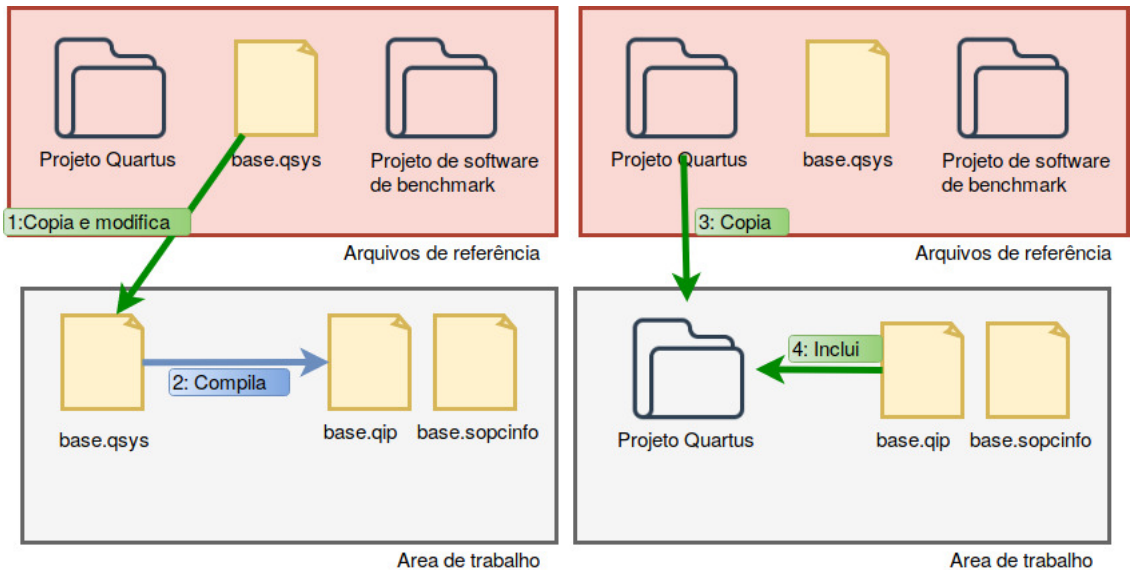
É possível criar e manipular projetos de HW e SW usando somente ferramentas de linha de comando do ambiente *Quartus*. No entanto, muitas delas utilizam arquivos contendo códigos *tcl* para determinar suas ações e o aprendizado dessa metodologia se apresentou muito laboriosa. Por isso, preferiu-se criar projetos de referência do *Quartus* e do *Platform Designer*, copiá-los quando necessário e modificá-los diretamente; seja com ferramentas do Python ou do Linux.

Os arquivos de referência incluem um projeto em branco *Quartus*, ou seja, uma estrutura de diretórios e arquivos específicos para que as ferramentas identifiquem um projeto válido. Além disso, incluem um projeto do *Platform Designer* que descreve um sistema como o ilustrado na Figura 28. Nesse sistema, as propriedades do Nios II possuem valores genéricos pois sempre serão trocados nos passos iniciais do *script* para os do cromossomo em questão.

Na Figura 32 vemos dois momentos que correspondem aos passos 1 a 4 listados anteriormente. Em verde, vemos ações realizadas puramente com Python. Já em azul, vemos as ações que usam tanto Python como alguma ferramenta do ambiente *Quartus Prime*. No primeiro momento (à esquerda) o projeto do *Platform Designer* é copiado e os

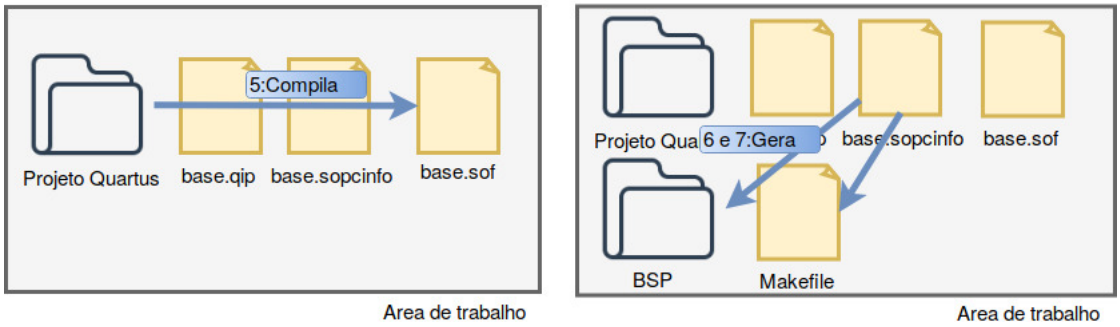
valores dos alelos do cromossomo são transferidos paras posições corretas do *.qsys*. Depois disso, o arquivo é compilado. No segundo momento (à direita), vemos o projeto do Quartus ser copiado e modificado para incluir o novo arquivo *.qip*.

Figura 32: Passos 1 a 4 do *Avaliador*



Já na Figura 33 vê-se outros dois momentos que correspondem aos passos 5 a 7. O primeiro momento (à esquerda) vemos a compilação do projeto de hardware que resulta no *bitstream* que será configurado na FPGA. No outro momento (à direita), o projeto de *software* é preparado. O arquivo *Makefile* aponta para os códigos fontes que estão no diretório de arquivos de referência.

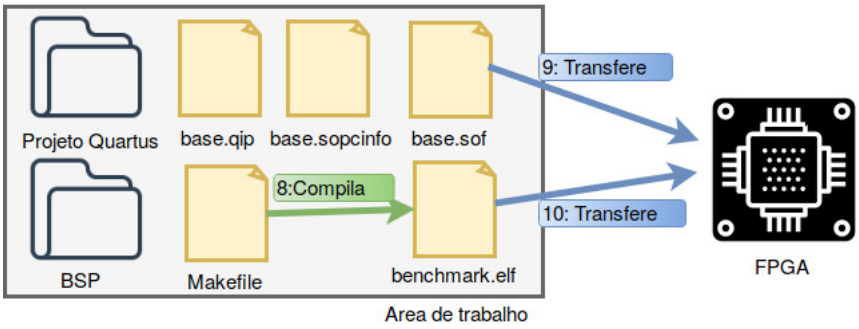
Figura 33: Passos 5 a 7 do *Avaliador*



Por fim, a Figura 34 ilustra o momento final correspondente aos passos 8 a 10. Nele o projeto de *software* é compilado e então, tanto o *bitstream* do *hardware* como o binário do *software* são transferidos para a FPGA. Vale ressaltar que isso só é possível se o dispositivo FPGA configurado no projeto *Quartus* estiver conectado via USB ao computador que executa o script. Depois disso, os valores dos objetivos são coletados. Os recursos de uso

da FPGA são obtidos de um arquivo de resumo gerado após a compilação do *hardware*. Já o desempenho da execução do *benchmark* são enviados pela FPGA via *debug* JTAG.

Figura 34: Passos 8 a 10 do *Avaliador*



8.5.1 Implementação Paralelizada

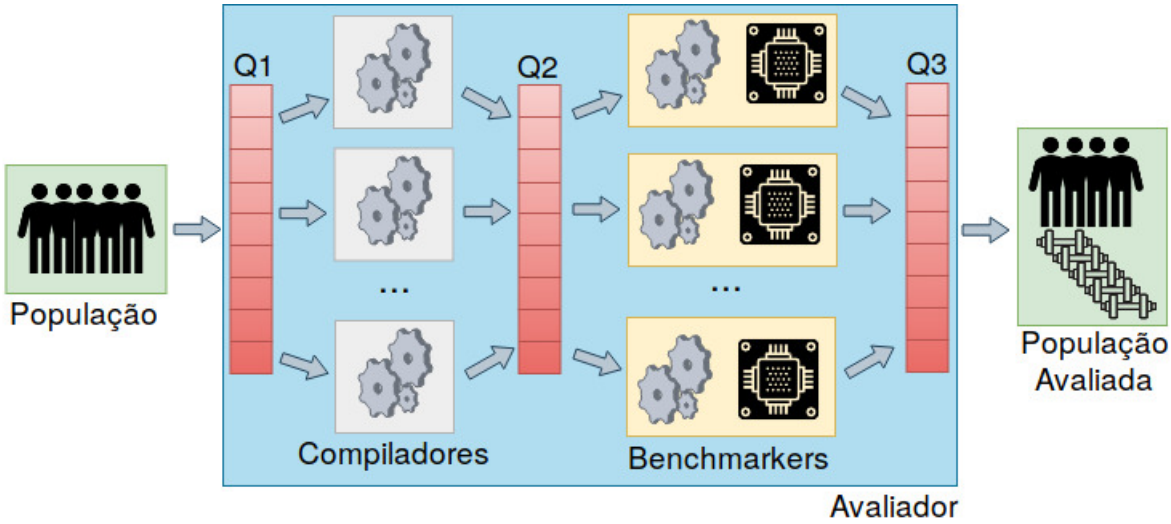
É fundamental para a paralelização de um programa determinar quais tarefas podem ser executadas concorrentemente e quais devem se manter sequenciais para manter a semântica esperada do programa. O processo descrito pelos 11 passos permite ser paralelizado em dois aspectos. (i) O processo de avaliar os indivíduos é independente, ou seja, avaliar x não influencia na avaliação de y. (ii) A compilação do indivíduo x é independente do *benchmark* de y. Isso é verdade para qualquer indivíduos x, y pertencentes a população sendo avaliada. Por conta dessa análise podemos separar os 11 passos descritos em dois processos independentes:

- **Compilador:** Responsável pelos passos 1 a 8. Essa tarefa deve, dado um cromossomo, preparar o *bitstream* e o binário correspondentes para serem transferidos a uma FPGA.

- **Benchmarker:** Responsável pelos passos 9 a 11. Essa tarefa deve, dado um *bitstream* e um binário, transferi-los para a FPGA e coletar os valores dos objetivos.

O primeiro aspecto paralelizável supracitado nos permite instanciar quantos processos forem possíveis sem qualquer prejuízo na semântica da aplicação. O segundo aspecto permite particionar o problema em dois tipos de processos. Vale ressaltar que o número de *Compiladores* não precisa ser igual ao número de *Benchmarkers*. Na Figura 35 vê-se a paralelização proposta.

Figura 35: Passos 8 a 10 do *Avaliador*



A comunicação entre os processos foi implementada segundo o paradigma *Produtor/Consumidor*. As filas Q1, Q2 e Q3 funcionam como as interfaces entre os processos envolvidos. Inicialmente, todos os cromossomos da população são inseridos na fila Q1. Enquanto houverem itens em Q1, os k *Compiladores* adquirem um cromossomo. Após compilar o cromossomo adquirido, o *Compilador* envia o cromossomo para a fila Q2. Enquanto houverem itens em Q2, os i *benchmarkers* adquirem um cromossomo. Após coletar os objetivos, as avaliações e o cromossomo são enviados a Q3. Após acabarem os cromossomos e toda a população estiver em Q3, a saída é estruturada e retornada.

É importante lembrar que uma paralelização só faz sentido enquanto instâncias independentes dos recursos necessários possam ser alocados para cada processo. Dessa forma, nenhum processo fica bloqueado aguardando um recurso compartilhado ser liberado. No caso dos *compiladores*, o recurso principal em questão são núcleos de processamento. Já no caso dos *Benchmarkers*, o recurso mais escasso são as próprias FPGAs. Na a arquitetura proposta só faz sentido instanciar i *benchmarkers* caso i FPGAs estejam disponíveis.

8.6 Operadores do AG

Como o AG em questão trata-se de um NSGA-II, os operadores de seleção de pais, a recombinação e a mutação foram implementados segundo os descritos em Deb et al. (2002).

- **Seleção de pais:** Seleção estocástica pura usando a população inteira.
- **Recombinação:** Recombinação de um corte com $p_c = 0.9$
- **Mutação:** Mutação simples com $p_m = 0.1$

8.7 Seleção do AG

Mais uma vez, como se trata da implementação do NSGA-II. A seleção usada foi o *Fast Nondominated Sorting* descrito na subseção 5.4.1. Esse método implementa a reinserção por elitismo.

8.8 Benchmarking

O objetivo $f_{execução}(\vec{x})$ quantifica o desempenho de um núcleo Nios II, $\vec{x}$, ao executar um algoritmo de referência, também chamado de *kernel* ou *benchmark*. Para colher essa informação, o desempenho deve ser calculado no próprio Nios II a fim de minimizar-se qualquer tipo de interferência. O desempenho em questão nada mais é que o tempo, ou quantidade de ciclos de máquina, gasto para se executar o *Kernel*. Por isso o *template* descrito no Algoritmo 8.3 foi elaborado.

Algoritmo 8.3: Pseudocódigo da base para *benchmark*

```
1  inicio
2  inicializa_timer()
3  limpa_cache_dados()
4  limpa_cache_instruções()
5  timestamp_inicio = colhe_valor_do_timer()
6  roda_kernel()
7  agora = colhe_valor_do_timer()
8  objetivo = agora - timestamp_inicio
9  envia_via_JTAG(objetivo)
10 reinicia_sistema()
11 fim
```

Como só há interesse em obter o numero de ciclos de máquina gastos, nenhum tipo de ferramenta de *profiling* foi necessário. Além disso ferramentas de *profiling* costumam injetar lógica dentro do programa analisado para viabilizar sua análise. O *overhead* gerado por esse código extra poderia empobrecer os resultados desnecessariamente. Para observar a evolução dos ciclos de máquina durante a execução do programa, o Nios II precisa de suporte de *hardware* de um temporizador. Isso justifica a presença do *Temporizador de alta resolução* presente na Figura 28.

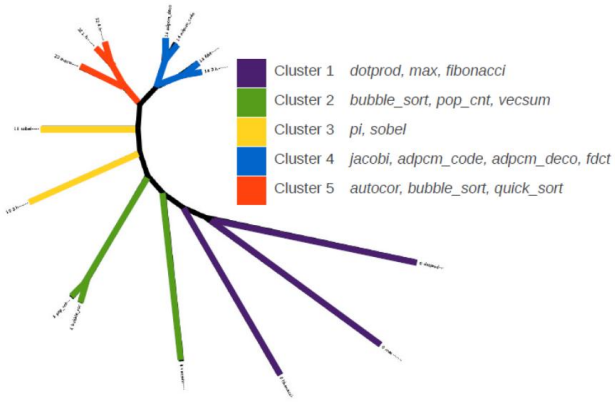
No Algoritmo 8.3, as linhas 2,3 e 4 preparam o sistema para o experimento, ou seja, zeram as *caches* e o temporizador. Na linha 5, o momento imediatamente antes do *benchmark* é armazenado. A linha 6 representa a execução do *kernel*. Na linha 7, o momento imediatamente depois do *benchmark* é colhido. Na linha 8 e 9 o objetivo é de fato calculado e enviado via JTAG para o computador executando o AG. Na linha 10 o

sistema é reiniciado para que uma nova avaliação do mesmo *kernel* seja realizada. Após 10 avaliações serem obtidas, uma média é realizada e $f_{execução}$ do *kernel* em questão é determinado. As linhas 2,3,4,5,6,9 e 10 são implementadas usando a API do HAL.

8.8.1 *Kernels* referência

Os *kernels* são pequenos algoritmos ou subproblemas comuns encontrados na academia e em problemas da engenharia. Muitas vezes quando um problema extenso ou complexo é reduzido em suas funções básicas, perfis de *kernels* conhecidos aparecem. Muitas vezes esses *kernels* se apresentam como o maior gargalo de uma aplicação. Como foi dito no início do capítulo, nesse trabalho será assumido um particionamento. Tal particionamento implica justamente que esses *kernels* deveriam ser executados em *software*. Em Silva (2016), é apresentada uma classificação de diversos *kernels* segundo seu grau de semelhança. Essa análise foi feita usando a ferramenta DAMICORE, um agrupador de códigos com perfis parecidos baseado em mineração de dados e identificação de padrões. Na Figura 36, pode-se ver o resultado dessa classificação.

Figura 36: Agrupamento de algoritmos com perfis próximos



Fonte: Silva (2016)

Foram escolhidos um *kernel* de cada agrupamento, ou *cluster*, apresentado a fim de amentar a variedade de perfis cobertos pelo estudo:

- **dotprod:** produto escalar de dois vetores (INSTRUMENTS, 2018c) (CODE, 2018a).
- **vecsum:** soma de todos elementos de um vetor (INSTRUMENTS, 2018b).
- **sobel:** filtro de detecção de bordas em uma imagem (INSTRUMENTS, 2018d).
- **adpcm_encode:** codificador de sinal segundo padrão adpcm (INSTRUMENTS, 2018a).

- **quick_sort:** ordenador de itens em um vetor (CODE, 2018b).

A implementação de cada *kernel* pode ser encontrada nas referências. Todos os algoritmos manipulam dados de tamanho 8,5Kb gerados aleatoriamente uma única vez. Dessa forma até o valor máximo proposto para as *caches* de 8192 bytes seriam preenchidos.

9 RESULTADOS

Como descrito no Capítulo 6, uma maneira de se determinar a qualidade de um conjunto de soluções, no contexto de uma otimização, é observar o hipervolume proveniente da fronteira de Pareto contida no mesmo. Com isso, é possível verificar a evolução de um MOGA, observando o crescimento do hipervolume a cada geração. Pode-se, então, determinar se as fronteiras estão caminhando na direção correta; ou seja, se estão realmente se aproximando da fronteira Pareto-Ótima.

A comparação entre hipervolumes foi a análise principal proposta nos resultados desse trabalho. No entanto, com isso só sabemos se o caminhar esta correto; não podemos garantir quando, ou mesmo se, chegaremos ao nosso destino sem de fato conhecê-lo. Conhecer nosso destino, significa conhecer a fronteira Pareto-Ótima e isso implica em uma busca exaustiva no espaço de design. Como realizar tal busca é inviável por conta do número de soluções existentes, é necessário estabelecer um estimador para a fronteira Pareto-Ótima do universo. A partir de uma amostragem aleatória de 1000 indivíduos, obteve-se uma fronteira de Pareto que seria o objetivo da otimização. Existe a possibilidade dessa amostra não ser representativa do universo que foi amostrado.

Para contornar essa questão e enriquecer o entendimento acerca do espaço de objetivos, a técnica de re-amostragem *Bootstrap* foi empregada. Essa técnica permite obter uma análise estatística da qualidade de uma amostra e da robustez dos indicadores calculados a partir dela. Nesse contexto, o hipervolume é considerado um indicador. São criadas pseudo-amostras sacando indivíduos aleatoriamente da amostra original, com repetição, e a partir delas um novo hipervolume é calculado. Cada pseudo amostra possui o mesmo tamanho que a amostra original. A partir da dispersão de hipervolumes obtida, uma análise estatística pode ser realizada. Em todos os momentos que o *Bootstrap* foi usado nesse trabalho, 100000 pseudo amostras foram analisadas.

Inicialmente se realizou otimizações de dois objetivos, $f_{execução}$ e f_{ALM}, para verificar o funcionamento de todas as partes da aplicação; uma vez que analisar os resultados em um espaço de duas dimensões é mais simples do que em um de três. Nesse caso foram efetuadas 5 execuções do NSGA-II para o *kernel adpcm*. Após esse experimento, passou-se para a otimização de três objetivos, $f_{execução}$, f_{ALM} e $f_{memória}$. Foram outras 5 execuções do NSGA-II, mas dessa vez, cada execução teve um *kernel* de referência diferente. Vale ressaltar que a intenção era realizar pelo menos 5 execuções do NSGA-II para cada *kernel* tanto para otimização de 2 objetivos como para a de 3 objetivos. No entanto, isso não foi possível por conta do tempo limitado. Por isso os experimentos de 2 objetivos foram pautados em evidenciar um melhor entendimento do NSGA-II, enquanto os experimentos de 3 objetivos tinham a intenção de mostrar os resultados da otimização propriamente

dita.

9.1 Ambiente de teste

Todos os experimentos foram realizados num computador com as seguintes características de Hardware e Software:

- i7-7700HQ
- 16GB RAM
- Kubuntu 18.04
- Linux kernel 4.15.38
- Quartus Prime 18
- Python 3.6

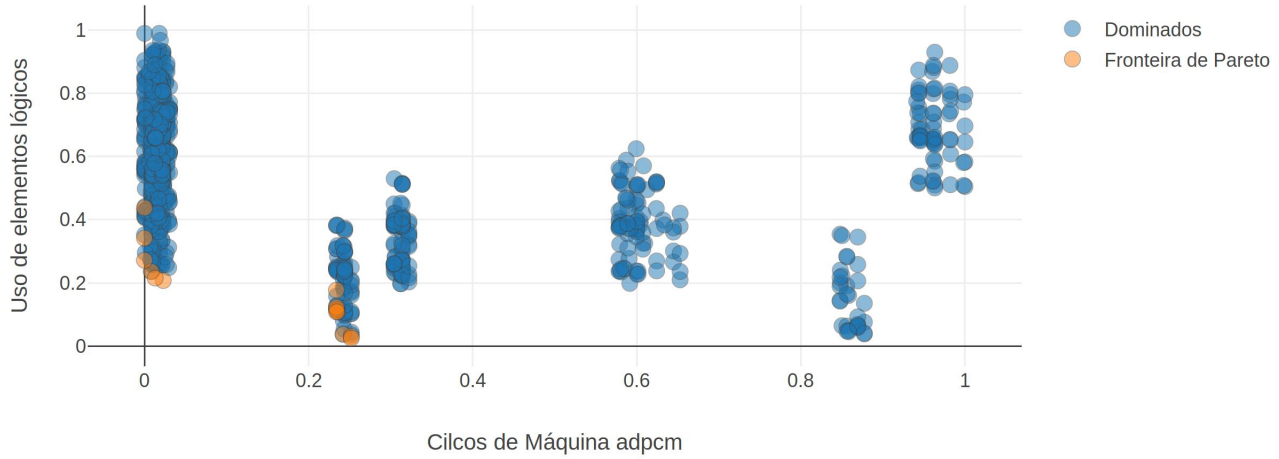
Foram utilizadas duas placas Terasic DE10-Nano com o SoC Cyclone V 5CSEBA6U23I7NDK que permitiram o paralelismo dos *benchmarks*.

9.2 Otimização de dois objetivos: $f_{execução}$ e f_{ALM}

9.2.1 Estimador da fronteira Pareto-Ótima para 2 objetivos

A análise do espaço de objetivos é muito importante para termos uma noção de como as soluções estão dispersas. Esse experimento será chamado de *Mil Cores* daqui em diante. Na Figura 37 pode-se ver o espaço de objetivos para o kernel *adpcm*. Podemos verificar que a fronteira de Pareto, em laranja, apresenta-se como esperado.

Figura 37: Espaço de objetivos para otimização $f_{execução}$ e f_{ALM} com kernel *adpcm*. Valores normalizados.

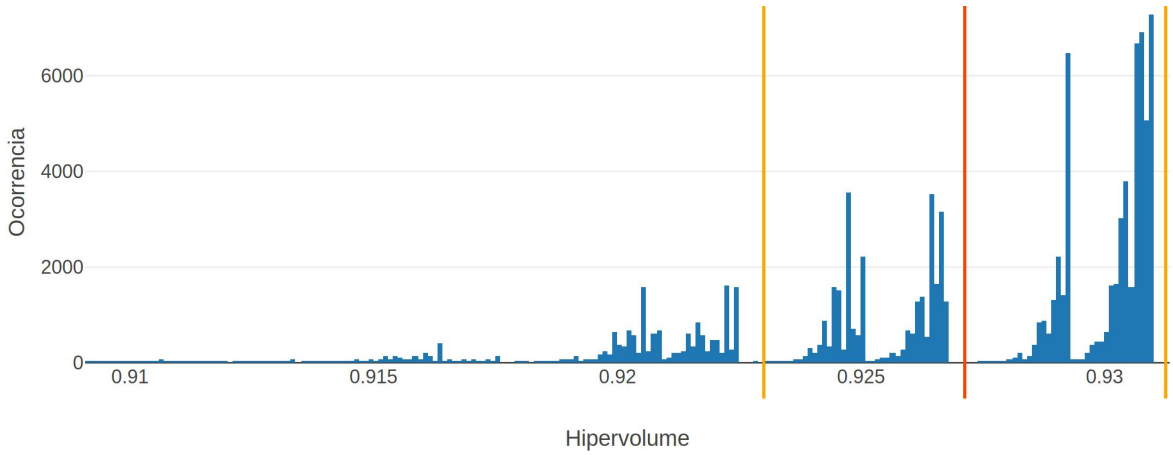


Gostaria-se que a fronteira de Pareto do experimento *Mil Cores* se aproximasse ao máximo da fronteira Pareto-ótima do problema. Novamente, só é possível saber se a fronteira de Pareto obtida é expressiva, se a Pareto-Ótima fosse conhecida.

Vale ressaltar que os indivíduos obtidos se agruparam em 6 regiões. Isso pode sugerir que, para o kernel *adpcm* existem poucos genes que realmente impactam o seu desempenho. Os demais genes causariam um impacto marginal no desempenho.

O experimento *Mil Cores* tomou cerca de 31h para ser completado. Isso implica numa média de 1,86 minuto para avaliar cada cromossomo. Para avaliar todo o espaço de *design* estimado em pouco mais de 76000 cromossomos, 2356 horas ou $98,2$ dias, seriam necessários. Isso confirma que uma busca exaustiva seria impraticável a não ser que um maior poder computacional estivesse disponível.

Figura 38: Histograma dos valores de hipervolume da fronteira de Pareto obtidos após 100000 iterações de Boostrap sobre da amostra obtida no experimento *Mil Cores* para o *kernel adpcm*. Valores normalizados.



Na Figura 38 vê-se um histograma da dispersão obtida a partir do *boostrapping* da amostra gerada em *Mil Cores*. A linha vermelha marca a média de todas as ocorrências. Enquanto os limites em laranja mostram o desvio padrão. No Quadro 2 pode-se ver os dados estatísticos referentes a dispersão. Portanto, para esse experimento esse seria a estimativa da fronteira Pareto-Ótima.

Quadro 2: Dados estatísticos da dispersão do hipervolume apresentada na Figura 38

Experimento	**Máximo**	**Mínimo**	**Média**	**Desvio Padrão**
Mil Cores *adpcm*	0.931	0.875	0.927	0.004

9.2.2 Otimização com NSGA-II

Foram executadas 5 vezes otimização de $f_{execução}$ e f_{ALM} para o kernel *adpcm*. Cada execução processou 20 gerações de 50 indivíduos; um total de 1000 cores por execução. A evolução do hipervolume das fronteiras de Pareto contidas em cada geração para as 5 execuções podem ser vistas na Figura 39 e Figura 40 (detalhe).

Figura 39: Evolução do hipervolume das fronteiras de Pareto para 5 execuções. Valores normalizados.

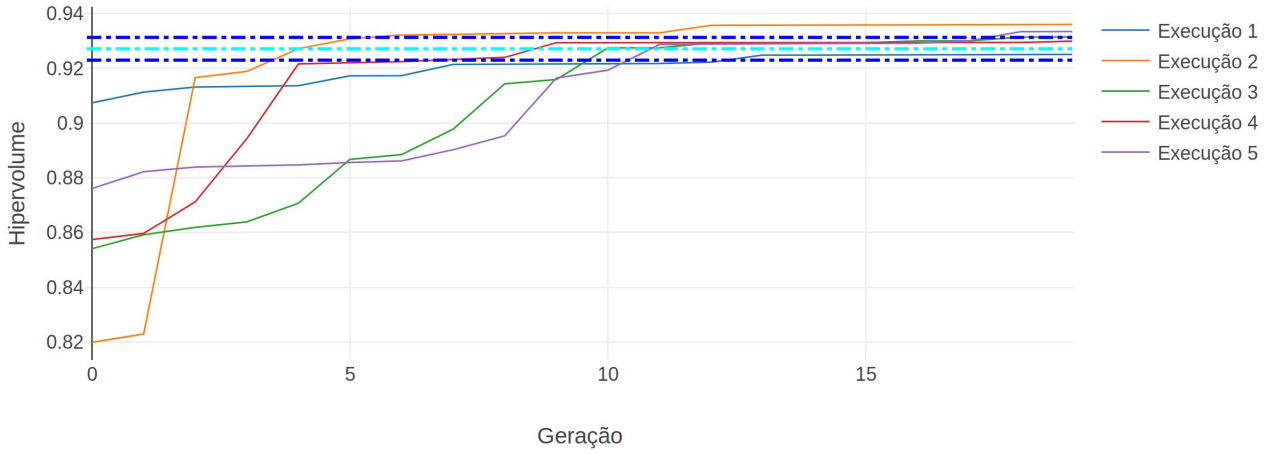
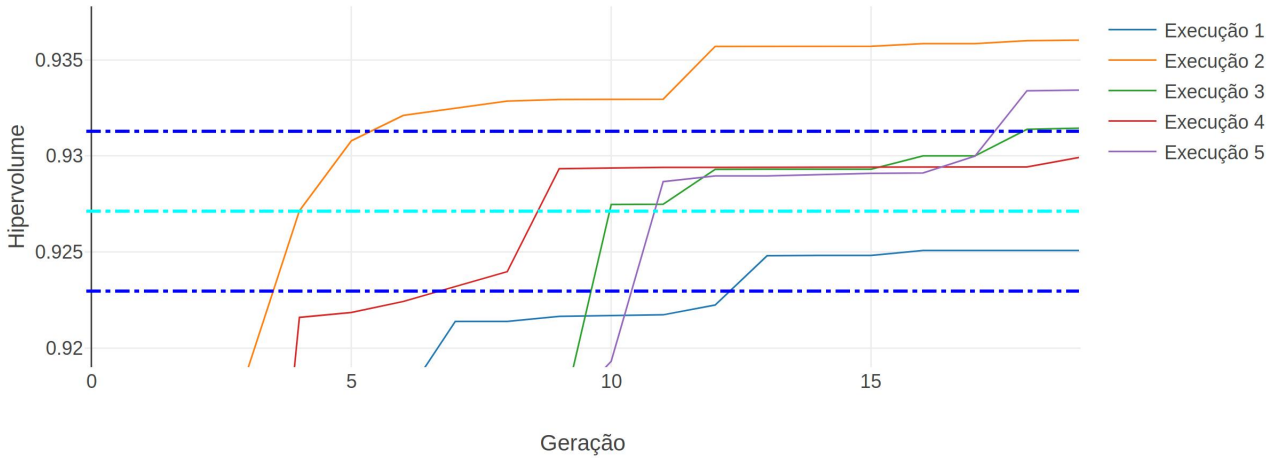


Figura 40: Evolução do hipervolume das fronteiras de Pareto para 5 execuções (detalhe). Valores normalizados.



As linhas pontinhadas azul clara e azul correspondem a média e aos limites do desvio padrão, respectivamente, da fronteira de Pareto em *Mil Cores*. Vê-se que em todos os casos a otimização convergiu para um valor dentro do intervalo definido pela media e desvio padrão estabelecidos como base. Além disso, para 2 execuções o hipervolume atingiu valores maiores que a média de referência; e para outras 2 execuções o hipervolume

foi maior que o limite superior definido pelo desvio padrão. Isso mostra que o AG teve resultados substancialmente melhores que o estimador de fronteira Pareto-Ótima.

Nas figuras 41 e 42, vemos um comparativo entre as fronteiras obtidas pelo NSGA-II e o estimador da fronteira Pareto-Ótima. Nota-se que o NSGA-II chegou à fronteiras de boa qualidade que dominam boa parte dos pontos da fronteira de *Mil Cores*. No entanto, podemos ver que os resultados estão bem agrupados. Isso é negativo pois faz parte de um problema de otimização multiobjetivo maximizar a diversidade de soluções. Apesar desse agrupamento sistêmico, na Figura 42 vê-se que um região de soluções só foi encontrada no NSGA.

Figura 41: Fronteira de Pareto: Mil cores vs. NSGA Execução 1. Valores normalizados.

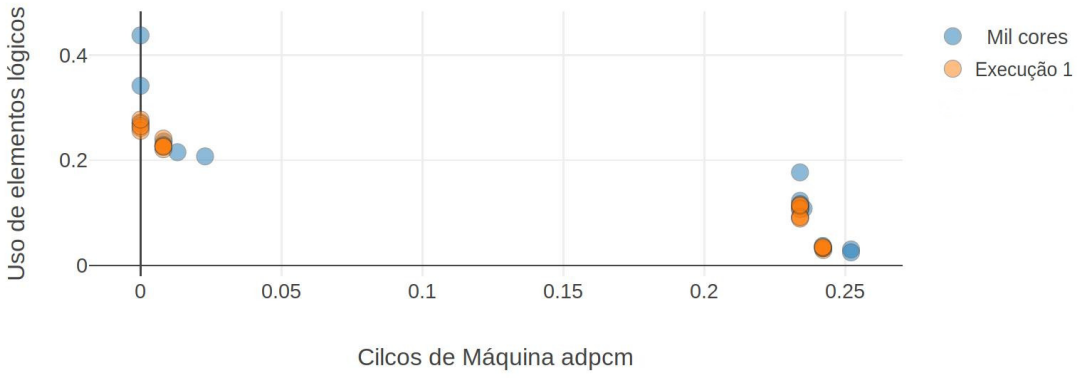
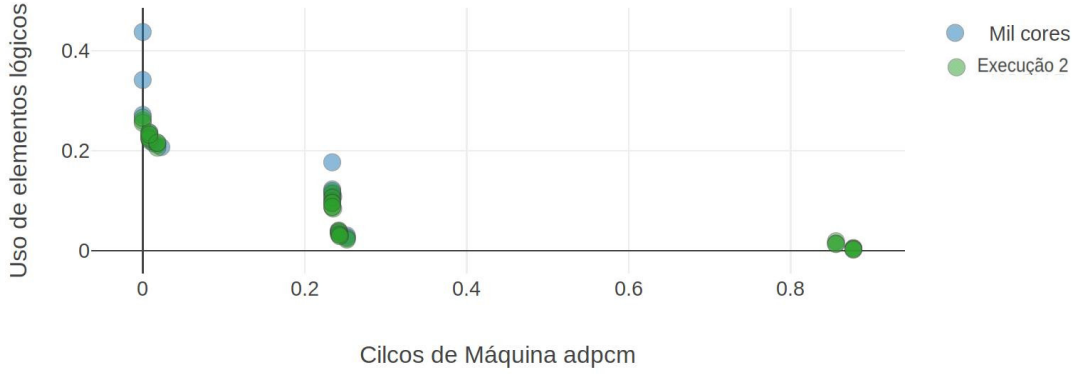


Figura 42: Fronteira de Pareto: Mil cores vs. NSGA Execução 2. Valores normalizados.



Além de analisar a evolução da qualidade das gerações por meio do hipervolume, é interessante também observar o ganho absoluto entre as soluções iniciais e as finais propostas pela otimização. Nos quadros 3, 4, 5, 6 e 7 vê-se uma comparação entre as soluções extremas, ou seja os melhores em cada métrica, da 1ª e da 20ª geração. Por exemplo, o extremo $f_{execução}$ é a solução com o melhor valor para essa métrica nessa população. Todos os valores foram normalizados em relação ao respectivo ponto da 1ª geração; dessa forma esta clara a evolução na 20ª geração.

Na maioria dos casos, vemos que a melhora da métrica em seu ponto extremo é pequena, de até no máximo 5%. No entanto, quando analisamos a segunda métrica do ponto extremo o ganho é mais expressivo; chegando a 50% no Quadro 7. Isso mostra que os extremos estão tendo o mesmo desempenho mas usando menos recursos; um dos objetivos da otimização.

Quadro 3: Evolução dos objetivos para 1ª execução: Primeira fronteira vs. última fronteira

	Extremo $f_{execução}$		**Extremo** f_{ALM}	
Geração	$f_{execução}$	f_{ALM}	$f_{execução}$	f_{ALM}
1ª	1	1	1	1
20ª	0,99	0,79	1	0,99

Quadro 4: Evolução dos objetivos para 2ª execução: Primeira fronteira vs. última fronteira

	Extremo $f_{execução}$		**Extremo** f_{ALM}	
Geração	$f_{execução}$	f_{ALM}	$f_{execução}$	f_{ALM}
1ª	1	1	1	1
20ª	0,99	0,86	1	0,99

Quadro 5: Evolução dos objetivos para 3ª execução: Primeira fronteira vs. última fronteira

	Extremo $f_{execução}$		**Extremo** f_{ALM}	
Geração	$f_{execução}$	f_{ALM}	$f_{execução}$	f_{ALM}
1ª	1	1	1	1
20ª	1	0,85	1	0,95

Quadro 6: Evolução dos objetivos para 4ª execução: Primeira fronteira vs. última fronteira

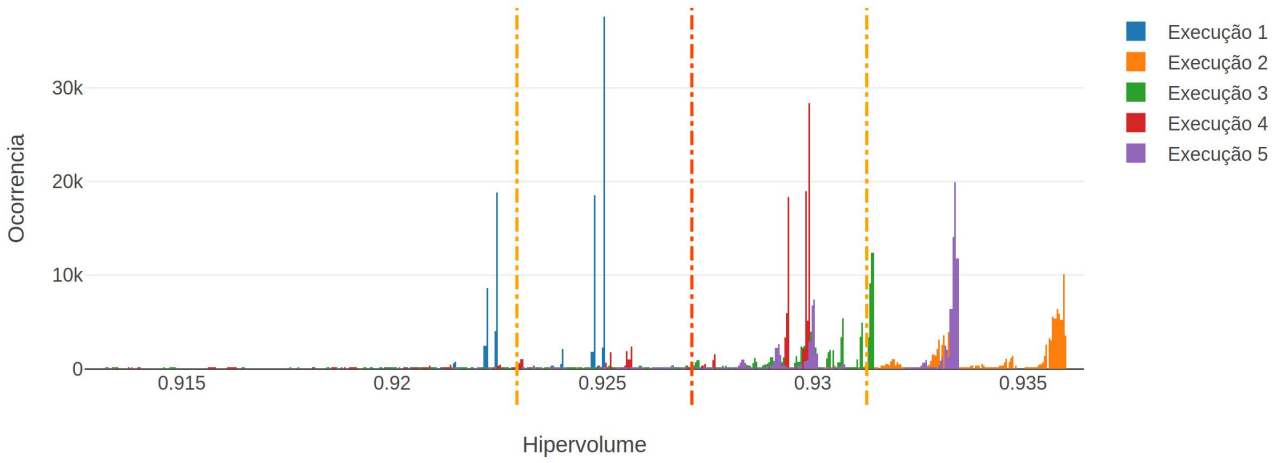
	Extremo $f_{execução}$		**Extremo** f_{ALM}	
Geração	$f_{execução}$	f_{ALM}	$f_{execução}$	f_{ALM}
1ª	1	1	1	1
20ª	0,98	0,95	1	0,94

Para verificar a robustez das fronteiras encontradas pelo NSGA-II, o *Bootstrap* foi novamente utilizado. Na Figura 43 vemos a dispersão das ocorrências do hipervolume da fronteira de Pareto referente a geração final para cada uma das cinco execuções. As linhas pontinhadas laranja e amarela correspondem a média e aos limites do desvio padrão, respectivamente, da fronteira de Pareto em *Mil Cores*. Na Quadro 8 vemos todos os dados estatísticos das figuras 38 e 43 compilados.

Quadro 7: Evolução dos objetivos para 5ª execução: Primeira fronteira vs. última fronteira

	Extremo $f_{execução}$		**Extremo** f_{ALM}	
Geração	$f_{execução}$	f_{ALM}	$f_{execução}$	f_{ALM}
1ª	1	1	1	1
20ª	1	0,79	0,50	0,97

Figura 43: Histograma das ocorrências do hipervolume da fronteira de Pareto após 100000 iterações de Bootstrap na última geração das 5 execuções. Valores normalizados.



Quadro 8: Dados estatísticos da dispersão apresentada na Figura 38 e Figura 43

Experimento	**Máximo**	**Mínimo**	**Média**	**Desvio Padrão**
Mil Cores *adpcm*	0.931	0.875	0.927	0.004
Execução 1	0.925	0.916	0.924	0.001
Execução 2	0.936	0.922	0.934	0.002
Execução 3	0.931	0.896	0.930	0.001
Execução 4	0.930	0.886	0.929	0.003
Execução 5	0.933	0.919	0.932	0.002

Foi possível verificar que as fronteiras encontradas são estatisticamente equivalentes ao estimador da fronteira Pareto-Ótima. Sendo que em 4 dos 5 experimentos, a média foi maior que a esperada. Nas figuras 44 e 45, temos um *boxplot* dos valores de hipervolume para *Mil Cores* e para as 5 execuções do NSGA-II. Portanto, que em 80% das execuções, o NSGA-II resultou em uma fronteira de Pareto melhor que nosso estimador da fronteira Pareto-Ótima.

Figura 44: Boxplot das dispersões de hipervolume do experimento *Mil Cores* e execuções de NSGA-II. Valores normalizados.

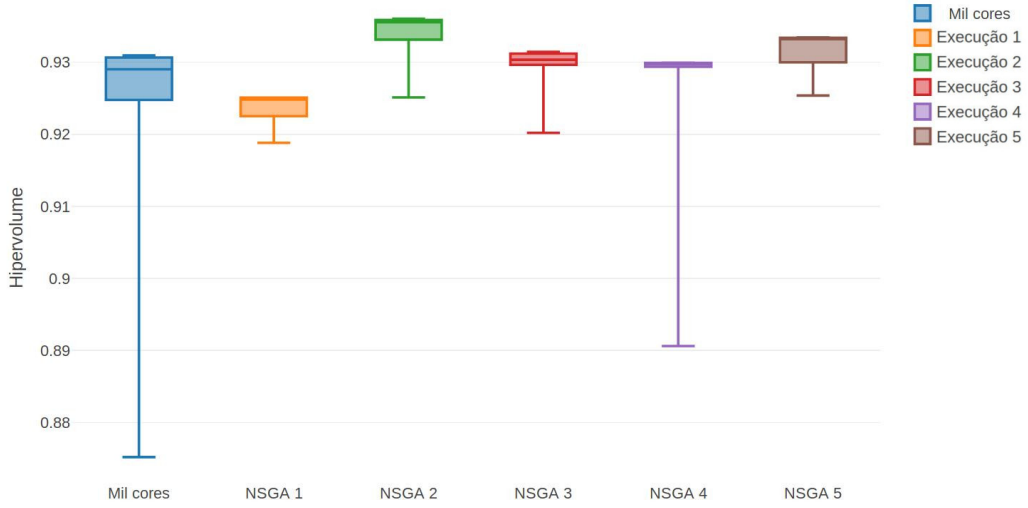
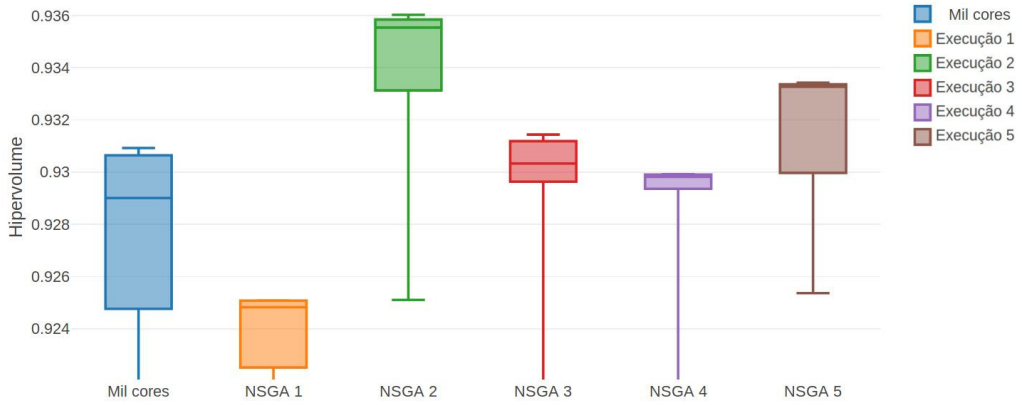


Figura 45: Boxplot das dispersões de hipervolume do experimento *Mil Cores* e execuções de NSGA-II (Detalhe). Valores normalizados.



9.3 Otimização de três objetivos: $f_{execução}$, f_{ALM} e $f_{memória}$

Inicialmente faremos a análise do espaço de objetivos para se ter uma ideia da dispersão que estamos lidando. Os experimentos realizados aqui são análogos aos apresentado na otimização para dois objetivos. Nas figuras 46 e 47 vemos o espaço de objetivos e a fronteira de Pareto obtida, em laranja, para o *kernel adpcm* em uma amostragem aleatória de 1000 cores. Esse experimento será chamado de *Mil Cores 3 Obj.*, análogo ao *Mil Cores* da seção anterior. Dessa forma serão obtidas os estimadores das fronteiras Pareto-Ótima para cada *kernel* de referência.

Figura 46: Espaço de objetivos (visão 1). x = $f_{execução}$, y = f_{ALM}, z = $f_{memória}$. Valores normalizados.

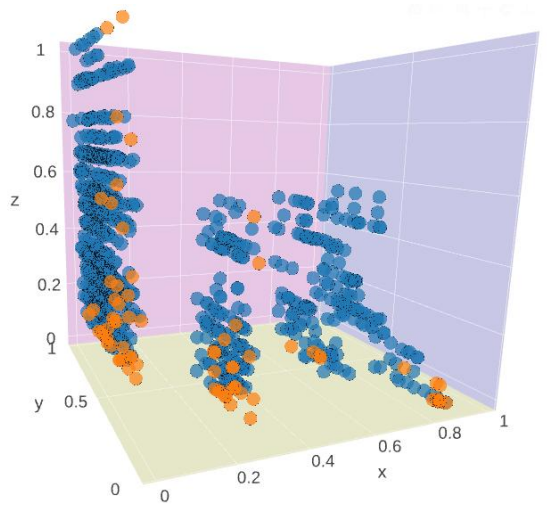


Figura 47: Espaço de objetivos (visão 2). x = $f_{execução}$, y = f_{ALM}, z = $f_{memória}$. Valores normalizados. Valores normalizados.

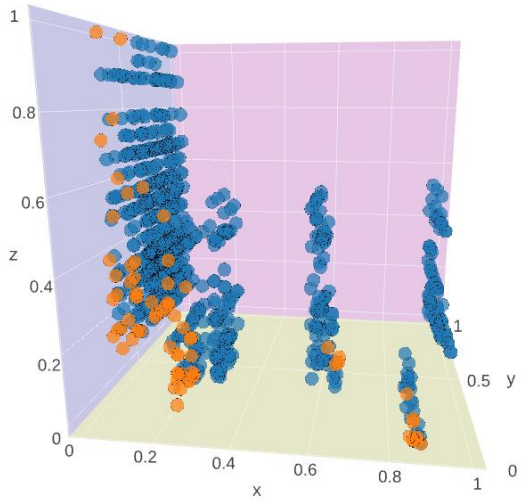


Figura 48: Fronteiras de Pareto *Mil Cores 3 Obj.* vs. Fronteiras de Pareto NSGA-II para kernel *adpcm* (visão 1). x = $f_{execução}$, y = f_{ALM}, z = $f_{memória}$. Valores normalizados. Valores normalizados.

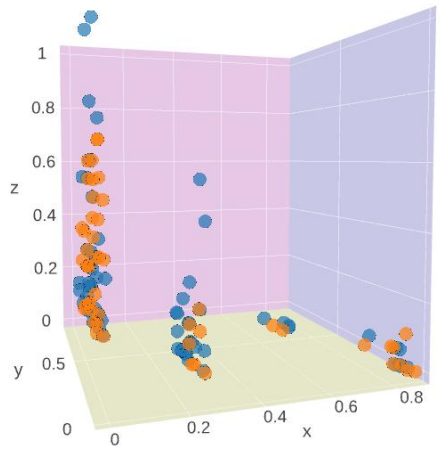
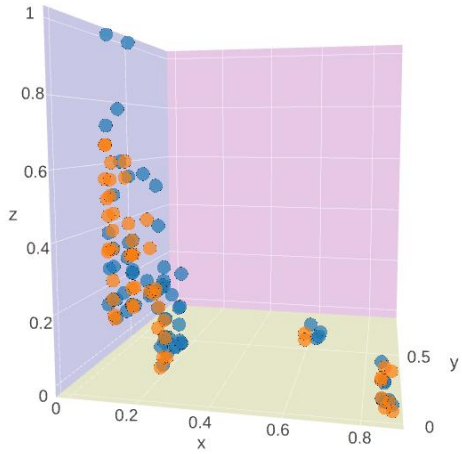
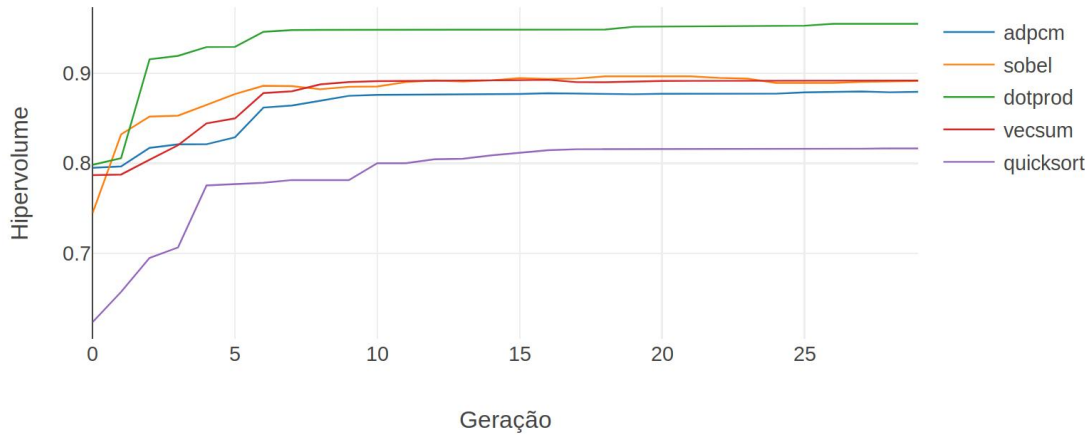


Figura 49: Fronteiras de Pareto *Mil Cores 3 Obj.* vs. Fronteiras de Pareto NSGA-II para kernel *adpcm* (visão 2). x = $f_{execução}$, y = f_{ALM}, z = $f_{memória}$. Valores normalizados. Valores normalizados.



Vê-se o mesmo agrupamento no espaço de objetivos observado no experimento *Mil Cores* da otimização de 2 objetivos, mais uma vez sugerindo que existam poucos alelos que de fato impactam na $f_{execução}$ de um individuo. As figuras 48 e 49 mostram que a fronteira encontrada esta próxima da referência *Mil Cores 3 Obj.*. Vemos também que para 3 objetivos o problema de diversidade de soluções melhorou consideravelmente. Na Figura 50 tem-se a evolução da fronteira de Pareto para cada experimento. Observa-se que aqui a convergência ocorre por volta da 5ª geração, ainda mais rapidamente que no experimento de dois objetivos. Isso é esperado pois é conhecido que o NSGA-II tem problemas de convergência prematura para otimizações de mais de 2 objetivos. Foram computadas 30 gerações de 50 indivíduos em cada execução do NSGA-II.

Figura 50: Evolução do hipervolume para 5 otimizações com *kernels* diferentes. Valores normalizados.



Nos quadros 9, 10, 11, 12 e 13, vê-se a comparação entra a 1ª e a 30ª geração em relação aos pontos extremos de cada objetivo. Aqui vemos mudanças mais variadas quando comparadas a mesma análise para a otimização de dois objetivos. Houveram mais resultados iguais e as melhoras que ocorreram foram mais expressivas. É interessante ver que em alguns casos os objetivos pioraram. No caso de um piora no objetivo do próprio ponto extremo, ou seja $f_{execução}$ piorar para o "Extremo $f_{execução}$"vemos que a fronteira esta se expandindo em extensão. Quando a piora é nos demais objetivos, temos uma retração na extensão da fronteira. É importante notar que sempre há alguma melhora, apesar de uma eventual piora, em algum objetivo. É garantindo que os pontos da fronteira final, são os menos dominados.

Mais uma vez o *Bootstrap* foi empregado nas fronteiras obtidas pelo NSGA-II e nas amostragens aleatórias de referência. Os resultados podem ser vistos nas figuras 51,52,53,54 e 55. Os dados estatísticos compilados econtram-se nos quadros 14,15,16,17 e 18. Nos histogramas, as linhas azul escura e vermelha são as médias da dispersão para *Mil Cores 3Obj.* e NSGA respectivamente; e os intervalos delimitados pelas linhas azul claro e

marrom são os desvios padrão para para *Mil Cores 3Obj.* e NSGA respectivamente.

Quadro 9: Evolução dos objetivos para o *kernel adpcm*: Primeira fronteira vs. última fronteira

	Extremo $f_{execução}$			Extremo f_{ALM}			Extremo $f_{memória}$		
Geração	$f_{execução}$	f_{ALM}	$f_{memória}$	$f_{execução}$	f_{ALM}	$f_{memória}$	$f_{execução}$	f_{ALM}	$f_{memória}$
1ª	1	1	1	1	1	1	1	1	1
20ª	0,98	0,85	0,91	2,06	0,94	0,97	1,22	0,82	0,99

Quadro 10: Evolução dos objetivos para o *kernel vecsum*: Primeira fronteira vs. última fronteira

	Extremo $f_{execução}$			Extremo f_{ALM}			Extremo $f_{memória}$		
Geração	$f_{execução}$	f_{ALM}	$f_{memória}$	$f_{execução}$	f_{ALM}	$f_{memória}$	$f_{execução}$	f_{ALM}	$f_{memória}$
1ª	1	1	1	1	1	1	1	1	1
20ª	1	0,75	1	1	0,99	1	1	1	1

Quadro 11: Evolução dos objetivos para o *kernel sobel*: Primeira fronteira vs. última fronteira

	Extremo $f_{execução}$			Extremo f_{ALM}			Extremo $f_{memória}$		
Geração	$f_{execução}$	f_{ALM}	$f_{memória}$	$f_{execução}$	f_{ALM}	$f_{memória}$	$f_{execução}$	f_{ALM}	$f_{memória}$
1ª	1	1	1	1	1	1	1	1	1
20ª	1	0,85	0,97	1	0,95	0,98	1,24	0,68	0,99

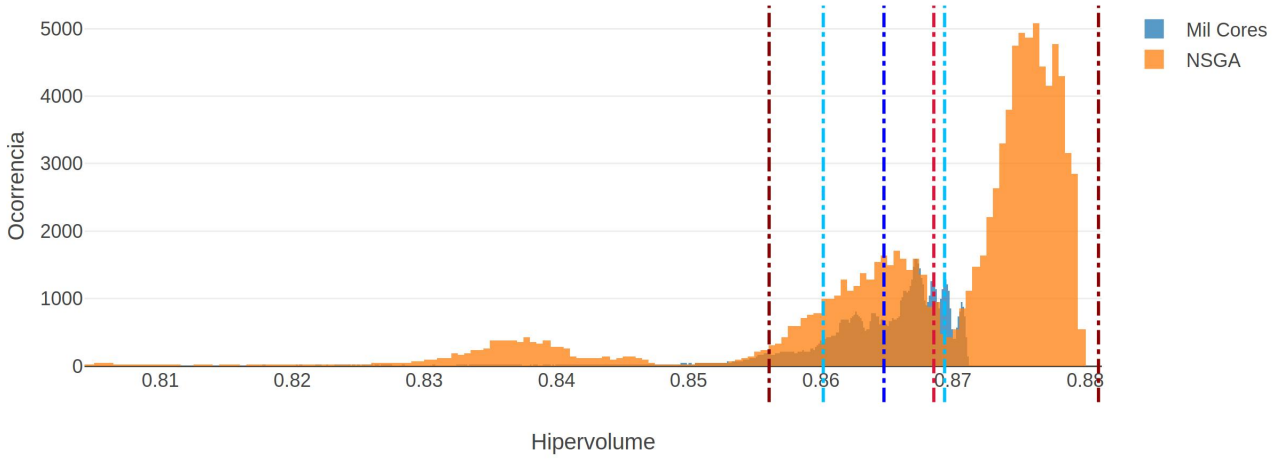
Quadro 12: Evolução dos objetivos para o *kernel dotprod*: Primeira fronteira vs. última fronteira

	Extremo $f_{execução}$			Extremo f_{ALM}			Extremo $f_{memória}$		
Geração	$f_{execução}$	f_{ALM}	$f_{memória}$	$f_{execução}$	f_{ALM}	$f_{memória}$	$f_{execução}$	f_{ALM}	$f_{memória}$
1ª	1	1	1	1	1	1	1	1	1
20ª	1	0,90	1	1	0,95	1	1	0,85	1

Quadro 13: Evolução dos objetivos para o *kernel quicksort*: Primeira fronteira vs. última fronteira

	Extremo $f_{execução}$			Extremo f_{ALM}			Extremo $f_{memória}$		
Geração	$f_{execução}$	f_{ALM}	$f_{memória}$	$f_{execução}$	f_{ALM}	$f_{memória}$	$f_{execução}$	f_{ALM}	$f_{memória}$
1ª	1	1	1	1	1	1	1	1	1
20ª	1,02	0,73	1,02	1	0,90	0,99	1	1	1

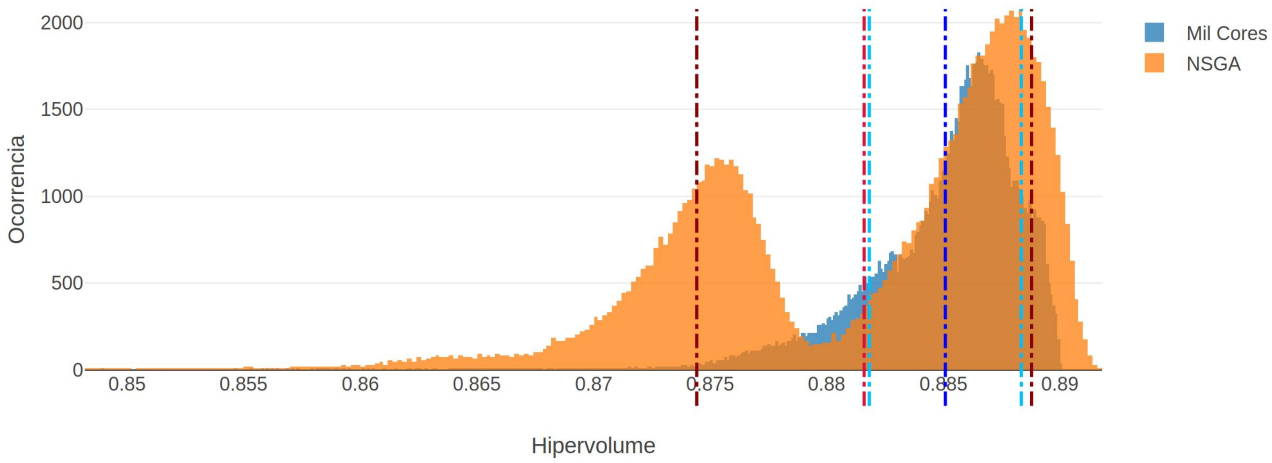
Figura 51: Histograma dos valores do hipervolume obtidos após 100000 iterações de *Bootstrap* sobre das fronteiras de Pareto encontradas pelo *Mil Cores 3 Obj.* e pelo NSGA. *Kernel* usado *adpcm*. Valores normalizados.



Quadro 14: Dados estatísticos da dispersão apresentada na Figura 51

Experimento	**Máximo**	**Mínimo**	**Média**	**Desvio Padrão**
Mil Cores *adpcm*	0.871	0.818	0.865	0.005
NSGA *adpcm*	0.880	0.722	0.869	0.012

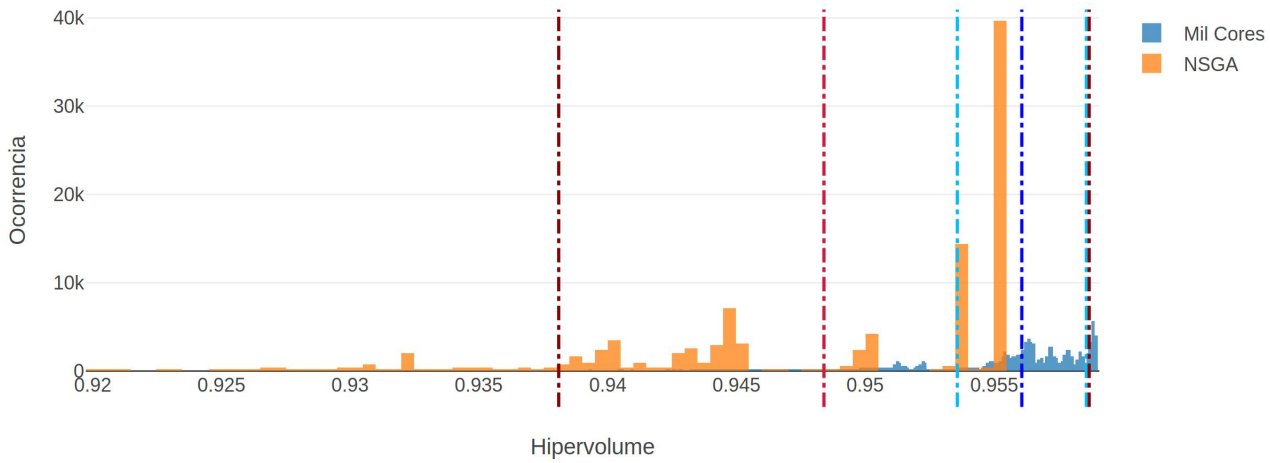
Figura 52: Histograma dos valores do hipervolume obtidos após 100000 iterações de Bootstrap sobre das fronteiras de Pareto encontradas pelo *Mil Cores 3 Obj.* e pelo NSGA. Kernel usado *sobel*. Valores normalizados.



Quadro 15: Dados estatísticos da dispersão apresentada na Figura 52

Experimento	**Máximo**	**Mínimo**	**Média**	**Desvio Padrão**
Mil Cores *sobel*	0.890	0.8149	0.885	0.003
NSGA *sobel*	0.892	0.824	0.881	0.007

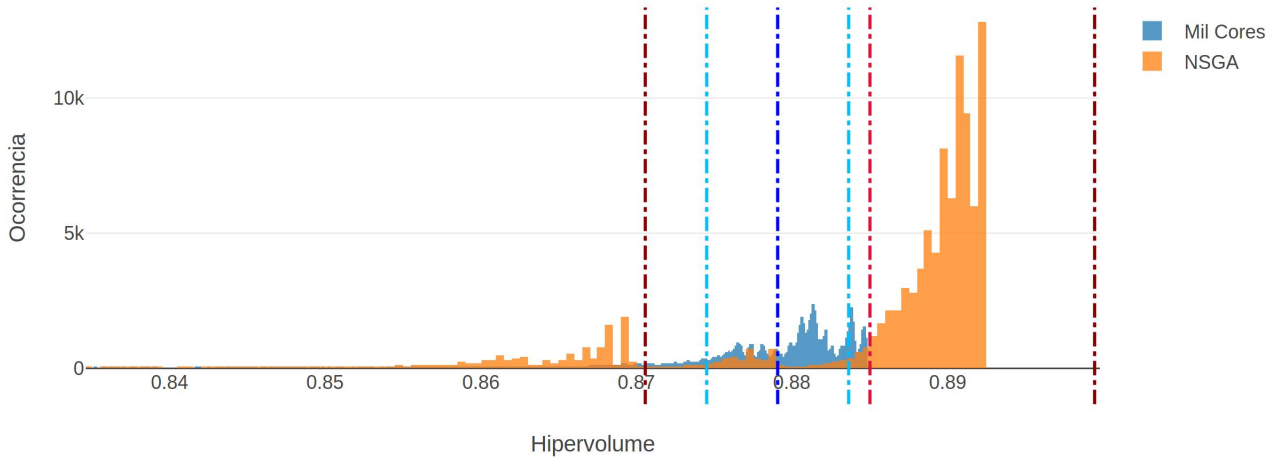
Figura 53: Histograma dos valores do hipervolume obtidos após 100000 iterações de Bootstrap sobre das fronteiras de Pareto encontradas pelo *Mil Cores 3 Obj.* e pelo NSGA. Kernel usado *dotprod*. Valores normalizados.



Quadro 16: Dados estatísticos da dispersão apresentada na Figura 53

Experimento	**Máximo**	**Mínimo**	**Média**	**Desvio Padrão**
Mil Cores *dotprod*	0.959	0.939	0.956	0.003
NSGA *dotprod*	0.955	0.708	0.948	0.010

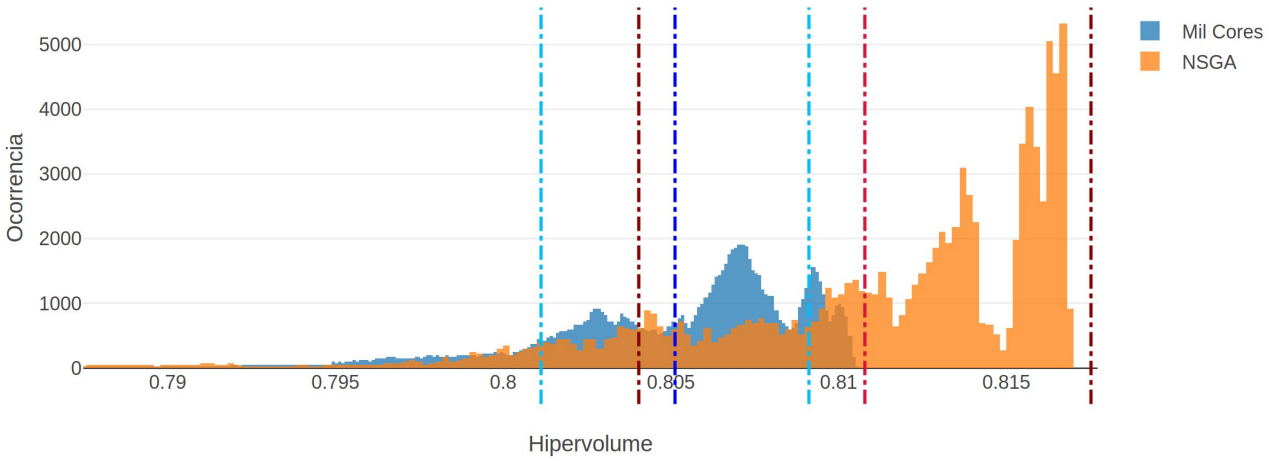
Figura 54: Histograma dos valores do hipervolume obtidos após 100000 iterações de Bootstrap sobre das fronteiras de Pareto encontradas pelo *Mil Cores 3 Obj.* e pelo NSGA. Kernel usado *vecsum*. Valores normalizados.



Quadro 17: Dados estatísticos da dispersão apresentada na Figura 54

Experimento	**Máximo**	**Mínimo**	**Média**	**Desvio Padrão**
Mil Cores *vecsum*	0.885	0.831	0.879	0.005
NSGA *vecsum*	0.892	0.686	0.885	0.014

Figura 55: Histograma dos valores do hipervolume obtidos após 100000 iterações de Bootstrap sobre das fronteiras de Pareto encontradas pelo *Mil Cores 3 Obj.* e pelo NSGA. Kernel usado *quicksort*. Valores normalizados.



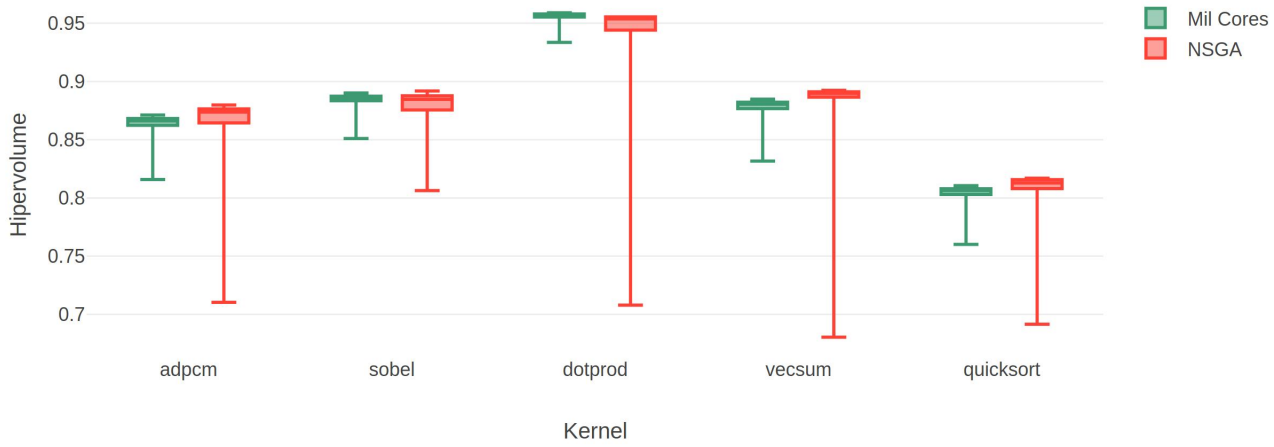
Vemos que em 3 (*adpcm, vecsum e quicksort*) dos 5 experimentos, foram obtidos resultados estatisticamente equivalentes ou melhores que a referência de *Mil Cores 3Obj.*

Quadro 18: Dados estatísticos da dispersão apresentada na Figura 55

Experimento	**Máximo**	**Mínimo**	**Média**	**Desvio Padrão**
Mil Cores *quicksort*	0.811	0.765	0.805	0.004
NSGA *quicksort*	0.817	0.693	0.811	0.007

Somente no experimento correspondente a *dotprod*, tivemos um resultado pior mas ainda sim com intersecção nas regiões de delimitadas pelos desvios padrão. Na figura Figura 56 tem-se um *boxplot* para as dispersões de hipervolume encontradas.

Figura 56: Boxplot das dispersões de hipervolume *Mil Cores 3Obj.* e NSGA-II para os 5 kernels diferentes. Valores normalizados.



10 CONCLUSÕES

O objetivo desse trabalho era realizar uma prova de conceito: verificar a viabilidade do uso da meta-heurística dos algoritmos genéticos para uma otimização multiobjetivo no contexto de co-projeto hardware/software. Assumiu-se um particionamento em que os *kernels* de referência seriam executados em *software* e a partir disso, uma otimização do hardware seria realizada com o objetivo de obter as soluções de melhor desempenho segundo tempo de execução e uso de recursos computacionais. Para isso, foi implementada uma versão do NSGA-II, em conjunto com as ferramentas de desenvolvimento de Hardware e Software da Intel/Altera, para criar uma ferramenta capaz de evoluir o soft-core Nios II para melhor se adaptar a uma aplicação de referência.

Apesar dos experimentos apresentarem resultados promissores, tanto para otimizações de dois como de três objetivos, não é possível confirmar a eficiência do AG implementado. Por conta de recursos computacionais limitados e tempo finito, não foi possível realizar uma quantidade razoável de experimentos para ter uma confiança maior nos resultados. Tentou-se minimizar essa carência com técnicas de re-amostragem mas isso tem suas limitações. O que temos é um forte indício de que o AG seria uma boa alternativa para problemas de otimização de co-projeto HW/SW.

Na subseção 8.5.1 foi mostrado como a tarefa de avaliação dos cromossomos foi paralelizada. Essa particularidade na implementação foi fundamental para os resultados desse trabalho. Sem ela, teria sido impossível realizar os experimentos no tempo disponível. Vale ressaltar que a arquitetura proposta é escalável. Caso mais núcleos de processamento estivessem disponíveis, mais experimentos poderiam ter sido realizados.

O NSGA-II é um MOGA amplamente conhecidos mas que já está ultrapassado em alguns aspectos. Por exemplo, O NSGA-II não verifica correlação entre os genes. Relação esta que se levada em consideração pode-se levar a melhores resultado mais rapidamente. MOEAs mais modernos, como os *Algoritmos de estimação de Distribuição*, modelam a relação entre genes e tendem a desempenhar melhor. Além disso verificou-se experimentalmente a esperada convergência prematura do NSGA-II para otimizações de mais de 2 objetivos.

Um dos diferenciais desse trabalho foi a metodologia de avaliação empregada. Realizar os *benchmarks* com *kernels* de perfil estudados em um hardware real traz um bom valor aos resultados por estarem mais próximos da realidade.

O uso de Algoritmos Evolutivos, em particular os Algoritmos Genéticos, para otimização de arquiteturas de soft-cores mostrou-se uma opção promissora e digna de atenção futura. Caso melhor estudado e refinado, uma ferramenta que use esses conceitos

poderia trazer ganhos reais no contexto da engenharia.

10.1 Trabalhos futuros

Como foi mencionado, o número de experimentos realizados não foi o ideal. Em um primeiro momento futuro, seria interessante contornar o principal gargalo da execução dessa ferramenta: núcleos de processamento. Como a arquitetura da implementação proposta possibilita uma paralelização escalável, seria interessante realizar os experimentos em um ambiente altamente distribuído como um *cluster*. Dessa forma, seria possível se realizar mais experimentos e ter uma confiança estatística superior dos resultados. Além disso, seria possível realizar uma busca completa no espaço de design e assim ter uma ideia definitiva com relação ao desempenho da otimização; já que estaríamos usando de fato a fronteira Pareto-Ótima, invés de um estimador.

Nesse trabalho somente características do Nios II foram incluídas na modelagem dos cromossomos. No entanto, seria possível estender a flexibilidade ao sistema inteiro. Dessa maneira um sistema completo em hardware poderia ser otimizado; e não só um de seus elementos isoladamente. A modelagem desses elementos inclusive, poderia ser feita de maneira a possibilitar a existência de sistemas heterogêneos a nível de CPU, como descritos em Silva (2016), ou a nível de sistema. Além disso, outros objetivos também poderiam ser incluídos como, por exemplo, gasto energético, custo estimado de produção etc.

O foco dos resultados desse trabalho foram primariamente referentes ao processo de otimização e a qualidade das soluções propostas. No entanto, seria interessante realizar uma análise acerca de quais fatores configuram uma boa solução. Ou seja, quais são os alelos que de fato impactam no desempenho e como isso se relaciona ao perfil do *kernel* de referência.

REFERÊNCIAS

AUGER, A. et al. Theory of the hypervolume indicator: Optimal -distributions and the choice of the reference point. **Foundations of Genetic Algorithms**, Estados Unidos da America, 2009.

AZEVEDO, C. R. B.; ARAUJO, A. F. R. Correlation between diversity and hypervolume in evolutionary multiobjective optimization. **IEEE Congress of Evolutionary Computation**, p. 2743–2750, 2011.

BOBDA, C. **INTRODUCTION TO RECONFIGURABLE COMPUTING**: Architectures, algorithms, and application. Alemanha: Kluwer Academic Publishers, 2007.

BONDALAPATI, K.; PRASANNA, V. K. Reconfigurable computing systems. **Proceedings of the IEEE**, v. 90 No. 7, p. 1201 – 1217, 2002.

BONETTI, D. R. F. **Aumento da eficiencia do calculo da energia de van der Waals em algoritmos geneticos para predição de estruturas de proteinas**. 2010. Tese (Doutor em Ciências) — Instituto de Ciências Matemáticas e de Computação de São Carlos, Universidade de São Paulo, São Carlos, 2010.

CANTU-PAZ, E. A survey of parallel genetic algorithms. **CALCULATEURS PARALLELES, RESEAUX ET SYSTEMS REPARTIS**, v. 10, p. 1201 – 1217, 1998.

CATANIA, V.; MALGERI, M.; RUSSO, M. Applying fuzzy logic to codesign partitioning. **IEEE Micr**, v. 17 No. 3, p. 62 – 70, 1997.

CODE, R. Dot product. 2018. Disponível em: <https://rosettacode.org/wiki/Dot_product>.

_____. Sorting algorithms/quicksort. 2018. Disponível em: <https://rosettacode.org/wiki/Sorting_algorithms/Quicksort>.

COMPTON, K.; HAUCK, S. An introduction to reconfigurable computing. **IEEE Computer**, 2000.

DEB, K. Multi-objective genetic algorithms: Problem difficulties and construction of test problems. **Evolutionary Computation**, Estados Unidos da America, 1999.

_____. An efficient constraint handling method for genetic algorithms. **Comput. Methods Appl. Mech. Engrg.**, India, v. 186, p. 311 – 338, 2000.

DEB, K. et al. A fast and elitist multiobjective genetic algorithm: Nsga-ii. **IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION**, Vol. 6 No. 2, p. 182 – 197, 2002.

DEB, K.; SRINIVAS, N. Muiltiobjective optimization using nondominated sorting in genetic algorithms. **Evolutionary Computation**, Vol. 2 No. 3, p. 221–248, 1994.

DIAS, M. A. **Co-Projeto de hardware/software para correlação de imagens**. 2011. Dissertação (Mestrado em Ciências) — Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos, 2011.

FRANK, C. P. et al. Evolutionary multi-objective multi-architecture design space exploration methodology. **Optimization and Engineering**, v. 19 No. 2, p. 359–381, 2018.

HIDALGO, J.; LANCHARES, J. Functional partitioning for hardware - software codesign using genetic algorithms. **EUROMICRO 97. Proceedings of the 23rd EUROMICRO Conference: New Frontiers of Information Technology**, Hungria, p. 631 – 638, 1997.

HILLIS, D. M.; BULL, J. J. An empirical test of bootstrapping as a method for assessing confidence in phylogenetic analysis. **Systematic Biology**, v. 42, p. 182 – 192, 1993.

HOLLAND, J. H. **Adaptation in Natural and Artificial Systems**: An introductory analysis with applications to biology, control, and artificial intelligence. Estados Unidos da America: The MIT Press, 1975.

INSTRUMENTS, T. **Hands-On: Realizing the MSP430 Signal Chain through ADPCM**. [S.l.], 2018. Disponível em: <http://www.ti.com/lit/ml/slap122/slap122.pdf>. Acesso em: 01 novembro 2018.

_____. **TMS320C55x DSP Programmer's Guide**. [S.l.], 2018. Disponível em: <http://www.ti.com/lit/ug/spru376a/spru376a.pdf>. Acesso em: 01 novembro 2018.

_____. **TMS320C64x DSP Library Programmer's Reference**. [S.l.], 2018. Disponível em: <http://www.ti.com/lit/ug/spru565b/spru565b.pdf>. Acesso em: 01 novembro 2018.

_____. **TMS320C64x Image/Video Processing Library**. [S.l.], 2018. Disponível em: <http://www.ti.com/lit/ug/spru023b/spru023b.pdf>. Acesso em: 01 novembro 2018.

INTEL. **Quartus II Scripting Reference Manual**. [S.l.], 2013. Disponível em: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/tclscriptrefmnl.pdf>. Acesso em: 02 novembro 2018.

_____. **Nios II Processor Reference Guide**. [S.l.], 2018. Disponível em: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2cpu-nii5v1gen2.pdf>. Acesso em: 01 novembro 2018.

_____. **Nios II Software Developer's Handbook**. [S.l.], 2018. Disponível em: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2sw_nii5v2gen2.pdf>. Acesso em: 01 novembro 2018.

KANG, E.; JACKSON, E.; SCHULTE, W. An approach for effective design space exploration. **Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems**, p. 33 – 54, 2010.

KOUDIL, M. et al. Using artificial bees to solve partitioning and scheduling problems in codesign. **Applied Mathematics and Computation**, v. 186 No. 2, p. 1710 – 1722, 2007.

MARTINS, J. P. **Análise da aprendizagem de ligações em otimização evolutiva**. 2015. Tese (Doutor em Ciências) — Instituto de Ciências Matemáticas e de Computação de São Carlos, Universidade de São Paulo, São Carlos, 2015.

MARTINS, L. G. A. **Exploração de sequências de otimização do compilador baseada em técnicas hibridas de mineração de dados complexos**. 2015. Tese (Doutor em Ciências) — Instituto de Ciências Matemáticas e de Computação de São Carlos, Universidade de São Paulo, São Carlos, 2015.

MICHELI, G. D.; GUPTA, R. K. Hardware/software co-design. **Proceedings of the IEEE**, Alemanha, v. 85 No. 3, p. 349 – 365, 1997.

MISHRA, A. et al. Hardware software co-design using profiling and clustering. **2012 International Conference on Communication, Information Computing Technology**, 2012.

SASTRY, K.; GOLDBERG, D.; KENDALL, G. Genetic algorithms. In: _____. **Search Methodologies**. Estados Unidos da America: Springer, 2005. cap. 4, p. 97–125.

SILVA, B. de A. **Um método de otimização da relação desempenho/consumo de energia para arquiteturas multi-cores heterogeneas em FPGA**. 2016. Tese (Doutor em Ciências) — Instituto de Ciências Matemáticas e de Computação de São Carlos, Universidade de São Paulo, São Carlos, 2016.

SOARES, A. H. M. **Algoritmos de estimação de distribuição baseados em árvores filogenéticas**. 2014. Tese (Doutor em Ciências) — Instituto de Ciências Matemáticas e de Computação de São Carlos, Universidade de São Paulo, São Carlos, 2014.

TEICH, J. Hardware/software codesign: The past, the present, and predicting the future. **Proceedings of the IEEE**, Alemanha, v. 100, p. 1411 – 1430, 2012.

YIANNACOURAS, P.; ROSE, J.; STEFFAN, J. G. The microarchitecture of fpga-based soft-processors. **Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems**, Canada, p. 202 – 212, 2005.

ZITZLER, E.; THIELE, L. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. **IEEE Transactions on Evolutionary Computation**, Vol. 3 No. 4, p. 257 – 271, 1999.