UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
UNIVERSIDADE DO ESTADO DO RIO GRANDE DO NORTE
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

JULIENE VIEIRA DO COUTO

GERAÇÃO DE CÓDIGO OTIMIZADO VISANDO A EXPLORAÇÃO DE PARALELISMO NA ARQUITETURA IPNOSYS

MOSSORÓ - RN

2016

JULIENE VIEIRA DO COUTO

GERAÇÃO DE CÓDIGO OTIMIZADO VISANDO A EXPLORAÇÃO DE PARALELISMO NA ARQUITETURA IPNOSYS

Dissertação apresentada ao Mestrado em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação da Universidade Federal Rural do Semi-Árido e da Universidade do Estado do Rio Grande do Norte como requisito para a obtenção do título de Mestre em Ciência da Computação.

Linha de pesquisa: Projeto de Sistemas e Circuitos

Orientador: Silvio Roberto Fernandes de Araújo, Prof. Dr.

MOSSORÓ-RN

2016

V871g Vieira do Couto, Juliene.
GERAÇÃO DE CÓDIGO OTIMIZADO VISANDO A EXPLORAÇÃO DE PARALELISMO NA ARQUITETURA IPNOSYS / Juliene Vieira do Couto. - 2016.
107 f. : il.

Orientador: Silvio Roberto Fernandes de Araújo.
Dissertação (Mestrado) - Universidade Federal Rural do Semi- árido, Programa de Pós-graduação em Ciência da Computação, 2016.

1. Arquiteturas Paralelas. 2. Compilador. 3. IPNoSys. 4. Otimização. 5. Nível de otimização. I. Fernandes de Araújo, Silvio Roberto, orient.

O serviço de Geração Automática de Ficha Catalográfica para Trabalhos de Conclusão de Curso (TCC´s) foi desenvolvido pelo Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo (USP) e gentilmente cedido para o Sistema de Bibliotecas da Universidade Federal Rural do Semi-Árido (SISBI-UFERSA), sendo customizado pela Superintendência de Tecnologia da Informação e Comunicação (SUTIC) sob orientação dos bibliotecários da instituição para ser adaptado às necessidades dos alunos dos Cursos de Graduação e Programas de Pós-Graduação da Universidade.

GERAÇÃO DE CÓDIGO OTIMIZADO VISANDO A EXPLORAÇÃO DE PARALELISMO NA ARQUITETURA IPNOSYS

Dissertação apresentada ao Mestrado em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação da Universidade Federal Rural do Semi-Árido e da Universidade do Estado do Rio Grande do Norte como requisito para a obtenção do título de Mestre em Ciência da Computação.

Linha de pesquisa: Projeto de Sistemas e Circuitos

APROVADA EM: 09 / 09 / 2016

Prof. Dr. Sílvio Roberto Fernandes de Araújo
(Orientador – UFERSA)

Prof. Dr. Raimundo Santos Moura
(Examinador Externo - UFPI)

Prof. Dr. Paulo Gabirel Gadelha Queiroz
(Examinador Interno - UFERSA)

Profa. Dra. Angélica Félix de Castro
(Examinadora Interna - UFERSA)

A Francisca Francinete (*in memorian*), minha avó materna, que com certeza se estivesse entre nós estaria muito feliz e orgulhosa de mais uma etapa concluída em minha vida.

A Expedito João (*in memorian*), meu avô paterno, que sempre cuidou muito bem de mim, e de onde ele estiver está bastante orgulhoso do seu "xodó".

A Joubert Januário (*in memorian*), meu irmãozinho que muito cedo se tornou um anjinho. Não tenho dúvidas de que lá do céu ele estará sempre torcendo por mim e bastante feliz por mais uma etapa da minha vida concluída.

Ao meu filho (*in memorian*) que virou um anjinho durante esse mestrado. Nem cheguei a conhecer seu rostinho, mas em tão pouco tempo me transformou e me fez descobrir o que é o amor verdadeiro.

A José Januário, Maria Antônia, Juliete Vieira e Januário Vieira, meus pais e meus irmãos, meu alicerce, pessoas incondicionais em minha vida, que sempre me incentivaram a não desistir de nada, é a quem recorro sempre que preciso de apoio. Quero dedicar este trabalho que também é fruto do esforço e apoio de vocês.

AGRADECIMENTOS

Primeiramente, quero agradecer a Deus que ilumina e protege meu caminho todos os dias. Sou grata pelo dom da vida, pela saúde e pela força que Ele me transmite. Nesses últimos 2 anos passei por muitos problemas pessoais, "caindo" algumas vezes, que influenciaram diretamente meu rendimento, mas Ele me "levantou" logo em seguida e me ajudou a concluir esse mestrado.

Aos meus pais, José Januário e Maria Antônia, os melhores pais do mundo, meu exemplo de vida. Nada que eu venha a falar conseguirei expressar todo o meu agradecimento a essas pessoas. Obrigada por todo investimento em minha educação, pelo amor, paciência, dedicação, apoio, cada lágrima derramada e cada sorriso compartilhado comigo, simplesmente por acreditar em mim e por serem meus pais. Muito obrigada, farei o possível e o impossível para sempre retribuir com muito orgulho cada gesto de vocês.

Aos meus irmãos, Juliete Vieira, Januário Vieira e Joubert Januário (*in memorian*) obrigada simplesmente por me aguentarem, pelo apoio e confiança depositados em mim. Obrigada por compartilharem e comemorarem comigo cada etapa concluída. Essa conquista também é de vocês.

A família LAACOSTE, especialmente aos meus quase irmãos, Álamo Silva e Dênis Freire, que compartilharam comigo não somente 2 anos, mas sim quase 9 anos de muita luta árdua. Obrigada pela paciência, apoio em cada momento que me desesperei, pela palavra e ombro amigo quando mais precisei.

A todos os meus familiares e amigos que se alegram a cada conquista minha e torcem pelo meu próximo passo nessa caminhada da vida, em especial a minha tia Teté que foi como uma segunda mãe, minha tia Aia que contribuiu com a base de meus estudos com muita paciência para me ensinar, e ao meu esposo David Douglas pelo companheirismo, dedicação e compreensão nesses últimos anos.

A meu orientador Silvio Roberto pela dedicação, orientação, ensinamentos, paciência, críticas construtivas, por entender todas as vezes que precisei parar para me recompor e também pelos "puxões de orelha".

A CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pela concessão da bolsa durante todo o período de realização desta dissertação.

“A persistência é o caminho do êxito.”

Charles Chaplin

RESUMO

As arquiteturas paralelas necessitam de código otimizado que explore seus novos recursos. Algumas arquiteturas seguem o paradigma da máquina de Von Neumann, enquanto que outras divergem desse modelo, um exemplo é o processador IPNoSys. Esse processador foi baseado em redes-em-chip e apresenta um modelo de computação dirigido a pacotes o que reflete no seu modelo de programação. Inicialmente, essa arquitetura possuía um montador e um simulador e necessitava de um compilador. Em trabalhos posteriores compiladores para a IPNoSys foram desenvolvidos, mas nenhum explorou completamente as características dessa arquitetura. Com isso, o objetivo deste trabalho é definir uma etapa de otimização de código no compilador IPNoSys, considerando características não exploradas como o paralelismo e melhorando seu código gerado. O módulo de otimização oferece três níveis de otimização. A fim de avaliar o módulo criado, efetuou-se uma comparação do tempo de execução e do tamanho dos códigos gerados nos três níveis de otimização. Foi obtido que um nível de otimização apresentou melhor tempo de execução, porém gerou aplicações com um maior tamanho, enquanto que outro nível apresentou um menor tamanho. Além disso, houve uma melhoria nos códigos gerados.

Palavras-chave: Arquiteturas Paralelas. Compilador. IPNoSys. Otimização. Nível de otimização.

ABSTRACT

Parallel architectures require optimized code that exploits its new features. Some architectures follow the paradigm of Von Neumann machine, while others differ from this model, such as IPNoSys processor. This processor is based on network-on-chip and features a package-driven computer model driven which reflects in its programming model. Initially, this architecture had an assembler and a simulator and needed a compiler. In later papers compilers for IPNoSys have been developed, but none fully explored the features of this architecture. Thus, the objective of this paper is to define a code optimization step in IPNoSys compiler, considering characteristics unexploited as parallelism and improving your generated code. The optimization module offers three levels of optimization. In order to evaluate the created module, made a comparison of the execution time and the size of codes generated in the three levels of optimization. It was obtained that an optimization level showed better run time, but generated applications with a larger size, while another level showed a smaller size. Furthermore, there was an improvement in the generated code.

Keywords**:** *Parallel archtectures. Compiler. IPNoSys. Optimization. Levels of Optimization.*

LISTA DE FIGURAS

LISTA DE TABELA

LISTA DE ABREVIATURAS E SIGLAS

AST	*Abstract Syntax Tree*
BSD	*Berkeley Software Distribution*
C2PDL	*C to PDL*
CPU	*Central Processing Unit*
DCT	*Discrete Cosine Transform*
GCC	*GNU Compiler Collection*
IPNoSys	*Integrated Processing NoC System*
JIT	*Just in time*
LLVM	*Low Level Virtual Machine*
LLVM-IR	*Low Level Virtual Machine Intermediate Representation*
MAU	*Memory Access Unit*
MIMD	*Multiple Instruction Streams, Multiple Data Stream*
MISD	*Multiple Instruction Streams, Single Data Stream*
NoC	*Network-on-Chip*
PDL	*Package Description Language*
RLE	*Run-Length Encoding*
RPU	*Routing and Processing Unit*
RTL	*Register Transfer Language*
SIGPLAN	*Special Interest Group on Programming Languages*
SIMD	*Single Instruction Stream, Multiple Data Stream*
SISD	*Single Instruction Stream, Single Data Stream*
SSA	*Single Static Assignement*
UIUC	*University of Illinois at Urbana-Champain*
ULA	Unidade Lógica e Aritmética

SUMÁRIO

1 INTRODUÇÃO

A demanda pela geração de código otimizado tem crescido devido ao aumento da complexidade dos atuais projetos de microprocessadores, visando a exploração e o melhor uso dos novos recursos dessas arquiteturas. Otimização consiste na transformação dos programas, na qual uma nova versão semanticamente igual ao código inserido é gerado, porém com desempenho superior – por exemplo: em tempo de compilação, tempo de execução ou tamanho do código (AHO; SETHI; ULLMAN, 2007). Nos compiladores modernos pode-se aplicar otimizações a um código no processo de compilação. Em alguns dos principais compiladores utilizados atualmente, tais como: GCC (*GNU Compiler Collection*) (GCC, 2015) e a infraestrutura do LLVM (*Low Level Virtual Machine*) (LLVM, 2015), várias otimizações estão disponíveis, algumas das quais serão apresentadas no Capítulo 2, e o programador faz a escolha de qual otimização utilizar no programa (XAVIER, 2014).

Durante as duas últimas décadas, pesquisadores descobriram que a qualidade do código após a otimização variava de acordo com o programa (CAVAZOS; O'BOYLE, 2006; COOPER et al., 1999; FURSIN et al., 2005; HOSTE; EECKHOUT, 2008). Apesar da melhora no desempenho, não existe uma garantia de que o código resultante seja ótimo (LOUDEN, 2004). Além disso, um código pode ser transformado de modo que haja perda de desempenho (XAVIER, 2014).

Existem dois tipos de otimizações: independentes da máquina e dependentes da máquina. As otimizações independentes da máquina são transformações realizadas no código alvo sem considerar características da arquitetura alvo, já das otimizações dependentes da máquina necessitam dessas características (LOUDEN, 2004).

As principais técnicas de otimização podem ser aplicadas em uma vasta gama de processadores. Alguns desses processadores apresentam diversas características em comum com a máquina de Von Neumann. Esse tipo de máquina segue um modelo de programas e dados armazenados em uma memória central, com o processador organizado com um caminho de dados e um de controle, o qual funciona como uma máquina de estados do tipo busca instrução, decodifica, executa, armazena o resultado e volta a buscar uma nova instrução. Em busca da melhoria do desempenho, várias arquiteturas seguem organizações e paradigmas diferentes do modelo tradicional de Von Neumann e por isso são classificadas como não convencionais (ADAMATZKY, 2014).

Um processador não convencional proposto por ARAUJO (2008) chamado de IPNoSys

(*Integrated Processing NoC System*) foi criado com o objetivo de alto desempenho, baseando-se nas características das redes em chip (NoC – *Network on chip*). Essa arquitetura explora as vantagens existentes nas NoCs, como paralelismo, reusabilidade e escalabilidade. O código executável dessa arquitetura é formatado em um modelo de pacote, assim como é usado nas NoCs, o qual é executado enquanto esse pacote vai trafegando pelas várias unidades de execução (FERNANDES; SILVA; KREUTZ, 2010). Sua linguagem *assembly* apesar de abstrata, do ponto de vista do código de máquina, remete a várias características dos pacotes e por isso é chamada de PDL (*Package Description Language*). Inicialmente, a arquitetura oferecia além de um simulador descrito em *SystemC* (ACCELLERA, 2015), um montador que realiza análise sintática e semântica do código PDL e faz a geração de código binário para a arquitetura. IPNoSys utiliza um modelo de programação conhecido na literatura como "tudo explícito", que é de responsabilidade do programador indicar como o programa deverá ser particionado, o mapeamento de suas partes, como essas partes devem se comunicar e serem sincronizadas (ARAUJO, 2012).

A implementação de código PDL é árdua, especialmente em programas robustos. Dessa forma, existia a necessidade do desenvolvimento de um compilador que traduzisse um código escrito em uma linguagem de alto nível, como C, para o código PDL. Houve alguns trabalhos de desenvolvimento de um compilador para a arquitetura IPNoSys, como será visto posteriormente no Capítulo 2. Em todos esses trabalhos, as características da arquitetura não foram exploradas completamente.

Beneficiando-se da geração de código PDL de um dos compiladores da IPNoSys, foi criada uma etapa de otimização para esse compilador. Este otimizador oferece três diferentes níveis, entre os quais o último nível realiza a exploração do paralelismo dessa arquitetura.

O objetivo deste trabalho é a geração automática de código otimizado, a partir de código C, para a arquitetura IPNoSys, explorando as características dessa arquitetura. A ferramenta proposta permite ao usuário escolher o nível de otimização que deseja baseando-se em qual característica (tempo de execução ou tamanho do código) ele necessita melhorar.

Este texto está organizado da seguinte forma: no Capítulo 2, é apresentada a Fundamentação Teórica, na qual aborda-se conceitos e técnicas de Otimização de Código presentes na literatura; definições de Arquiteturas Paralelas e a Arquitetura IPNoSys, que corresponde ao processador alvo deste trabalho; e ainda, algumas Ferramentas de Compilação e Otimização e Trabalhos Relacionados. No Capítulo 3, descreve-se o Compilador Otimizante C2PDL. No capítulo 4, são mostrados os Resultados de Validação. E por fim, no Capítulo 5, são expostas as Considerações Finais e Trabalhos Futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são expostos assuntos que darão um melhor embasamento para o entendimento deste trabalho. O objetivo deste trabalho é a geração de código otimizado visando a exploração do paralelismo em uma arquitetura não convencional. Inicialmente, é necessário conhecer o conceito e algumas técnicas de otimização de código presentes na literatura. Em seguida, são mostradas as arquiteturas paralelas, que podem ser de dois tipos: arquiteturas convencionais e arquiteturas não convencionais. Foi escolhida como arquitetura alvo deste trabalho, a arquitetura não convencional IPNoSys. Ela é apresentada na seção 2.2.6, ilustrando-se o seu funcionamento, modelo de computação, de programação e sua exploração do paralelismo. Em seguida, são apresentadas algumas ferramentas de compilação e otimização disponíveis para uso em projetos diversos: a infraestrutura de compiladores LLVM, a ferramenta Clang e o GCC. E por fim, são apresentados os trabalhos relacionados.

2.1 Otimização de código

O compilador corresponde a um programa que recebe como entrada um código escrito em uma determinada linguagem de programação, chamada de linguagem fonte, e o traduz para um programa equivalente em uma outra linguagem, chamada de linguagem alvo, relatando ao usuário, se houver, a presença de erros no programa fonte. Normalmente, a linguagem fonte é uma linguagem de alto nível com grande capacidade de abstração, como C, C++ ou Java, e a linguagem alvo é a linguagem de máquina *assembly* que usa as instruções da arquitetura na qual será executada (LOUDEN, 2004).

Um compilador típico é composto pelas seguintes fases sequenciais: Análise Léxica, Análise Sintática, Análise Semântica, Geração de Código Intermediário e Geração de Código Alvo. Na Análise Léxica, os *tokens* presentes na entrada são identificados, agrupados, classificados e são enviados para a próxima etapa; a Análise Sintática, verifica a existência de erros de sintaxe no programa; a Análise Semântica, realiza a verificação de tipos, isto é, verifica-se a compatibilidade de tipos entre variáveis e expressões; na Geração de Código Intermediário, é produzido o código de três endereços; e na Geração de Código Alvo, é gerado o código *assembly* do processador escolhido para o compilador.

Geralmente, alguns códigos desenvolvidos não são ideais, pois necessitam de transformações para a obtenção de um melhor desempenho. Portanto, existe ainda uma etapa

do compilador chamada de otimização de código que corresponde a realização de algumas transformações no código. Essa etapa pode ser implementada logo após a geração de código intermediário, ou seja, as otimizações são independentes da arquitetura alvo ou após a geração de código alvo, ou seja, otimizações são dependentes da máquina alvo. As otimizações dependentes da máquina alvo são transformações no programa que necessitam de características da arquitetura, enquanto as otimizações independentes não precisam (AHO; SETHI; ULLMAN, 2007).

A implementação de otimizações demanda algumas propriedades que precisam ser obedecidas:

- A saída produzida por uma dada entrada não pode ser modificada, ou gerar um erro, que inicialmente não estava presente no código-fonte.
- Uma otimização precisa melhorar o código por algum fator mensurável, seja em tempo ou espaço. Porém, nem sempre isso é possível, algumas vezes uma retardação no programa é gerada, à proporção que a otimização melhora outros elementos.
- O esforço gasto na implementação da otimização precisa valer. Não faz sentido que o programador gaste esforço intelectual para desenvolver uma transformação visando a melhoria do código e o compilador gaste tempo adicional de compilação nos programas, se esse esforço não for recompensado quando o programa alvo for executado (AHO; SETHI; ULLMAN, 2007; LOUDEN, 2004).

As transformações em um programa podem ser em dois níveis: local ou global. Para um melhor entendimento desses níveis é necessário conhecer a definição de bloco básico ou trecho de código em linha reta. Bloco básico é um trecho de código, no qual:

- O primeiro comando inicia o bloco básico;
- Qualquer comando rotulado, ou que seja alvo de um comando de desvio corresponde ao início de um novo bloco básico;
- Qualquer comando de desvio, condicional ou incondicional, finaliza um bloco básico.

Isto é, blocos básicos são trechos de um programa, cujas instruções são executadas em ordem – linha reta – da primeira até a última (LOUDEN, 2004).

Assim, uma otimização em nível local acontece quando apenas informações do bloco básico são necessárias para realizar a transformação, enquanto uma otimização em nível global ocorre quando informações de mais de um bloco básico são necessárias. Algumas otimizações podem ser realizadas nos dois níveis. Geralmente, as transformações no nível local são realizadas primeiro (AHO; SETHI; ULLMAN, 2007).

Para a construção de um otimizador útil é necessária a identificação de oportunidades para otimização que sejam produtivas. Dependendo da finalidade do compilador, é que o conjunto de otimizações são escolhidas. Por exemplo, na construção de um compilador que será utilizado em um curso introdutório de programação não é necessária à aplicação de otimização, visto que esse compilador executará os programas raramente, desperdiçando o esforço para a implementação de otimização que poderia ser gasto em outro critério, como: a emissão de boas mensagens de erros auxiliando os participantes. Já um compilador para um programa que será executado várias vezes deve ser otimizado, como: programas de previsão do tempo (LOUDEN, 2004).

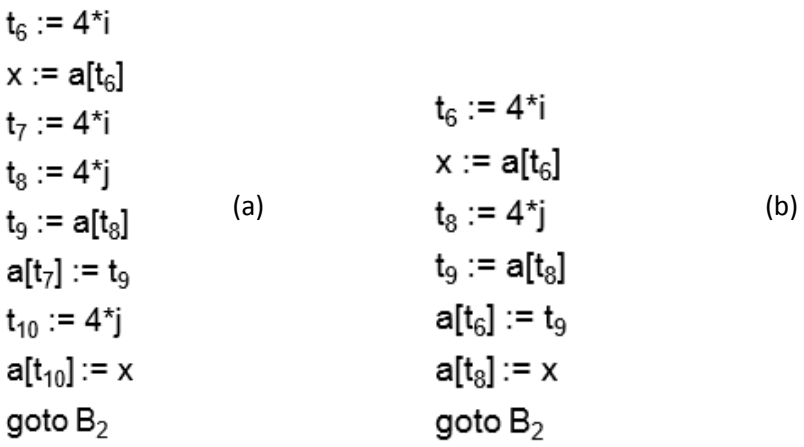
Quando se deseja otimizar algum programa, é necessário realizar uma análise na execução desse, a fim de identificar qual o trecho de código cujo tempo de execução é maior. Geralmente, os *loops* são ótimos candidatos para a otimização. Existem ferramentas que auxiliam o desenvolvedor nessa tarefa, registrando o perfil de execução do programa e permitindo a identificação dos trechos mais executados, como *gprof* (GPROF, 2016). Conhecendo esses trechos, aplica-se uma técnica de otimização mais adequada (RANGEL, 2000). Na literatura estão presentes muitas técnicas de otimização e algumas delas são objeto de estudo das próximas subseções.

2.1.1 Eliminação de Subexpressões Comuns

Essa técnica de otimização é aplicada quando se tem a presença de subexpressões comuns no código, isto é, quando uma mesma expressão aparece mais de uma vez em um trecho de código e o valor de suas variáveis não é alterado em todas as ocorrências (LOUDEN, 2004).

Sendo assim, pode-se evitar o recálculo dessa expressão e utilizar seu valor já calculado. Como se pode observar no exemplo da Figura 1 (a), as atribuições "t_7" e "t_{10}" armazenam as subexpressões comuns "4*i" e "4*j", respectivamente. Os valores dessas subexpressões não são modificados, podendo assim ser eliminados, como ilustra a Figura 1 (b), substituindo o valor de "t_7" por "t_6" e de "t_{10}" por "t_8".

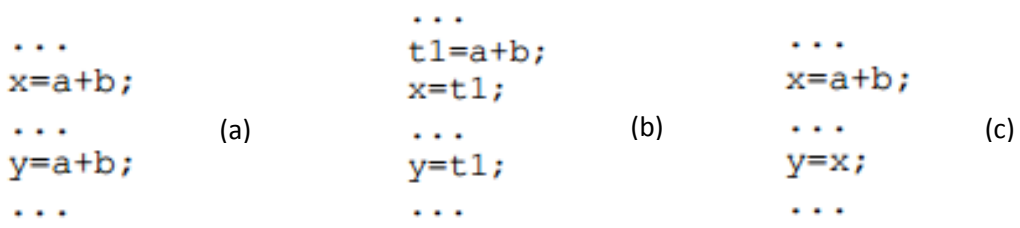
Figura 1 - Eliminação de subexpressões comuns: (a) antes da técnica (b) após a técnica



Fonte: Adaptado de AHO; SETHI; ULLMAN (2007)

Um outro exemplo está ilustrado na Figura 2 (a), a expressão "a + b" possui duas ocorrências, se os valores de "a" e de "b" não forem modificados ao longo do trecho de código, o valor dessa expressão pode ser guardada em uma variável temporária e utilizado posteriormente, como mostra a Figura 2 (b). Ou ainda, caso a variável "x" possua o valor do cálculo da subexpressão comum, ela pode ser utilizada, dispensando o uso da variável temporária, como se pode observar na Figura 2 (c).

Figura 2 - (a) Subexpressão comum; Eliminação de subexpressão comum: (b) Com a utilização de variável temporária e (c) Sem a utilização de variável temporária



Fonte: RANGEL (2000)

Os comandos que ocorrem entre as subexpressões comuns precisam ser analisados para garantir que os valores das variáveis da expressão não são alterados no decorrer do código. Caso esses comandos não estejam no mesmo bloco básico, é necessário realizar uma análise para identificar quais outros blocos básicos executam essas expressões.

2.1.2 Propagação de cópias

A transformação da propagação de cópias ocorre quando o código possui atribuição do tipo “f = g”, conhecidas por enunciados de cópia ou cópias. Essa otimização consiste em substituir “g” por “f”, sempre que possível, logo após a atribuição de cópia “f = g” (AHO; SETHI; ULLMAN, 2007).

O seguinte trecho de código pode ser melhorado por meio da eliminação da variável “x”:

```
x = t3
a[t2] = t5
a[t4] = x
goto B2
```

Para esse exemplo, tem-se a atribuição “x = t3” que consiste em uma cópia, após a aplicação da transformação da propagação de cópia, o trecho se transforma em:

```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

Posteriormente, tem-se a oportunidade da eliminação da atribuição “x = t3” (RANGEL, 2000).

2.1.3 Eliminação de código morto

Essa técnica corresponde à eliminação de um código que durante a execução do programa não pode ser alcançado, conhecido como código morto. Uma determinada variável está viva em um determinado ponto do programa, se seu valor for utilizado posteriormente, caso contrário estará morta a esse ponto. Existem algumas situações que o código morto pode ser identificado (RANGEL, 2000):

- Após uma instrução que encerra um programa ou uma função. Por exemplo: um comando após a instrução *return*.

```
int f (int x){
  return x++;              /* o incremento não será executado */
}
```

- Após um teste, cuja condição nunca será satisfeita.

```
#define DEBUG 0
...
if (DEBUG){
...                        /* este código não será executado */
}
```

- Após um comando de desvio que não é alvo de nenhum outro desvio.

```
    goto x;
    i = 3;
    ...
x:  ...
```

Geralmente um programador não insere código morto a seu programa. O código morto ocorre após algumas transformações no programa ocasionando código inalcançável.

Normalmente, a otimização da propagação de cópias transforma a atribuição de cópia em código morto. Para exemplificar, considere novamente o seguinte trecho de código já apresentado na seção anterior após a otimização da propagação de cópias:

```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

Aplicando a eliminação de código morto a esse código, o transforma em:

```
a[t2] = t5
a[t4] = t3
goto B2
```

Visto que a variável "x", tornou-se inalcançável, ou seja, código morto para esse ponto do programa, eliminando sua atribuição.

2.1.4 Renomeação de variáveis temporárias

Geralmente, durante a geração de código intermediário são inseridas variáveis temporárias que não são necessárias. Essa técnica consiste na renomeação de variáveis – temporárias ou não – que irão armazenar os valores temporários (RANGEL, 2000).

Por exemplo, durante a geração de código intermediário para a atribuição "x = a + b;" será gerado o seguinte trecho de código:

```
t1 = a + b;
x = t1;
```

Nesse exemplo, a variável "t_1" pode ser eliminada e a expressão armazenada em "x", o transformando em:

```
x = a + b;
```

Considere o seguinte código:

```
t1 = a + b;
x = t1;
y = (a + b)* c;
z = d + (a + b);
```

O código intermediário gerado é:

```
t1 = a + b;
x = t1;
t2 = a + b;
t3 = t2 * c;
y = t3;
t4 = a + b;
t5 = d + t4;
z = t5;
```

Aplicando a técnica de eliminação de subexpressão comum, algumas variáveis desnecessárias podem aparecer. Fazendo a eliminação das duas últimas cópias de “a + b”, o código após a transformação será:

```
t1 = a + b;
x = t1;
t2 = t1;
t3 = t2 * c;
y = t3;
t4 = t1;
t5 = d + t4;
z = t5;
```

Para esse exemplo, todas as variáveis temporárias podem ser eliminadas. Aplicando a técnica de renomeação de variáveis temporárias, o código resultante será:

```
x = a + b;
y = x * c;
z = d + x;
```

2.1.5 Transformações algébricas ou Uso de identidades algébricas

As otimizações algébricas possuem três classes importantes, são elas: transformações algébricas simples, redução de capacidade e transposição para constantes.

A primeira classe corresponde a aplicação de transformações com base em propriedades algébricas, como: comutatividade, associatividade, identidade, etc. Para exemplificar temos a atribuição “x = a + b* c;” como a soma possui a propriedade comutativa, essa atribuição pode ser transformada em “x = b * c + a;”. Isso corresponde a trocar o código (RANGEL, 2000):

```
Load b
Mult c
Store t1
Load a
Add t1
Store x
```

Pelo seguinte código

```
Load b
Mult c
Add a
Store x
```

Eliminando a variável "t_1" e todas as instruções que a manipulam.

Outro exemplo, aplicando a propriedade da associatividade, pode-se realizar a troca de "x = (a + b) + (c + d);" por "x = ((a +b) + c) + d);", que corresponde ao mesmo que transformar o código:

```
Load a
Add b
Store t1
Load c
Add d
Store t2
Load t1
Add t2
Store x
```

Em

```
Load a
Add b
Add c
Add d
Store x
```

Eliminando as variáveis temporárias "t_1" e "t_2".

Outro exemplo ainda para a primeira classe, é a aplicação de identidades aritméticas simples, como:

```
x + 0 = 0 + x = x
x – 0 = x
x * 1 = 1 * x = x
x / 1 = x
```

A segunda classe corresponde a redução de capacidade, no qual um operador mais caro é substituído por um mais barato, como em:

```
x^2 = x * x
2.0 * x = x + x
x/2 = x * 0.5
```

E a terceira classe consiste na transposição para constantes. Cujas expressões são avaliadas em tempo de compilação e seus valores substituídos. Por exemplo, a expressão "2*3.14" é substituída por "6.28".

2.1.6 Dobramento de constantes

Essa técnica pode ser aplicada quando aparecem expressões ou subexpressões compostas de valores constantes no código a ser transformado. Esses valores podem ser calculados durante a compilação evitando seu recálculo repetidas vezes. Por exemplo, no código (RANGEL, 2000):

```
#define N 100
...
while (i<N-1){
...
}
```

O cálculo de "N – 1" não precisa ser calculado repetidamente, seu valor pode ser pré-calculado e substituído por "99".

2.1.7 Otimizações de *loop*

Em um programa, o maior processamento gasto é consumido em *loops* ou laços de repetição. Dessa maneira, é recomendável que somente as operações realmente necessárias estejam presentes dentro de *loops* (FUJII, 2012).

Os *loops* são os locais mais importantes para as otimizações, principalmente os *loops* internos, visto que é onde é gasto o maior tempo de execução. Visando a diminuição desse tempo, pode-se diminuir o número de instruções do *loop* mais interno, porém há um aumento na quantidade de código fora do *loop* (RANGEL, 2000).

Existem várias técnicas que visam otimizar *loops*, as mais importantes segundo Aho, Sethi e Ullman (2007) são: movimentação de código e redução de força ou de capacidade. A primeira otimização consiste em transferir o código para fora do *loop*. Já a segunda, substitui uma operação mais cara e complexa por outra mais barata e simples. Enquanto para Down e Severance (1998) são: desvio condicional e desenrolamento de laços. Essas técnicas são expostas nas próximas subseções.

2.1.7.1 Movimentação de código

Essa transformação diminui a quantidade de código no laço. E consiste em transferir uma expressão, que independentemente do número de iterações do laço (computação laço-

invariante) sempre gera o mesmo resultado, para antes do mesmo. A expressão "antes do laço" assume a existência de uma entrada para o laço (RANGEL, 2000). Por exemplo, em:

```
while (i <= limite - 2)  // o valor de limite não é modificado
```

A expressão "`limite - 2`" possui o mesmo valor independente das iterações do *loop*. Assim, aplicando a técnica da movimentação de código, resultará no seguinte código:

```
t = limite - 2
while (i <= t)  // os valores de limite ou de t não são modificado
```

Antes da aplicação dessa técnica é necessário verificar se:

- O lado direito da atribuição é composto por constantes ou por variáveis cujos valores não se alteram dentro do *loop*.
- Nenhum valor da variável do lado esquerdo da atribuição, dentro ou fora do *loop*, poderá ter um valor diferente do valor que tinha antes no código original.

O valor da atribuição deverá ser o mesmo do código inicial, após a aplicação dessa técnica. Porém, dependendo da quantidade de iterações que o *loop* executará, essa otimização poderá piorar o programa. Visto que, no caso em que nenhuma iteração é executada, o comando dentro do *loop* não seria executado nenhuma vez, já com o comando fora do *loop* ele será executado pelo menos uma vez.

Para exemplificar a aplicação dessa técnica, observe o seguinte exemplo:

```
for (i=0; i< N; i++){
     k = 2 * N;
     f(k*i);
}
```

Após a otimização, o código é transformado em:

```
k = 2*N;
for (i = 0;i < N; i++){
     f(k*i);

}
```

Visto que, o valor da variável 'k' sempre será o mesmo independentemente do número de iterações. Sendo movimentado para fora do *loop*. Esse comando sempre será executado uma vez.

2.1.7.2 Redução de força ou de capacidade

Essa técnica consiste na substituição de operações mais caras por operações mais baratas, cujos resultados são os mesmos. Quando um programador deseja calcular o comprimento da concatenação de duas cadeias em C, geralmente ele utiliza a linha de código

"`strlen(strcat(s1, s2))`" para isso. Essa operação pode ser substituída pela soma dos comprimentos das cadeias, através da linha de comando "`strlen(s1) + strlen(s2)`".

Um outro exemplo para a aplicação dessa técnica é no cálculo do quadrado de um número. A linha de comando para realizar essa operação consiste em "`pow(x, 2)`". Na execução desse comando é calculado o valor da seguinte expressão: "$e^{2.0 * \ln x}$". Esse cálculo corresponde a uma operação cara para o compilador e pode ser substituído pela multiplicação "`x * x`" produzindo o mesmo resultado (RANGEL, 2000).

Outros exemplos que essa técnica pode ser aplicada, é na troca de:

- Exponenciação por multiplicação: a operação de exponenciação não corresponde a uma operação nativa do processador, sendo assim para executa-la é necessária a chamada de uma função. Uma solução é substitui-la por sua operação manual que corresponde à multiplicação do valor por ele mesmo a quantidade de vezes apresentado em sua potência, como em: $10^5 \Rightarrow 10*10*10*10*10$.
- Divisão de valor real por multiplicação inversa: para executar a operação de multiplicação, menos recursos do processador são utilizados, comparado à operação de divisão. Para realizar operações com valores discretos é recomendável a substituição da operação de divisão pela multiplicação, mantendo a relação do valor discreto com o resultado do cálculo. Por exemplo, a / 2 ⇒ a * 0,5, as duas equações possuem o mesmo resultado (FUJII, 2012).

2.1.7.3 Desvio condicional em laços de repetição

Em laços de repetição deve-se evitar a utilização de operações com desvio condicional, visto que esse comando consome instruções extras. Em determinados casos, os desvios condicionais podem ocorrer (FUJII, 2012):

- Desvio condicional invariante: o resultado da condição será sempre o mesmo, podendo ser descartado;
- Desvio condicional independente do laço de repetição: a condição é independente do valor testado e do índice da iteração. A transformação pode ser realizada através do desenrolamento de laço.
- Desvio condicional dependente do laço de repetição: a condição é dependente de algum valor que se modifica a cada iteração do laço. Para esse caso, não é possível otimizar.

2.1.7.4 Desenrolamento de laços

Esta técnica consiste em desdobrar o laço de repetição, ou seja, realizar as iterações de um laço manualmente, removendo a operação de incremento do índice e economizando a alteração do endereço do apontador da pilha a cada iteração (FUJII, 2012).

O trecho de código abaixo contém um exemplo de soma de duas matrizes 4x4. Para realizar essa operação são utilizados dois laços de repetição, um para as linhas e o outro para as colunas:

```
for (i=0; i<4; i++)
      for (j=0; j<4; j++)
            m(i, j)= m1(i, j) + m2(i, j)
```

Após aplicada a técnica de desenrolamento de laço, o trecho de código anterior é transformado no seguinte código:

```
for (i=0; i<4; i++){
      m(i,0) = m1(i,0) + m2(i,0)
      m(i,1) = m1(i,1) + m2(i,1)
      m(i,2) = m1(i,2) + m2(i,2)
      m(i,3) = m1(i,3) + m2(i,3)
```

Como pode-se observar, o laço referente as colunas foi removido, e para cada coluna inserido um comando manualmente. Esse processo também pode ser realizado com o laço referente às linhas, totalizando 16 comandos de atribuição (FUJII, 2012).

Essa técnica é bastante comum e utilizada na maioria dos compiladores atuais (DOWN; SEVERANCE, 1998).

2.1.8 Eliminação de sub-rotinas

Uma chamada a uma sub-rotina inclui *overhead* de armazenar endereços nos registradores, além da leitura e manipulação de argumentos (DOWN; SEVERANCE, 1998). Caso uma sub-rotina seja invocada diversas vezes para a realização de uma simples tarefa, prejudicará o desempenho do programa e isso não pode acontecer (FUJII, 2012). Dessa forma, há a otimização da eliminação de sub-rotinas.

A Figura 3 ilustra um exemplo de código que contém uma função chamada ‘soma’, que é utilizada para auxiliar na soma de matrizes. Essa função é invocada para realizar a soma dos elementos correspondentes aos índices “i” e “j”. Seu resultado é armazenado na variável matriz

com os respectivos valores de “i” e “j”. Se a matriz for muito grande, seu tempo de execução aumentará devido ao *overhead* de chamadas de sub-rotinas (FUJII, 2012).

Figura 3 - Trecho de código que possui uma função que auxilia na soma de matrizes

```
1 int soma(int M1[LINHA][COLUNA], int M2[LINHA][COLUNA], int i, int j) {
2   int resultado;
3   resultado = M1[i][j] + M2[i][j];
4   return resultado;
5 }
6 ...
7 for(int i=0; i<LINHA; i++)
8   for(int j=0; j<COLUNA; j++)
9     matriz[i][j] = soma(M1,M2,i,j);
```

Fonte: FUJII (2012)

Uma solução para esse caso é não utilizar a função e seu cálculo ser realizado diretamente, como mostra a Figura 4.

Figura 4 – Eliminação da sub-rotina soma da Figura 3

```
1 for(int i=0; i<LINHA; i++)
2   for(int j=0; j<COLUNA; j++)
3     matriz[i][j] = M1[i][j] + M2[i][j];
```

Fonte: FUJII (2012)

2.1.9 Macro

Macro corresponde a uma alternativa para a utilização de pequenas funções. Que constituem pequenos procedimentos substituídos em tempo de compilação. Na primeira passagem do compilador pelo programa, as macros são executadas, diferentemente das sub-rotinas que são incluídas na etapa de ligação (DOWN; SEVERANCE, 1998). Sempre que o compilador se deparar com um trecho de código que corresponde a uma macro, esse código é substituído.

As macros na linguagem de programação C são definidas através do comando “`#define`”. A Figura 5 contém a definição da macro para calcular a média aritmética de dois números (linha 1), chamada de `calcula_media`, que recebe dois argumentos. Quando essa

macro é chamada (linha 6), o compilador faz a substituição do padrão da macro pela sua definição, resultando em "`media = (a + b) / 2;`" (FUJII, 2012).

Figura 5 - Trecho de código contendo uma macro

```
1  #define calcula_media(x,y) (x+y)/2
2  main()
3  {
4    float a=10, b=20;
5    float resultado;
6    media = calcula_media(a,b);
7    printf(''%fz\n'',media);
8  }
```

Fonte: FUJII (2012)

2.1.10 Função *inlining*

A substituição de uma chamada da função pelo código da função é chamada de *inlining*. Essa otimização é utilizada quando se tem um código grande para ser definido em uma macro (WADLEIGH; CRAWFORD, 2000). Ao realizar essa substituição haverá uma maior probabilidade de otimizações no código do programa, e ainda será eliminado o *overhead* da chamada da função (armazenamento e restauração dos registradores), melhorando assim o desempenho do programa.

Porém a utilização dessa otimização aumenta o tamanho do código-fonte do programa, seu tempo de compilação e pode diminuir o nível de compreensão do código. Em algumas sub-rotinas, o uso de *inlining* não é suportado, como em (FUJII, 2012):

- Sub-rotinas recursivas;
- Sub-rotinas que utilizam variáveis locais estáticas;
- Sub-rotinas com número de argumentos indefinidos.

2.1.11 Otimização *peephole*

Essa técnica é aplicada ao código objeto e conhecida como otimização de janela (MORGAN, 1997). Essa otimização examina poucas instruções contíguas na busca por padrões, em que exista a possibilidade de melhora no código. Alguns dos padrões são:

- Instruções de desvio para instrução seguinte;
- Instruções de desvio para uma instrução de desvio condicional;
- Conjunto de desvios condicionais e incondicionais que podem ser substituídos por uma instrução de desvio condicional para a condição oposta (por exemplo: desvio condicional caso a condição seja verdadeira em conjunto com um desvio incondicional (JT/J) em alguns casos, pode ser substituída por um desvio condicional caso a condição seja falsa (JF)).

Como algumas instruções podem ser eliminadas, geralmente, a implementação da otimização requer a renumeração das instruções, ou ainda, a substituição de instruções por outras mais curtas. Mais de uma varredura no código é necessária, pois uma mudança em uma instrução pode criar oportunidades de otimização para as seguintes. Por exemplo, se uma instrução for substituída por uma mais curta ou houver a eliminação de uma instrução inútil, a distância entre uma instrução de desvio e a instrução alvo pode diminuir, substituindo um desvio longo por um desvio curto.

Considere a seguinte sequência de instruções:

```
 1:[ := a c  - ]
 2:[ J  - -  3 ]
 3:[ JT a -  7 ]
 4:[ J  - -  5 ]
 5:[ JT b -  7 ]
 6:[ J  - -  9 ]
 7:[ := c a  - ]
 8:[ J  - - 11 ]
 9:[ := c b  - ]
10:[ J  - - 11 ]
11:[ J= a b 13 ]
12:[ J  - -  1 ]
```

Nessa sequência são identificados os padrões:

- Desvios para a instrução seguinte em 2, 4 e 10, essas instruções podem ser eliminadas;
- Na instrução 5, há um desvio para a instrução 7, se a condição for verdadeira, caso contrário, a instrução 6 é executada e nela há um desvio para a instrução 9. O teste JT da instrução 5 é invertido

```
5′:[ JF b -  9 ]
6′:[ J  - -  7 ]
```

- Podendo eliminar a instrução 6’ que corresponde a um desvio para a instrução imediatamente seguinte;
- O teste J da instrução 11 também pode ser invertido

```
11′:[ J≠ a b  1 ]
12′:[ J  - - 13 ]
```

- A instrução 12’ pode ser eliminada.

Após essas transformações, a sequência de instruções é:

```
 1:[ := a c  - ]
 3:[ JT a -  7 ]
 5’:[ JF b -  9 ]
 7:[ := c a  - ]
 8:[ J  - - 11 ]
 9:[ := c b  - ]
11’:[ J≠ a b  1 ]
```

E após a renumeração, o resultado será:

```
1:[ := a c  - ]
2:[ JT a -  4 ]
3:[ JF b -  6 ]
4:[ := c a  - ]
5:[ J  - -  7 ]
6:[ := c b  - ]
7:[ J≠ a b  1 ]
```

Houve uma redução de 5 instruções, ou seja, uma redução de mais de 40% do número de instruções. Os resultados dessa otimização geralmente são significativos (RANGEL, 1999).

2.2 Arquiteturas Paralelas

Von Neumann propôs uma máquina composta de uma memória, uma Unidade Lógica e Aritmética (ULA), uma única Unidade Central de Processamento (CPU – *Central Processing Unit*) e uma unidade de controle (HENNESSY; PATTERSON, 2014). Sendo conhecida por máquina de Von Neumann.

A máquina de Von Neumann segue um modelo em que os programas e dados são armazenados em uma memória central e seu processador é composto por um caminho de dados e um controle, funcionando como uma máquina de estados do tipo busca instrução, decodifica, executa, armazena o resultado e volta a buscar uma nova instrução.

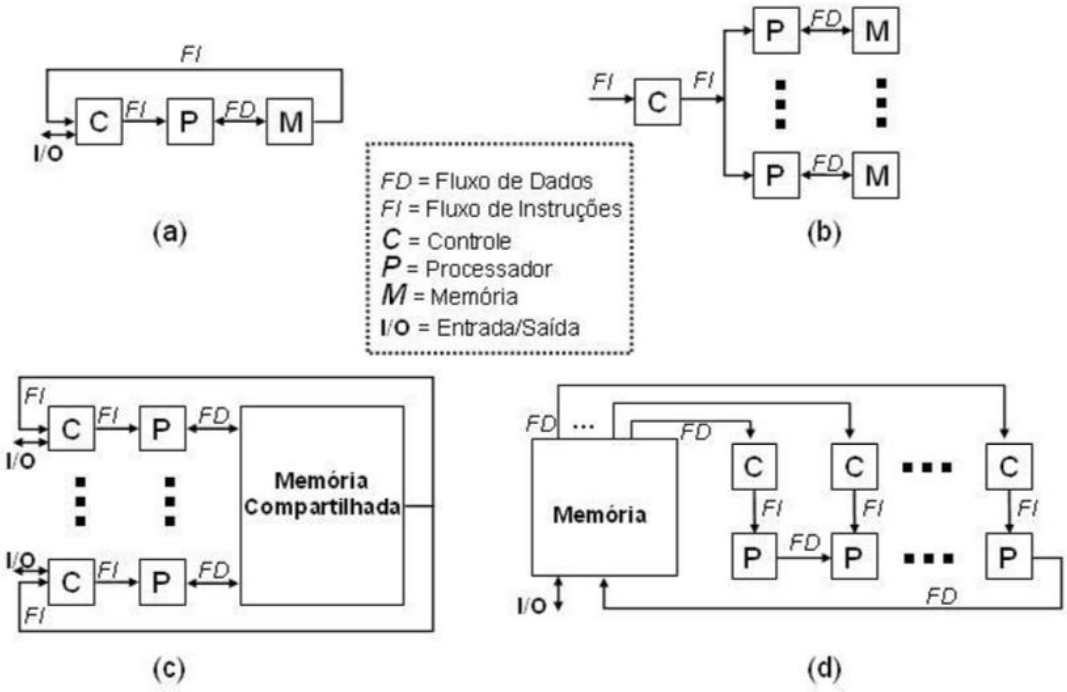
As arquiteturas convencionais que surgiram posteriormente foram baseadas nessa máquina. Existem diferentes classificações para essas arquiteturas, a mais aceita é a taxonomia de Flynn (FLYNN, 1972). Nela, o paralelismo que corresponde a execução simultânea de várias sub-tarefas visando a diminuição do tempo de execução da tarefa global, é classificado de acordo com duas características: a sequência de instruções e de dados. Sua classificação apresenta quatro categorias, conforme apresenta-se na Figura 6:

- SISD (*Single Instruction Stream, Single Data Stream*): existe um fluxo único de instruções e um fluxo único de dados, apresentando um processamento sequencial. Nessa categoria é possível a exploração do paralelismo em nível de instrução. Nesse modelo, o fluxo

de instrução alimenta a unidade de controle que ativa o processador para realizar sua execução utilizando o fluxo único de dados que são lidos da memória, na qual os resultados também são armazenados. Exemplo: arquiteturas que seguem a máquina de Von Neumann.

- SIMD (*Single Instruction Stream, Multiple Data Stream)*: composta por vários processadores e uma memória de dados para cada um. Os processadores executam diferentes fluxos de dados paralelamente em um único fluxo de instruções, determinado pelo controlador. Exemplo: computador vetorial.
- MISD (*Multiple Instruction Streams, Single Data Stream*): nessa categoria um único conjunto de dados é utilizado por vários fluxos de instruções. Essa categoria não possui processador comercial e são conhecidas como máquinas sistólicas.
- MIMD (*Multiple Instruction Streams, Multiple Data Stream*): correspondem aos computadores verdadeiramente paralelos, pois operam múltiplos fluxos de instruções sobre vários fluxos de dados. Exemplos: redes em *chip* e processadores multinúcleos.

Figura 6 - Taxonomia de Flynn a) SISD; b) SIMD; c) MIMD com memória compartilhada; d) MISD



Fonte: REGO (2006)

As arquiteturas que seguem o paradigma da máquina de Von Neumann são chamadas de arquiteturas convencionais. Visando a melhoria no desempenho, algumas arquiteturas foram desenvolvidas e não seguem esse paradigma, sendo classificadas como arquiteturas não

convencionais. A arquitetura IPNoSys é um exemplo dessas arquiteturas e será apresentada na próxima seção.

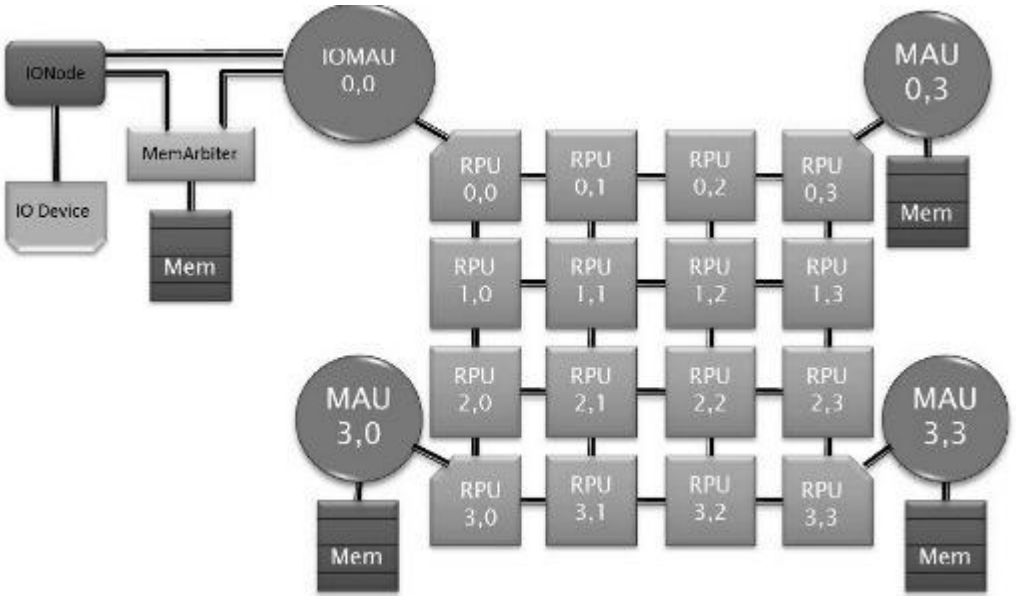
2.2.1 Arquitetura IPNoSys

A Arquitetura IPNoSys é um processador de propósito geral baseado em redes em chip. Nela, os dados e as instruções circulam pela rede em forma de pacotes, que são processados durante seu trajeto. A IPNoSys explora as vantagens existentes nas NoCs, como paralelismo, reusabilidade e escalabilidade (ARAUJO; OLIVEIRA; SILVA, 2009). Nas subseções abaixo são apresentados o funcionamento da arquitetura, seu modelo de computação, de programação e como essa arquitetura explora o paralelismo.

2.2.1.1 Funcionamento e Modelo de Computação

A Figura 8 ilustra a Arquitetura IPNoSys com topologia do tipo malha 2D de dimensão quadrada, ou seja, apresenta número de linhas e colunas iguais. O tamanho de sua malha pode ser aumentada sem a perda de desempenho, tornando o processador escalável. Nessa arquitetura, o formato de pacotes é utilizado, contendo dados e instruções (ARAUJO, 2012). A estrutura apresentada na Figura 7 foi utilizada neste trabalho de Dissertação e corresponde a implementação proposta por ARAUJO (2012).

Figura 7 - Arquitetura IPNoSys

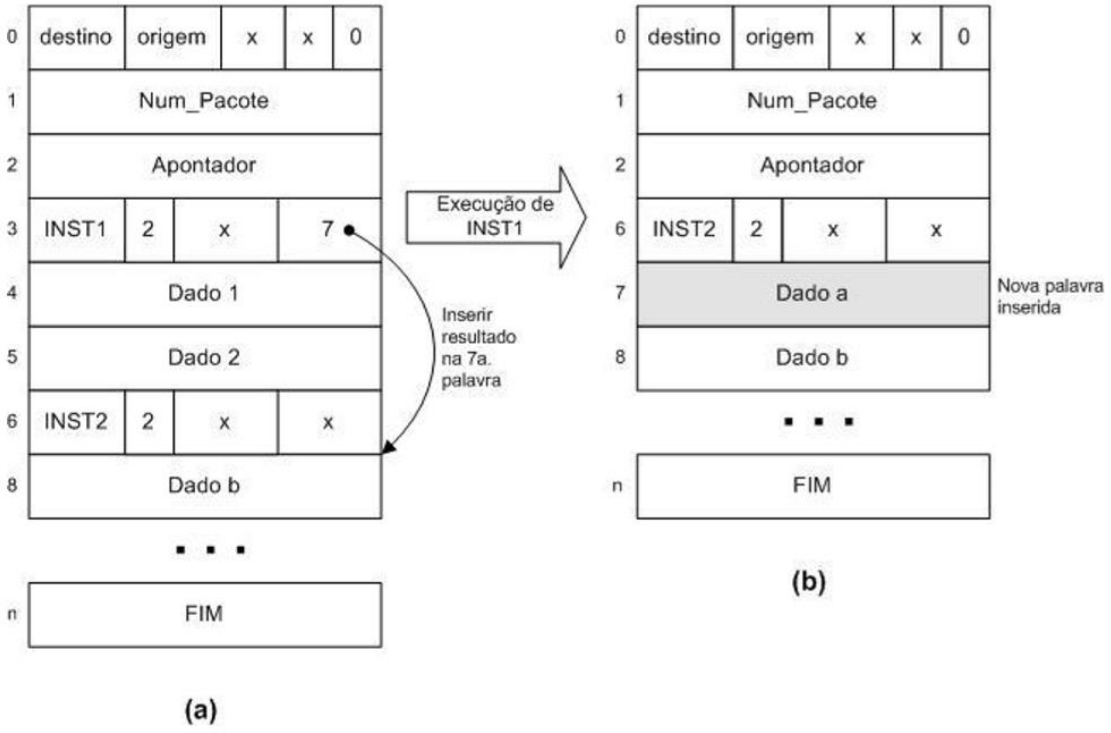


Fonte: ARAUJO (2012)

Seus roteadores são chamados de RPU (*Routing and Processing Unit*) por possuírem a capacidade de roteamento e de processamento. Os pacotes que circulam pela rede são roteados e uma parte do pacote é processada em cada RPU por onde passa. Dessa forma, as RPUs executam instruções enquanto encaminha os pacotes até o seu destino.

Enquanto o pacote caminha pela rede, cada RPU executa no mínimo uma instrução que encontra-se no início do pacote. Ao realizar a execução dessa instrução, ela e seus respectivos operandos deixam de pertencer ao pacote. O resultado da execução de uma instrução poderá gerar dados que servirão de operandos para instruções posteriores. Assim, o que sobrou do pacote será encaminhado para a próxima RPU, onde esse processo se repete (FERNANDES; SILVA; KREUTZ, 2010). Esse processo corresponde ao modelo de computação da IPNoSys, que está ilustrado na Figura 8.

Figura 8 – Pacote: (a) antes da execução de INST1 (b) depois a execução de INST1



Fonte: ARAUJO (2008)

Na Figura 8 (a), ilustra-se um determinado pacote composto por algumas instruções, cuja primeira instrução é chamada de INST1. Ao executar essa instrução, é produzido um resultado para a sétima palavra do pacote, onde esse resultado (Dado a) é inserido. O pacote após a execução da instrução INST1, ilustrado na Figura 8 (b), é encaminhado para a próxima RPU sem essa instrução e seus respectivos operandos, e a próxima instrução que será executada é INST2, cujo processo se repete (ARAUJO, 2008).

Ainda observando a estrutura da IPNoSys da Figura 7, pode-se identificar outros elementos, são eles: MAU, IOMAU, IONode e MemArbiter. A unidade chamada de MAU (*Memory Access Unit*) possui a função de introduzir os pacotes na rede, além de controlar o acesso de escrita e leitura na memória. A IPNoSys possui quatro MAUs, que são conectadas a um módulo de memória. Existe uma MAU diferente, chamada de IOMAU, que compartilha a memória conectada com o IONode, com o controle de acesso pelo MemArbiter. O IONode é um sistema de entrada e saída, que faz acesso direto com a memória. Assim, dados são trocados entre os dispositivos de entrada e saída e a memória através do IONode (ARAUJO, 2012).

2.2.1.2 Modelo de Programação

A IPNoSys possui um modelo de programação próprio. Nesse modelo, o programador é o responsável por particionar seu programa, fazer o mapeamento, indicar como será feita a comunicação e a sincronização dessas partes (ARAUJO, 2012).

Essa arquitetura possui um conjunto contendo 32 instruções (Anexo A). Cada instrução é executada por um elemento da IPNoSys. Assim, as instruções: aritméticas (4), lógicas (4), deslocamento (2), condicional (6), incondicional (1), auxiliar (2) e chamada de procedimento (2) são executadas pela RPU. Já as instruções: de acesso à memória (3), sincronização (4) e entrada/saída (2), são executadas pela MAU. Estando todas disponíveis para serem utilizadas pelo programador, com exceção das instruções de sistema (2) que funcionam como mecanismo de sistema operacional.

Seu modelo de programação baseia-se na descrição dos programas utilizando pacotes, que são injetados e executados na arquitetura. Cada pacote é formado por um conjunto de instruções que são empilhadas de acordo com a dependência de dados. Uma instrução somente será executada quando estiver no topo da pilha, ou seja, quando estiver no início do pacote juntamente com seus operandos.

Existem quatro tipos de pacotes: controle, regular, interrompido e *caller*. Os pacotes de controle apresentam apenas uma instrução que é executada em uma MAU. Os pacotes regulares são compostos por instruções que são executadas em qualquer RPU. Já os pacotes interrompidos, correspondem a pacotes regulares que esperam por um evento de E/S. E por fim, o pacote *caller* também é um pacote regular que apresenta uma chamada a uma função.

Dentre as instruções pertencentes a esse conjunto de instruções da IPNoSys, oito foram apresentadas com detalhes, visto que são necessárias para o entendimento deste trabalho. Detalhes das demais instruções podem ser vistos em (Anexo A) e em ARAUJO (2012).

A única instrução contida em um pacote de controle e executada em uma RPU corresponde a instrução RELOAD. Essa instrução realiza a inserção do valor carregado da memória anteriormente solicitado pela instrução LOAD. Quando uma instrução LOAD é identificada, a RPU cria um pacote de controle contendo essa instrução e envia para uma MAU executar. Assim, a execução do pacote é paralisada até que o valor solicitado chegue a RPU através da instrução RELOAD. Dessa forma, o tempo de espera corresponde a enviar o LOAD até a MAU que carrega o dado e o tempo desta MAU até a RPU trazendo o dado em um RELOAD.

A instrução SEND envia um valor para a MAU que faz a inserção em um pacote que esteja na memória durante sua injeção. A instrução STORE armazena um determinado valor em uma posição de memória.

A instrução EXEC faz a injeção de um pacote que está armazenado na memória. A instrução SYNEXEC somente injetará um determinado pacote quando os sinais de sincronismos dos demais pacotes chegarem a MAU. Esses sinais de sincronismos são executados através da instrução SYNC.

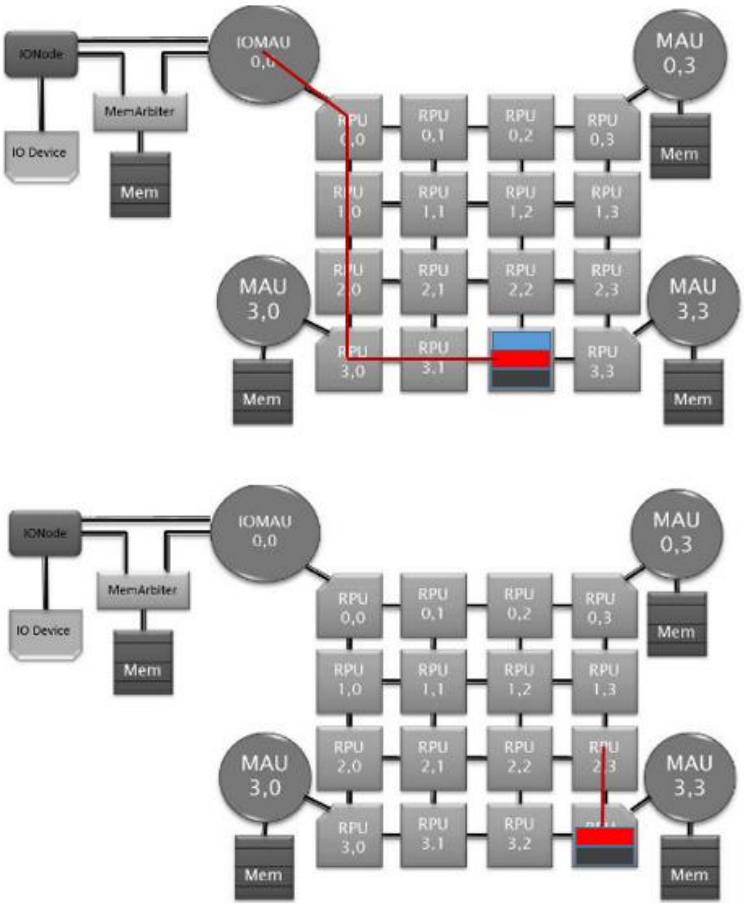
Por fim, a instrução COPY é utilizada quando se deseja fazer cópias de um determinado valor para ser usado como operando de outras instruções. Pode-se realizar até duas cópias de um determinado valor em cada instrução COPY.

As instruções que são executadas pelas MAUs representam um custo extra para a IPNoSys, que dependendo da distância entre a RPU que decodificou a instrução e a MAU destino podem ser significativos. Visto que essas instruções necessitam de um tempo para serem transportadas até a MAU, especialmente a instrução LOAD que, como visto anteriormente, realiza uma espera dobrada, pois é gasto tempo para realizar uma requisição da posição de memória e também para esperar o retorno desse valor. Dessa forma, instruções *LOAD* precisam ser usadas com muita cautela ou mesmo evitadas. Uma solução para isso é a sua substituição por uma instrução *COPY*. Eliminando assim o tempo de espera que a instrução de carregamento gastaria e ainda economizando no tamanho do código, visto que a instrução *COPY* guarda duas cópias de um determinado valor que podem ser utilizados como dois operandos de instruções posteriores, já o valor da instrução *LOAD* só pode ser utilizado como um operando.

A Figura 9 ilustra o funcionamento das instruções *LOAD* e *COPY*. Nessa figura, o retângulo maior representa o pacote que está trafegando pela rede e localiza-se na RPU 3,2. No início desse pacote há uma instrução *LOAD* (retângulo azul da Figura 9 (a)) que será executada pela MAU do canto superior esquerdo. Essa instrução é transportada até a MAU passando por 5 RPUs (linha em vermelho). Durante esse trajeto, o pacote é paralisado e aguarda o resultado

dessa instrução. Esse resultado realiza o trajeto inverso até chegar a RPU 3,2. Apresentando uma espera dobrada. Após isso, a instrução é removida do pacote e ele é movido para a próxima RPU. Na Figura 9 (b), o início do pacote contém uma instrução *COPY*, que é executada pela RPU corrente, nesse exemplo a RPU 3,3. Depois de sua execução o pacote caminha (linha em vermelho) para a próxima RPU.

Figura 9 – Funcionamento das instruções (a) *LOAD* e (b) *COPY*



(a)

(b)

Fonte: AUTORIA PRÓPRIA (2016)

Seu modelo de programação utiliza a linguagem *assembly* PDL (*Package Description Language*), que consiste em uma representação abstrata dos pacotes e foi desenvolvida para auxiliar o programador na implementação de aplicações em um nível intermediário entre linguagens de alto nível e o formato de pacotes.

O código PDL utiliza um montador para ser processado e traduzido para o código objeto da IPNoSys. A Figura 10 ilustra a gramática utilizada para definir a PDL.

Figura 10 - Gramática da PDL utilizada pela IPNoSys

```
program       : decProgram decData  package+  END_PROGRAM
decProgram    : PROGRAM ID
decData       : DATA (ID ('=' INT)?)*
package       : (packageId | functionId) mauAddress instructions EXIT? END;
packageId     : PACKAGE ID
functionId    : FUNCTION ID
mauAddress    : ADDRESS MAU
instructions  : (instruction)+
instruction   : label? operation
label         : ID ':'
operation     : CONDITIONAL_OPERATOR result ';' operand operand ';'
              | INCONDITIONAL_OPERATOR result ';'
              | SINGLE_OPERATOR ';'
              | UNARY_OPERATOR result result? ';' operand ';'
              | BINARY_OPERATOR result result? ';' operand operand ';'
              | MULTIPLE_OPERATOR result result? ';' (operand)+ ';'
              | IO_OPERATOR result ';' operand operand operand ';'
              | FUNCTION_OPERATOR result ';' (operand)+ ';'
              | RETURN_FUNCTION ';' operand ';'
result        : ID | INT | MAU | IOMAU
operand       : ID | INT | ID ',' ID | ID ',' INT | INT ',' ID | INT ',' INT | ID '=' INT
```

Fonte: ARAUJO (2012)

Na PDL, existe a diferenciação entre palavras maiúsculas e minúsculas. Essa linguagem possui também algumas palavras reservadas que estão ilustradas na definição de *tokens* da linguagem na tabela da Figura 11.

Figura 11 – *Tokens* da PDL

```
PROGRAM                 : 'PROGRAM'
FUNCTION                : 'FUNCTION'
DATA                    : 'DATA'
PACKAGE                 : 'PACKAGE'
ADDRESS                 : 'ADDRESS'
MAU                     : 'MAU_0' | 'MAU_1' | 'MAU_2' | 'MAU_3'
IOMAU                   : 'IOMAU'
EXIT                    ; 'EXIT'
END                     : 'END'
END_PROGRAM             : 'END_PROGRAM'
CONDITIONAL_OPERATOR    : 'BE' | 'BNE' | 'BL' | 'BG' | 'BLE' | 'BGE'
INCONDITIONAL_OPERATOR  : 'JUMP'
SINGLE_OPERATOR         : 'NOP'
UNARY_OPERATOR          : 'NOT' | 'LOAD' | 'EXEC' | 'SYNC' | 'COPY'
BINARY_OPERATOR         : 'ADD' | 'SUB' | 'MUL' | 'DIV' | 'AND' | 'OR' | 'XOR' | 'RSHIFT' | 'LSHIFT' | 'SEND'
MULTIPLE_OPERATOR       : 'STORE' | 'SYNEXEC'
IO_OPERATOR             : 'IN' | 'OUT'
FUNCTION_OPERATOR       : 'CALL'
RETURN_FUNCTION         : 'RETURN'
ID                      : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
INT                     : '0'..'9'+
COMMENT                 :  '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
WS                      :  ( ' ' | '\t' | '\r' | '\n' ) {$channel=HIDDEN;}
```

Fonte: ARAUJO (2012)

De acordo com a gramática apresentada na Figura 10, todo programa inicia com a palavra reservada PROGRAM seguido do nome do programa que é determinado pelo programador. Em seguida, tem-se a palavra reservada DATA, na qual as variáveis do programa

são definidas, sendo inicializadas ou não. A inicialização é realizada incluindo-se após o identificador da variável o sinal de igualdade e um valor inteiro. A palavra DATA é obrigatória mesmo que o programador não declare nenhuma variável.

Logo depois é realizada a declaração de um ou mais pacotes para o programa. Para isso é utilizada a palavra reservada PACKAGE ou a palavra FUNCTION seguida do nome determinado pelo programador. Após isso, é definido o endereço da MAU que injetará o pacote na rede, por meio da palavra reservada ADDRESS seguida do endereço da MAU definido pelo programador. IPNoSys possui 4 MAUs que são identificadas pelas palavras reservadas: MAU_0 (canto superior esquerdo), MAU_1 (canto superior direito), MAU_2 (canto inferior esquerdo) e MAU_3 (canto inferior direito).

Entre a declaração de um pacote e o final desse, por meio da palavra reservada END, são descritas as instruções que definem o que o pacote faz. Antes da palavra END, pode-se utilizar a palavra EXIT, que define o fim do programa. Porém, sua presença é obrigatória em apenas um dos pacotes que constituem o programa. Com essa instrução, o programador identifica o fim da execução do programa, visto que um programa é formado por um ou mais pacotes, que podem ser executados várias vezes e na ordem definida para execução.

Na Figura 12, apresenta-se um exemplo genérico de um programa PDL. Suas instruções são escritas utilizando-se o identificador da instrução corrente, seguida por 0, 1 ou 2 indicadores de posições, cujos resultados serão inseridos, e finalizando com ponto e vírgula. Em seguida, seus operandos são definidos e também deve finalizar com ponto e vírgula. As posições de resultados e os operandos são definidos por meio de identificadores (linhas 10 e 14) ou valores inteiros (linha 13).

Para o caso das posições de resultados, um identificador deve ser uma palavra definida pelo programador e usado como operando de uma instrução posterior. Um identificador de resultado definido como operando de uma instrução anterior é vista como um erro pelo *assembler*, já que os resultados não podem ser inseridos em posições anteriores àquela da instrução em execução.

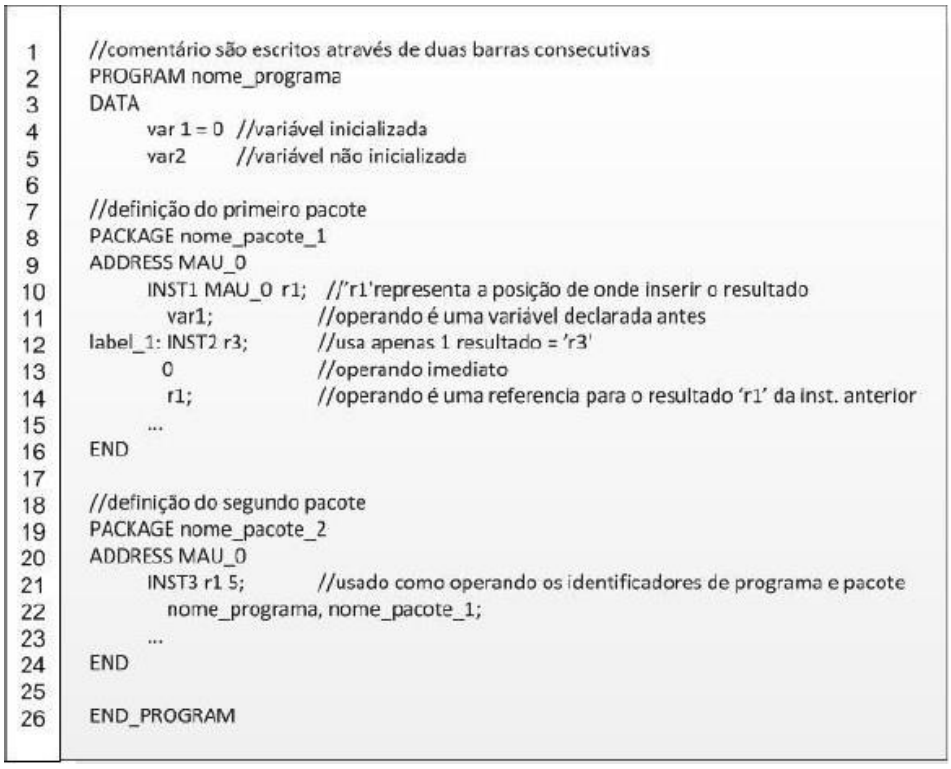
Já para o caso de utilizar um inteiro como posição de resultado (linha 21), deve-se especificar a posição de uma palavra que indique um operando de uma instrução posterior, contando inclusive com as palavras que serão injetadas durante a execução. No caso dos operandos, um identificador pode ser uma variável declarada na seção de dados (linha 11) ou uma referência para um identificador de resultado (linha 14). Se o identificador de pacote for utilizado como operando em uma instrução, o nome do programa precisa ser incluído seguido

por uma vírgula e o identificador do pacote (linha 22). Se um inteiro for usado como operando, será considerado seu valor imediato (linha 13).

Em instruções de desvio, identificadores de rótulo são utilizados, como pode ser visto na linha 12. A linha 10 ilustra uma instrução de controle, cuja primeira posição corresponde a identificação da MAU.

END_PROGRAM finaliza o programa, como pode ser visto na linha 26, depois que todos os pacotes são descritos. Os comentários são definidos por duas barras invertidas e podem ser usados em qualquer parte do programa.

Figura 12 - Programa PDL genérico

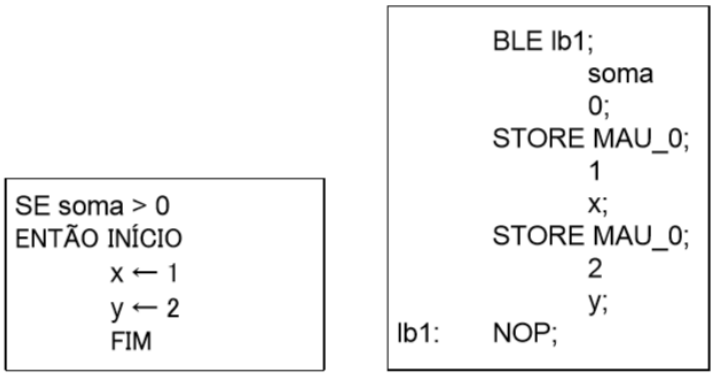


```
1   //comentário são escritos através de duas barras consecutivas
2   PROGRAM nome_programa
3   DATA
4         var 1 = 0  //variável inicializada
5         var2       //variável não inicializada
6
7   //definição do primeiro pacote
8   PACKAGE nome_pacote_1
9   ADDRESS MAU_0
10        INST1 MAU_0 r1;   //'r1'representa a posição de onde inserir o resultado
11          var1;           //operando é uma variável declarada antes
12  label_1: INST2 r3;      //usa apenas 1 resultado = 'r3'
13          0               //operando imediato
14          r1;             //operando é uma referencia para o resultado 'r1' da inst. anterior
15        ...
16  END
17
18  //definição do segundo pacote
19  PACKAGE nome_pacote_2
20  ADDRESS MAU_0
21        INST3 r1 5;       //usado como operando os identificadores de programa e pacote
22          nome_programa, nome_pacote_1;
23        ...
24  END
25
26  END_PROGRAM
```

Fonte: ARAUJO (2012)

Na Figura 13 (a) apresenta-se um exemplo de código contendo uma estrutura condicional simples em pseudocódigo. Nela, há apenas uma comparação (‘soma’ maior que 0), em caso de verdade, as atribuições para ‘x’ e ‘y’ são executadas, caso contrário o programa é encerrado. Em PDL, as estruturas condicionais são implementadas com instruções de desvios condicionais, como ilustrado em Figura 13 (b). Nesse código é realizada uma comparação inversa, ou seja, se soma for menor ou igual a 0 a execução salta para o *label* ‘lb1’, não executando as instruções *STORES*, que correspondem as atribuições das variáveis ‘x’ e ‘y’.

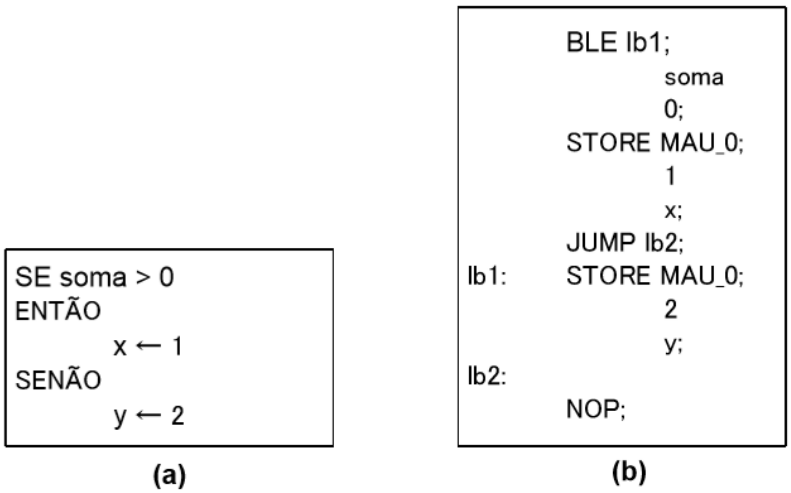
Figura 13 - Estrutura condicional simples: (a) em pseudocódigo; (b) em PDL



Fonte: FERNANDES; SILVA (2016)

Na Figura 14 é ilustrado um código contendo uma estrutura condicional composta em pseudocódigo (a) e em (b) seu correspondente em PDL. Nesse código, se soma for maior que 0 é atribuído o valor 1 a ‘x’, caso contrário, é atribuído o valor 2 a ‘y’. No código em PDL é feita a comparação inversa, de modo que, se soma for menor ou igual a 0 é feito um desvio para ‘lb1’ e a atribuição para ‘y’ é realizada. Caso contrário, a atribuição para ‘x’ ocorre e em seguida, um salto para ‘lb2’ é realizado, evitando assim o armazenamento para a variável ‘y’.

Figura 14 - Estrutura condicional composta: (a) em pseudocódigo; (b) em PDL



Fonte: FERNANDES; SILVA (2016)

Matrizes também são implementadas em PDL, todos os elementos são declarados e inicializados na seção DATA do pacote que contém o programa, como é ilustrado na Figura 15 para a matriz mat1[2][2]. A leitura de uma posição da matriz é feita por meio da seguinte fórmula: endereço_do_elemento [i][j] = j*m + i + endereço_base, em que ‘j’ corresponde ao valor da coluna corrente, ‘m’ ao valor total das colunas da matriz, ‘i’ ao valor da linha corrente

e ‘endereço_base’ ao endereço da primeira posição da matriz na seção DATA. Para o exemplo da Figura 16, o valor de ‘endereço_base’ para mat1 é 0. Caso uma outra matriz seja declarada logo em seguida, o valor de ‘endereço_base’ será 4.

Figura 15 – Declaração e inicialização da matriz mat1[2][2] em PDL

```
2 DATA
3         mat100 = 2
4         mat101 = 2
5         mat110 = 2
6         mat111 = 2
```

Fonte: AUTORIA PROPRIA (2016)

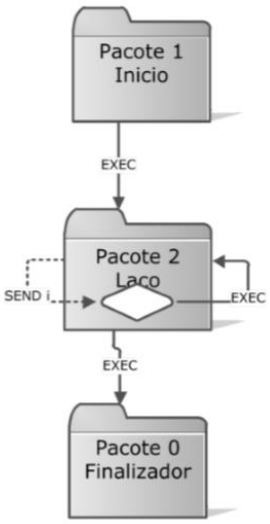
Vetores são implementados da mesma forma que matrizes, a diferença é que os valores de ‘m’ e ‘j’ são 0, transformando a fórmula em: endereço_do_elemento [i] = i + endereço_base.

Mais detalhes da implementação na linguagem PDL pode ser visto em ARAUJO (2012).

2.2.1.3 Paralelismo

Na Figura 16 é ilustrada a estrutura e a execução de pacotes de um programa que contém um laço de repetição.

Figura 16 - Estrutura e execução de um programa que contém um laço de repetição em PDL



Fonte: ARAUJO (2012)

Em PDL, cada laço de repetição é representado por dois pacotes, cujos nomes são escolhidos pelo programador. Na Figura 16, os pacotes são chamados de: “Inicio” e “Laço”. O pacote “Inicio”, corresponde ao conjunto de instruções que são executadas antes do laço de repetição. Por meio da instrução EXEC esse pacote chama o pacote “Laço” para ser executado. O pacote “Laço” representa o corpo do laço de repetição. Nesse pacote há uma instrução que

corresponde a uma condição. Enquanto essa condição for verdadeira, esse pacote é executado por meio da instrução *EXEC*, que chama ele mesmo. E por intermédio da instrução *SEND*, os valores para os operandos de destino são enviados a cada nova iteração do laço. Todos os operandos de destino são inicializados com um valor *default*, seguindo este padrão: operando destino "= valor *default*". Quando a condição não for mais satisfeita, a execução do laço é finalizada por meio da instrução EXEC que chama o pacote após o laço, na Figura 16 representado por "Finalizador". O pacote "Finalizador" contém a instrução EXIT que encerra o programa.

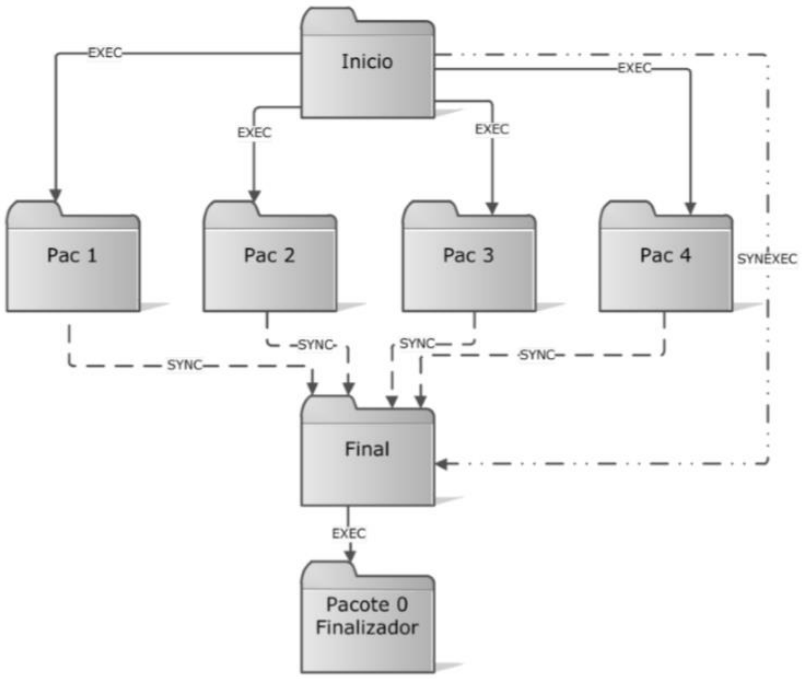
O paralelismo dessa arquitetura é representado na forma de um *pipeline* de pacotes injetados pela mesma MAU e pela injeção de pacotes simultâneos por meio de diferentes MAUs. No primeiro caso, um pacote é injetado logo após a injeção completa do pacote anterior. Já no segundo caso, conhecido na literatura como padrão de paralelismo *Fork-Join* (RAUBER; RÜNGER, 2010), o código é dividido em vários pacotes, os quais são injetados em MAUs diferentes criando fluxos de execução paralela. Em algum momento esses fluxos precisarão ser sincronizados para o encerramento do programa. Os códigos podem ser paralelizados em até quatro fluxos de execução, visto que a arquitetura disponibiliza de quatro MAUs para a injeção de pacotes.

Na Figura 17, ilustra-se o modelo do segundo caso de paralelismo em um código PDL. O pacote "Inicio" faz a injeção dos pacotes paralelos "Pac1", "Pac2", "Pac3" e "Pac4" (por meio da instrução EXEC) em uma MAU diferente. O pacote "Inicio" também indica o pacote que retornará o fluxo sequencial (pacote "Final") por intermédio da instrução SYNEXEC que lista os sinais de sincronismos que o pacote "Final" espera para ser injetado.

Dessa forma, ao final de cada pacote paralelo deve existir uma instrução SYNC destinada a MAU que injetará o pacote "Final". Os pacotes paralelos apresentam as mesmas instruções, com diferença nos endereços das variáveis e das MAUs, executam as operações em paralelo e enviam seus resultados parciais para o pacote "Final".

Quando todos os sinais de sincronismos chegarem até a MAU, o pacote "Final" é executado, ele agrupa todos os resultados parciais dos pacotes paralelos, gera o resultado final do programa e chama o pacote "Finalizador". Esse pacote encerra o programa. Da mesma forma que já apresentado para a Figura anterior, tanto a chamada quanto a descrição do pacote "Finalizador" podem ser omitidos com a substituição pela instrução EXIT.

Figura 17 - Modelo de paralelização de um código PDL



Fonte: ARAUJO (2012)

As aplicações em PDL são salvas em arquivo texto com a extensão ".asm" e submetida ao montador para análise. Caso não exista a presença de erros no arquivo, é gerado o código objeto equivalente com o mesmo nome do arquivo fonte, porém com a extensão ".ipn".

O arquivo objeto é submetido ao simulador para sua execução na arquitetura. Ao final é gerado um arquivo texto com os resultados obtidos da simulação, tais como: tempo de execução da aplicação, memória requerida, total de dados transmitidos, dentre outros.

2.3 Ferramentas de Compilação e Otimização

A implementação manual da etapa de otimização em um compilador é considerada uma tarefa árdua, necessitando de bastante empenho e tempo do desenvolvedor (COUTO, 2013). Contudo, dependendo do conjunto de otimizações escolhido, algumas vezes a utilização de alguma ferramenta automática não é possível. Isso acontece, pois apesar do grande número de otimizações disponíveis por essas ferramentas, o conjunto escolhido pode ser específico para a arquitetura alvo. Ao longo das últimas décadas, a construção de ferramentas automáticas para auxiliar na implementação de otimizações foram sendo desenvolvidas. Nesta seção são expostas duas ferramentas automáticas bastante conhecidas e utilizadas para a implementação de otimizações de código: o LLVM (LLVM, 2014), que corresponde a módulos utilizados no desenvolvimento de compiladores e otimização; e o GCC (GCC, 2015), que é o compilador dos

sistemas GNU e realiza tanto as análises de código quanto otimização de código. Ambos são *open source*, podem ser utilizados pela comunidade e com sua utilização há um ganho significativo em tempo de desenvolvimento.

2.3.1 LLVM

LLVM (*Low Level Virtual Machine*) é uma coleção de componentes modulares reutilizáveis para implementação de compiladores e de ferramentas para análise e otimização de código executável. Inicialmente foi um projeto de pesquisa da *University of Illinois* e hoje faz parte de um conjunto de subprojetos criados que utilizam esses componentes produzindo uma variedade de projetos comerciais e *open source*, também é bastante utilizado em pesquisas acadêmicas (LLVM, 2014) e na primeira distribuição Linux, o OpenMandriva Lx 3.0 (OPENMANDRIVA, 2016; SOFTPEDIA, 2016). Nessa distribuição seu compilador oficial é o Clang, que substituiu o GCC.

A licença de seu código é da *University of Illinois at Urbana-Champain* (UIUC) *Berkeley Software Distribution* (BSD) – *Style*. No ano de 2010, o projeto ganhou o prêmio da ACM *Special Interest Group on Programming Languages* (SIGPLAN) pelo reconhecimento do impacto que o LLVM obteve na comunidade de pesquisa em compiladores, por meio do grande número de publicações com referência ao projeto. Segundo Silva (2013), o LLVM possui como principais vantagens: o *design*, a linguagem independente e a extensibilidade.

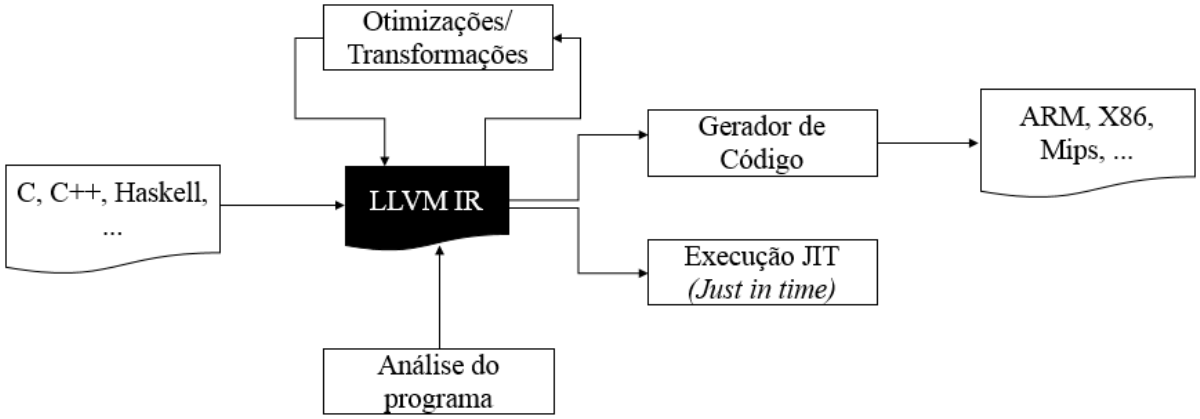
O LLVM apresenta *front-ends* para algumas linguagens de programação, tais como: C, C++, Haskell e Fortran. Ele compila programas implementados em uma dessas linguagens e os traduz para sua linguagem intermediária, chamada de LLVM-IR (*Low Level Virtual Machine Intermediate Representation*). Possui também *back-ends* para vários processadores, sendo capaz de gerar código para várias arquiteturas distintas (PARIZI, 2013).

A linguagem intermediária LLVM-IR é independente de linguagem de programação e de arquitetura alvo, ela reúne informações de alto nível dos programas, como por exemplo: tipos, para o suporte a análises e transformações e sua linguagem é bem próxima ao *assembly*, cujas instruções são no formato de três endereços. Sua representação é na forma SSA (*Single Static Assignement*), cuja atribuição de variável ocorre apenas uma vez. Esse formato facilita a implementação de otimizações. O LLVM possui um conjunto infinito de registradores, que são numerados sequencialmente no decorrer do programa (PARIZI, 2013).

A Figura 18 ilustra as etapas do processo de compilação de programas no LLVM. Um programa escrito em uma das linguagens suportadas pelo LLVM é transformado em LLVM-

IR. Em seguida, análises e otimizações são realizadas nesse código. A partir desse código intermediário pode-se gerar código objeto para uma arquitetura alvo por meio do gerador de código, ou ainda, pode ser executado por um tradutor JIT, em tempo de compilação, que converte as instruções do LLVM-IR em instruções de máquina, executando como se estivesse em uma máquina virtual, semelhante a linguagem de programação Java.

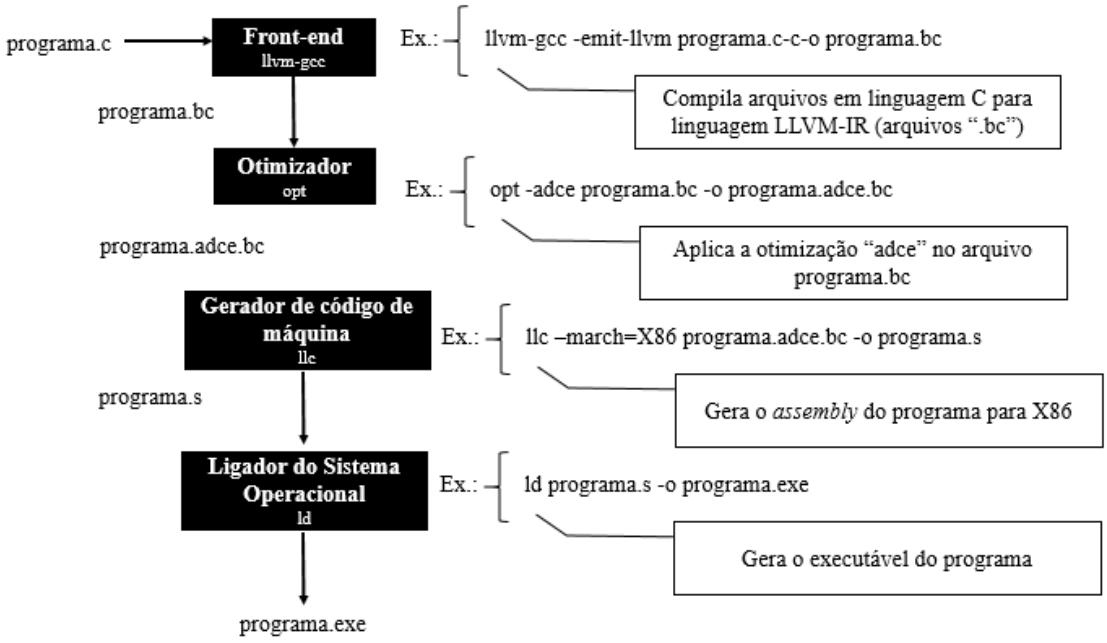
Figura 18 - Etapas do processo de compilação do LLVM



Fonte: Adaptação de PARIZI (2013)

A Figura 19 apresenta a sequência de comandos para a compilação de programas com o LLVM, iniciando com um programa escrito na linguagem de programação C – programa.c – até a geração do arquivo executável para a arquitetura alvo X86 – programa.exe. No exemplo dessa figura foi utilizada a otimização chamada de adce.

Figura 19 – Comandos e processo de compilação do LLVM



Fonte: Adaptação de PARIZI (2013)

Cada componente (módulo) do LLVM envolve os passos de análises do código ao longo do processo de desenvolvimento, criando ferramentas de código aberto que podem ser utilizadas e estendidas livremente (CID, 2010). Esses componentes não são utilizados somente no desenvolvimento de compiladores estáticos, são utilizados também em: interpretadores com geração de código em tempo de execução (compiladores *just in time* (JIT)), ferramentas de monitoramento de dados em tempo de execução (*profiling*), máquinas virtuais, ferramentas de análise estática, dentre outros (LATTER; ADVE, 2004). Quando módulos de baixo nível do LLVM são utilizados em projetos, esses são independentes da máquina alvo e suportam várias plataformas, tais como: x86, PowerPC, ARM, MIPS, dentre outras.

O LLVM é utilizado como uma plataforma robusta por projetistas de compiladores e pesquisadores na academia para o desenvolvimento de novas ideias e técnicas na área. Ele possui um grande número de contribuintes e usuários e uma vasta documentação. Esses fatores ajudam a diminuir o tempo no desenvolvimento de ferramentas. Com a infraestrutura LLVM, o desenvolvedor pode fazer uso apenas dos módulos necessários para a criação da ferramenta. Esses módulos estão bem documentados, assim o esforço é diminuído (CID, 2010).

O módulo responsável pela otimização e análise no LLVM é chamado por meio do comando *opt*. Ele apresenta como entrada um arquivo fonte LLVM, realiza as otimizações ou análises específicas e gera como saída os arquivos otimizados ou os resultados das análises (LLVM, 2014).

O comando *opt* tem os seguintes parâmetros: **opt** [*options*] [*filename*], nos quais *options* corresponde às opções utilizadas escolhidas pelo desenvolvedor, e *filename*, é o nome do arquivo de entrada, podendo ser no formato de linguagem *assembly* LLVM (.ll) ou no formato bitcode LLVM (.bc).

O LLVM apresenta três níveis padrões de otimização: O1, O2 e O3, os quais são formados por uma sequência de otimizações. Cada nível compreende um conjunto de otimizações acrescido das otimizações contidas no nível anterior (PEREIRA, 2011).

Mais detalhes e opções de otimizações podem ser encontradas em SPENCER; HENRIKSEN (2009).

2.3.1.1 CLANG

A ferramenta Clang é o *front-end* de C e C++ que utiliza a infraestrutura da geração de código do LLVM para a criação de um compilador completo. Essa ferramenta é bastante

documentada contribuindo com a redução do tempo necessário para a implementação das aplicações (Clang, 2012). Além do mais é *open-source*, ou seja, possui código aberto.

Clang é um compilador *self-hosted*, ou seja, ele é capaz de compilar o seu próprio código-fonte e fazer a geração de um executável funcional. Esse executável é capaz de compilar o mesmo código-fonte várias vezes e gerar executáveis funcionais a cada execução (DUARTE, 2014).

As principais metas desse compilador são:

- Compilação rápida e pouco uso de memória;
- Análise expressiva (por exemplo, mostrar os erros de uma forma mais fácil e clara para o usuário);
- Compatibilidade com o GCC;
- Código fonte simples e de fácil entendimento (KREMENEK, 2009).

A ferramenta automática Clang é capaz de gerar todo o *front-end* de um compilador. Realiza as análises léxica, sintática e semântica e faz a geração de código intermediário (LLVM-IR). O LLVM-IR corresponde a uma tradução linha a linha do código na linguagem de programação C ou C++. Para exemplificar seu funcionamento, alguns códigos foram inseridos e executados nessa ferramenta.

A Figura 20 (a) apresenta um trecho de código na linguagem C contendo declarações para três variáveis inteiras e operações de adição para essas variáveis dentro da função principal *main*. Como a tradução da Clang é linha a linha, caso o usuário insira duas instruções de carregamento para uma mesma variável e entre essas duas instruções seu valor não é utilizado, como ilustra o trecho de código da Figura 20 (a), o código gerado conterá essas duas instruções. Na Figura 20 (a), o valor para a variável "a" está sendo armazenado nas linhas 6 e 8. Porém, entre essas duas linhas não existe instrução que utilize o valor de "a" da linha 6. Tornando essa instrução (linha 6) totalmente dispensável e podendo ser eliminada. Contudo, como a tradução de Clang é fiel ao código do programa na linguagem de alto nível, seu código intermediário gerado apresentará a tradução para todas as instruções, como apresentado no código correspondente em LLVM-IR para esse trecho na Figura 20 (b). Caso a operação utilizada fosse uma subtração, multiplicação ou divisão, o código gerado seria semelhante.

Figura 20 – (a) Operação aritmética de soma em C (b) Código LLVM-IR do exemplo (a)

(a)

```c
int main() {
    int a, b, c;
    a = 1 + c;
    c = b + 3;
    a = b + 20;
}
```

(b)

```
; ModuleID = 'disser.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  %2 = load i32* %c, align 4
  %3 = add nsw i32 1, %2
  store i32 %3, i32* %a, align 4
  %4 = load i32* %b, align 4
  %5 = add nsw i32 %4, 3
  store i32 %5, i32* %c, align 4
  %6 = load i32* %b, align 4
  %7 = add nsw i32 %6, 20
  store i32 %7, i32* %a, align 4
  ret void
}

attributes #0 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = !{!"Ubuntu clang version 3.6.0-2ubuntu1 (tags/RELEASE_360/final) (based on LLVM 3.6.0)"}
```

Fonte: AUTORIA PRÓPRIA (2016)

Ainda na Figura 20 (b), pode-se observar entre as linhas 1 a 4 e 22 a 26 algumas informações de cabeçalho e rodapé que são geradas pelo Clang. Já entre as linhas 5 a 20, tem-se o código intermediário gerado. A linha 6 apresenta a definição da função *main* juntamente com seu abre chaves. Nas próximas três linhas tem-se as declarações das variáveis. O armazenamento na memória é realizado por meio da instrução *alloca*, que possui o parâmetro i32 indicando que será guardado um inteiro de 32 bits atribuído aos rótulos "%a" (linha 7), "%b" (linha 8) e "%c" (linha 9) que correspondem a endereços de memória.

A instrução *load* carrega o valor de uma variável da memória e nomeia com um rótulo que pode ser consumido dentro do programa. Seus parâmetros são o tipo seguido do nome da variável. A instrução *add* (linha 11) executa a adição entre variáveis da constante 1 e o valor

contido no rótulo “%2”, e seu resultado é armazenado em um novo rótulo, “%3”. Em seguida a instrução *store* é executada, armazenando o valor contido no rótulo “%3” na variável “%a”. Essa instrução possui como parâmetros o tipo da variável, seu rótulo, o tipo da variável destino e o rótulo destino, que representa o endereço de memória. As instruções de subtração, divisão e multiplicação, em LLVM-IR chamadas de *sub*, s*div* e *mul*, respectivamente, possuem os mesmos parâmetros da instrução de adição.

Para os próximos exemplos, as informações de cabeçalho e rodapé dos códigos intermediários foram omitidos.

A Figura 21 (a) exemplifica um desvio de fluxo por meio do comando *if*. Dentro da função *main*, tem-se a declaração e inicialização de duas variáveis inteiras e um desvio com *if*, cuja condição utiliza o operador booleano maior ou igual (>=). Se a condição for verdadeira, é atribuído o valor da variável “b” na variável “a”. Caso contrário, o programa é encerrado.

Figura 21 - (a) Desvio de fluxo (comando *if*) em C (b) Equivalente em LLVM-IR

```
4  void main() {
5      int a = 0, b = 1;
6      if (a >= b)
7      {
8          a = b;
9      }
10 }
```

(a)

```
5  ; Function Attrs: nounwind uwtable
6  define i32 @main() #0 {
7    %a = alloca i32, align 4
8    %b = alloca i32, align 4
9    store i32 0, i32* %a, align 4
10   store i32 1, i32* %b, align 4
11   %2 = load i32* %a, align 4
12   %3 = load i32* %b, align 4
13   %4 = icmp sge i32 %2, %3
14   br i1 %4, label %5, label %7
15
16 ; <label>:5                                      ; preds = %0
17   %6 = load i32* %b, align 4
18   store i32 %6, i32* %a, align 4
19   br label %7
20
21 ; <label>:7                                      ; preds = %5, %0
22   ret void
23 }
```

(b)

Fonte: AUTORIA PRÓPRIA (2016)

Na Figura 21 (b), apresenta o código intermediário correspondente a Figura 21 (a). O código gerado divide a instrução de desvio em duas instruções, dois *labels* são criados, o primeiro correspondendo ao bloco básico interno do comando e o segundo ao fluxo externo.

Para o desvio de fluxo, é utilizada a instrução de comparação *icmp* e seu resultado é atribuído a uma variável temporária. Essa instrução apresenta quatro parâmetros: o tipo de comparação (*sge*, do inglês *greater than equal*), o tipo das variáveis que serão comparadas (para inteiro é o i32) e as variáveis.

A instrução *br* (do inglês *branch*) é utilizada de duas maneiras: equivalente a um salto (linha 19) e contendo dois *labels*, cujo o desvio depende do resultado da condição da instrução anterior (linha 14). No primeiro caso, a instrução é formada pelo parâmetro *label*, que corresponde ao destino do desvio. Já o segundo caso, são utilizados como parâmetros: tipo do booleano, o rótulo da variável que armazena o resultado da comparação e dois *labels*: um para o caso da condição ser verdadeira e o outro caso seja falso, nessa ordem.

Na linha 19 há um desvio para o rótulo "7", que corresponde a instrução seguinte (linha 21). Durante a geração de código intermediário para laços de repetição, a Clang insere no código instruções de salto para a instrução imediatamente seguinte, sendo desnecessário.

As instruções com os demais operadores booleanos ocorre da mesma maneira, apenas é necessária a troca do parâmetro *sge* por *sle*, *slt*, *sgt* e *eq*, que correspondem a menor ou igual, menor que, maior que e igual, respectivamente.

A Figura 22 (a) ilustra um exemplo de código em C contendo o laço de repetição *for*. O laço *for* justamente com a declaração da variável "soma" estão dentro da função principal *main*. Esse laço possui dez iterações e dois incrementos para a variável soma. Os laços de repetição *for*, *while* e *do while*, assim como para o comando *if*, utilizam as instruções *icmp*, *br* e *labels* para fazer o desvio. Porém, com uma organização diferente da utilizada em *if*. Como pode-se observar na Figura 22 (b), o código intermediário gerado é visivelmente maior que os outros códigos apresentados anteriormente, pois operações desnecessárias são geradas.

Ainda observando a Figura 22 (b), pode-se ver que quatro *labels* foram gerados (dois são dispensáveis). Nas linhas 13 a 15 e 27 a 29 apresentam *labels* e instruções *br* que podem ser eliminadas, visto que o desvio corresponde a instrução imediatamente seguinte.

O código intermediário referente ao laço *for* inicia na linha 17 com o instrução de comparação e finaliza na linha 35, que contém o *label* para quando a comparação for falsa. O padrão das operações contidas dentro do corpo do *for* é o mesmo já mostrado anteriormente quando foram expostas as instruções aritméticas.

O código LLVM-IR para laço de repetição com *while* e *do while* é bastante semelhante ao código para o *for*.

Figura 22- (a) Trecho de código contendo o laço de repetição *for* (b) Equivalente LLVM-IR

```
4  int main() {
5      int soma = 0;
6
7      for (int u = 0; u < 10; u++)
8      {
9          soma += 1;
10         soma += 2;
11     }
12     return 0;
13 }
```

(a)

```
5  ; Function Attrs: nounwind uwtable
6  define i32 @main() #0 {
7    %1 = alloca i32, align 4
8    %soma = alloca i32, align 4
9    %u = alloca i32, align 4
10   store i32 0, i32* %1
11   store i32 0, i32* %soma, align 4
12   store i32 0, i32* %u, align 4
13   br label %2
14
15 ; <label>:2                                      ; preds = %10, %0
16   %3 = load i32* %u, align 4
17   %4 = icmp slt i32 %3, 10
18   br i1 %4, label %5, label %13
19
20 ; <label>:5                                      ; preds = %2
21   %6 = load i32* %soma, align 4
22   %7 = add nsw i32 %6, 1
23   store i32 %7, i32* %soma, align 4
24   %8 = load i32* %soma, align 4
25   %9 = add nsw i32 %8, 2
26   store i32 %9, i32* %soma, align 4
27   br label %10
28
29 ; <label>:10                                     ; preds = %5
30   %11 = load i32* %u, align 4
31   %12 = add nsw i32 %11, 1
32   store i32 %12, i32* %u, align 4
33   br label %2
34
35 ; <label>:13                                     ; preds = %2
36   ret i32 0
37 }
```

(b)

Fonte: AUTORIA PRÓPRIA (2016)

Como apresentado anteriormente, a Clang gera algumas instruções desnecessárias que precisam ser consideradas durante a criação de *softwares* que utilizará essa ferramenta, sendo necessário o desenvolvimento de uma etapa para a eliminação dessas instruções.

2.3.2 GCC

O *GNU Compiler Collection*, usualmente chamado de GCC foi escrito por Richard Stallman em 1987 como um compilador para sistemas operacionais GNU, criados para serem *softwares* totalmente livres. Em meados de 1997, um grupo de desenvolvedores formou o projeto EGCS que reuniu várias versões experimentais do GCC em apenas uma, e em abril de 1999, tornou-se a versão oficial do GCC (GCC, 2015).

O projeto GCC é mantido por vários desenvolvedores de todo o mundo, possui um ambiente aberto e é multiplataforma. Esse compilador frequentemente é utilizado quando se deseja desenvolver *software* e executá-lo em vários tipos de *hardwares* (GOUGH, 2004).

Inicialmente, o GCC suportava apenas a linguagem de programação C e era conhecido como *GNU C Compiler*, mas com o passar dos anos ele ganhou o suporte à outras linguagens, tais como: C++, Fortran, Ada, Java e Objective-C. Os processadores: Alpha, Amd, ARM, MIPS, PowerPC, SPARC e x86, pertencem ao conjunto suportado pelo GCC (GCC, 2015).

Sua interface externa é padrão para os compiladores disponíveis no sistema operacional Linux. O *driver* gcc é invocado pelos utilizadores que interpreta os argumentos do comando e toma a decisão de qual compilador utiliza para cada arquivo de entrada, em seguida o *assembler* é chamado e por fim, o *linker* é executado, sendo responsável por ligar os arquivos binários e produzir o arquivo executável (STALLMAN, 2003).

Assim como em outros compiladores, o GCC recebe como entrada um código-fonte e produz um código de linguagem *assembly*. Seu *front-end* analisa gramaticalmente as linguagens e gera uma árvore de sintaxe abstrata (*Abstract Syntax Tree* - AST) e seu *back-end* converte as ASTs em linguagem de transferência de registro (*Register Transfer Language* - RTL) do GCC. Várias otimizações são executadas e é gerado o código de máquina utilizando o *pattern matching* específico de cada arquitetura (STALLMAN, 2001).

O GCC é um compilador otimizante, cujo objetivo é aumentar a velocidade ou reduzir o tamanho dos arquivos executáveis que são gerados. Para a geração do arquivo executável mais rápido são considerados alguns fatores, tais como: várias combinações de instruções são geradas para cada comando do código fonte, cujas combinações são consideradas pelo compilador e a escolha da melhor é realizada; e a geração de código apropriado para cada processador e linguagem (GOUGH, 2004).

A otimização em nível de código-fonte é utilizada pelo GCC, a qual não precisa de nenhum conhecimento das instruções de máquina. Duas técnicas comuns são: eliminação de subexpressão comum e *inlining* de funções (apresentadas nas seções 2.1.1 e 2.1.9,

respectivamente, deste capítulo). Essas otimizações possuem o objetivo de aumentar a velocidade e reduzir o tamanho do programa.

O GCC também possui otimizações cujo objetivo é a geração de um código mais rápido, mas com o aumento do tamanho do arquivo executável. Ou ainda, a diminuição do tamanho do executável, com o aumento da lentidão do código, conhecido como *trade off* velocidade-espaço. Um exemplo daquele tipo de otimização é o desenrolamento de laço (*loop unrolling*), nela há um aumento na velocidade de *loops*, eliminando a verificação da condição de fim do *loop* em cada iteração. O código gerado são atribuições independentes, ideal para os processadores que suportam o paralelismo. O desenrolamento de laço aumenta a velocidade e o tamanho do arquivo executável, com exceção dos laços com apenas uma ou duas iterações.

Visando controlar o tempo de compilação, o uso da memória e o *trade-off* velocidade-espaço para o arquivo executável, o GCC possui níveis de otimização que são numerados de 0 a 3, e também opções individuais para otimizações específicas. A otimização é escolhida na linha de comando através da opção '-Onível', onde o nível corresponde a um valor entre 0 e 3. São eles (GOUGH, 2004):

- **'-O0'**

O GCC não executa nenhuma otimização e realiza a compilação mais simples do código fonte. A conversão do código fonte é direta para o arquivo executável, sem transformações.

- **'-O1' ou '-O'**

Esse nível ativa as otimizações que não necessita de *trade-off* velocidade-espaço. Nele, os arquivos executáveis gerados são menores e mais rápidos que os gerados através da opção '-O0'. Otimizações com maiores complexidades não são utilizadas nesse nível.

- **'-O2'**

Essa opção ativa as otimizações do nível 1 e outras otimizações adicionais, dentre elas, há o agendamento. Nesse nível são utilizadas as otimizações que não requerem *trade-off* velocidade-espaço, de modo que o arquivo executável não deve aumentar de tamanho. O tempo de compilação dos programas e a quantidade de memória são maiores que os exigidos no nível 1. Nele, tem-se uma otimização máxima sem que o tamanho do arquivo executável aumente, sendo a melhor escolha para o desenvolvimento de um programa. É o nível de otimização padrão dos *releases* de pacotes GNU.

- **'-O3'**

Essa opção ativa as otimizações dos níveis 1 e 2, além de otimizações mais custosas, como *inlining* de funções. Nesse nível, a velocidade e o tamanho do executável podem

aumentar. Em alguns casos, as otimizações não são favoráveis, essa opção pode deixar um programa mais lento.

- **‘-funroll-loops’**

Essa opção ativa o desenrolamento de laço, aumentando o tamanho do arquivo executável. É independente das outras otimizações.

- **‘-Os’**

As otimizações que reduzem o tamanho do arquivo executável são ativadas. É ideal para sistemas limitados no tamanho da memória ou no espaço em disco, pois o objetivo é a geração de um arquivo executável menor possível. Em alguns casos, um arquivo executável menor executará mais rápido, devido a uma melhor utilização de sua cache.

A utilização de uma otimização traz benefícios, porém é necessário que esse seja pesado em relação ao custo. O custo inclui uma maior complexidade na depuração e aumento no tempo e na quantidade de memória utilizada durante a compilação. Recomenda-se utiliza ‘-O0’ para a depuração e ‘-O2’ para o desenvolvimento.

A opção ‘-g’ de depuração pode ser utilizada em combinação com o uso de otimização no GCC. Essa combinação não é utilizada em outros compiladores. Quando se utiliza essas duas opções juntas, os rearranjos realizados pelo otimizador podem dificultar a visualização do que está acontecendo ao examinar um programa otimizado no depurador. Por exemplo, variáveis temporárias frequentemente são eliminadas e a ordenação de instruções pode ser modificada. A opção de depuração ‘-g’ é ativada juntamente com a opção de otimização ‘-O2’ por *default* em *releases* de pacotes GNU.

No processo de otimização, o compilador faz a análise do fluxo de dados, ou seja, ele verifica a utilização das variáveis e seus valores iniciais. Essa análise é a base de algumas otimizações. Durante essa análise pode haver a detecção de variáveis não inicializadas.

Na compilação de um código com otimização alguns avisos podem aparecer, que não aparecem se o código não apresentar nenhuma otimização. A opção ‘-*Wuninitialized*’ adverte quando algumas variáveis são lidas sem serem inicializadas.

Ao compilar um código com a opção ‘-*Wall*’ e alguma otimização, é produzido algum aviso. Se a otimização não for utilizada, nenhum aviso é produzido, visto que a análise de fluxo de dados não é realizada sem otimização. Para a produção de bons avisos, na prática, é preciso utilizar o nível de otimização ‘-O2’.

2.4 Trabalhos Relacionados

A geração de código *assembly* para qualquer arquitetura alvo corresponde a uma tarefa árdua, que precisa do conhecimento do processador e de tempo do desenvolvedor. Dessa forma, há a necessidade da criação de uma ferramenta de geração de código para determinada arquitetura. Assim, trabalhos que consistem em compiladores para um processador são expostos neste Capítulo. Inicialmente é apresentado o compilador Cetus. Em seguida, um gerador de aplicação e o compilador fila QRP-GCC. Por fim, alguns trabalhos que consistem em uma ferramenta de tradução de código em uma linguagem de programação de alto nível para o código *assembly* da arquitetura IPNoSys (PDL).

O compilador Cetus proposto por LEE; JOHNSON; EIGENMANN (2003) foi desenvolvido na linguagem de programação Java para a tradução de código serial C em OpenMP. Ele é formado por um conjunto de classes que realizam a geração de código intermediário do programa inserido como entrada. Sua tradução para OpenMP é realizada através de análises e transformações visando a paralelização e otimização automática do código fonte. Dessa forma, código sequencial inserido como entrada que possua laços de repetição, após as análises e transformações do Cetus é transformado em um código cujo laço é executado de forma paralela utilizando o OpenMP; gerando códigos paralelos que apresentam um desempenho relativamente melhor que seus correspondentes sequencial.

Em GONÇALVES *et al*. (2014) foi desenvolvido um gerador de aplicação que identifica laços no código-fonte escrito em forma linear, caso seja possível, esses laços são reescritos para a geração de código com o mesmo comportamento, porém com maior nível de paralelismo, que será executado no processador gráfico compatível com CUDA. O gerador lê o código C inserido como entrada, realiza uma análise nas dependências das instruções que estão presentes nos laços, se possível, elas são tratadas, e é realizada a divisão do corpo do laço. Em seguida, é gerado o código intermediário contendo informações acerca dos laços que devem ser paralelizados. E a partir desse código é gerado o código paralelo. Em seus testes, a aplicação de multiplicação de matrizes com dimensão de 25 milhões de elementos foi utilizada para avaliar o tempo de execução de sua versão sequencial e paralela. Os resultados mostraram que a versão paralela apresenta um melhor desempenho comparada a sua versão sequencial. Com isso, o autor justifica que atualmente ainda existem muitos programas importantes lineares em funcionamento. E que esses programas estão sobrecarregados em função da incapacidade do processamento de muitas tarefas ao mesmo tempo. A solução para isso é a paralelização automática de *software*, permitindo assim que programas antigos possam usufruir da tecnologia recente.

O compilador fila, chamado de QRP-GCC, foi desenvolvido em CANEDO; ABDERAZEK; SOWA (2006) para um processador baseado em fila paralelo ou *Queue Machines* (SCHMIT; LEVINE; YLVISAKER, 2002). Esse tipo de processador foi exposto na seção 2.2.1 deste capítulo e apresenta um modelo de computação semelhante a Arquitetura IPNoSys. O processador realiza a execução de programas paralelos, explorando o paralelismo em nível de instrução (ILP). Esse compilador utiliza a infraestrutura GCC para a geração de programas fila. Para isso, uma estrutura de dados FIFO (*first-in-first-out*) é utilizada e garante a exatidão das expressões dos programas. Uma vez que os operandos armazenados são removidos do topo da fila e o resultado das operações são armazenados no fim da fila.

O Modelo Computacional Fila consiste no mecanismo que transforma uma expressão em um programa fila e corresponde à ideia por trás do Processador Fila Paralela (PQP) e Processador Fila Registrador (QRP). Essa transformação consiste em algumas etapas. Inicialmente, o GCC gera as ASTs (Árvores Sintáticas Abstratas) para cada expressão do código fonte. Em seguida, essas árvores são transformadas em um nível mais baixo, representação independente de máquina chamada de *Register Transfer Language* (RTL). Na próxima etapa, são aplicadas a maioria das otimizações do GCC 3.3.3 e gerada uma RTL otimizada dependente de máquina, com as limitações da arquitetura alvo. Na etapa seguinte, as instruções são reordenadas, a cada leitura de um bloco básico as instruções são analisadas segundo a dependência de dados e reordenadas visando o paralelismo. O paralelismo em nível de instrução é conseguido por meio da junção de árvores no mesmo nível de profundidade, por intermédio do algoritmo de *Boytchev*. Esse algoritmo faz uma busca por operandos sem dependência de dados de cima para baixo com uso das declarações do bloco básico. E por fim, na última etapa o código *assembly* é gerado.

Um conjunto de programas foram utilizados para realizar a comparação entre os compiladores QRP-GCC e PQP-GCC. Notou-se que o número de instruções geradas pelo QRP-GCC foi inferior as instruções geradas pelo PQP-GCC. Isso é explicado pelas diferenças na arquitetura desses processadores, o processador QRP é composto por um conjunto de instruções flexíveis e por registradores filas e de acesso aleatório, ao contrário do processador PQP que dá menos liberdade na especificação dos operandos, fazendo a seleção de instrução e agendamento inadequado. As arquiteturas QRP e Sparc 64 utilizando o compilador GCC também foram comparadas, em relação aos seguintes pontos: número de instruções *assembly* geradas e tamanho do código em bytes. Em média, o QRP gerou programas maiores, devido ao *hardware* QRP, que necessita de operações especiais para instruções de desvio e procedimento de

chamada. Já em relação ao tamanho do código, para o QRP foi menor, uma vez que seu conjunto de instruções é de 16 bit.

A pesquisa dos autores em relação a processadores filas paralelas mostra que um compilador otimizador capaz de extrair ILP para o modelo computacional de fila é fundamental para manter a semântica do programa e atingir um alto desempenho. O processador QRP é adequado para programas embarcados.

O primeiro trabalho que consiste em uma ferramenta de tradução de código para IPNoSys foi proposto por PINTO; BANDEIRA; FERNANDES (2010) a partir de uma gramática miniC (subconjunto da linguagem C) para a geração de código PDL. Seu desenvolvimento foi na linguagem de programação Java sem a utilização de ferramentas automáticas, fazendo uso de técnicas de construção de compiladores tradicionais pertencentes a literatura. Seu *back-end* foi realizado em duas etapas: geração de código de três endereços e geração de código em PDL. A geração de código PDL foi realizada a partir do reconhecimento do tipo do código de três endereços (comandos sequenciais, tomadas de decisão ou estruturas de repetição) e traduzido para os respectivos conjuntos de macros PDL. Esse compilador realiza a geração de código PDL de estruturas sequenciais e tomadas de decisão. Apresentando como limitações: gramática reduzida, ausência da tradução de laços de repetição e segue o modelo de geração de código para processadores tradicionais, gerando muitas instruções de acesso a memória (*loads* e *stores*), como visto na seção 2.2.1, essas instruções apresentam um custo extra à arquitetura.

Visando a exploração do paralelismo da arquitetura IPNoSys, GADELHA; CORREA; KREUTZ (2011) criou uma ferramenta para a realização da tradução de código na linguagem de alto nível, ANSI C, em PDL. Essa ferramenta explora o paralelismo presente no processador or meio de sua interface de programação e várias técnicas para a resolução de problemas de concorrência. Em GADELHA; CORREA; KREUTZ (2011) foi utilizada a ferramenta Clang para o desenvolvimento do *front-end* do compilador e a ferramenta proposta corresponde ao *back-end* que transforma o código intermediário gerado pela Clang em código de máquina do IPNoSys. O paralelismo explorado nesse trabalho é por meio da geração de um conjunto de pacotes. Para isso, duas tarefas foram realizadas: identificação das dependências de dados no código e geração dos pacotes baseada nos seguintes requisitos: o programa deve possuir mais de um pacote e o pacote deve possuir mais de uma instrução regular. Sua geração de pacotes apresentou uma melhora no desempenho comparado com o compilador de PINTO; BANDEIRA; FERNANDES (2010). Uma vez que no compilador de PINTO; BANDEIRA; FERNANDES (2010) não existia a exploração de paralelismo e todas as instruções estavam em

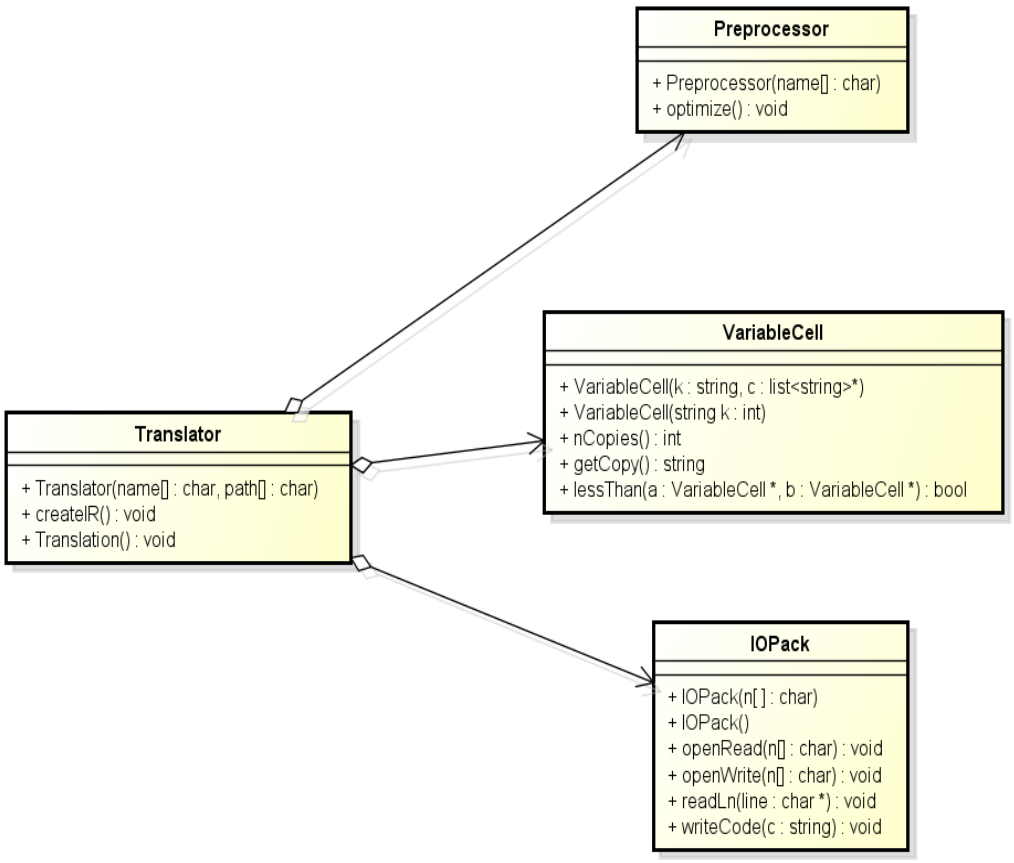
um mesmo pacote. No trabalho de GADELHA; CORREA; KREUTZ (2011), algumas deficiências são apresentadas, tais como: ausência de otimização em relação as instruções de *load* e *store* e a exploração do paralelismo em nível de instruções, visto que essa arquitetura explora melhor paralelismo em nível de tarefa.

O terceiro trabalho desenvolvido por GOMES; COUTO; ARAUJO (2014), cujo autor desta dissertação teve participação, será mostrado com mais detalhes, visto que foi utilizado para o desenvolvimento deste trabalho de dissertação. Chamaremos esse trabalho de COMPILADOR1. GOMES; COUTO; ARAUJO (2014) também fez uso da ferramenta Clang para o desenvolvimento do *front-end* do compilador, que gera automaticamente desde a primeira etapa do compilador, a análise léxica, até a geração de código intermediário (LLVM-IR). A partir do código gerado é realizada uma tradução para a linguagem *assembly* da arquitetura alvo, a PDL. O tradutor corresponde ao *back-end* do compilador juntamente com duas otimizações: eliminação de desvios desnecessários e redução de instruções *loads* e *stores* (serão detalhadas posteriormente). O objetivo dessa ferramenta é a geração de código considerando que as instruções *load* e *store* correspondem a instruções caras para o processador, como mostrado na seção 2.2.1.

Sua utilização consiste do usuário inserir no código do compilador o nome do arquivo em C que deseja realizar a tradução. Seu *back-end* corresponde a esse trabalho proposto e foi implementado em quatro classes: *Translator*, *Preprocessor*, *VariableCell* e *IOPack*.

A Figura 23 ilustra o diagrama de classes do *back-end*, mostrando quais atributos e métodos cada classe é composta. A classe *Translator* corresponde a classe principal e é formada por dois métodos e um construtor. A classe *Preprocessor* realiza as otimizações e é composto de um método. A classe *VariableCell* corresponde a uma célula da árvore que é formada pela chave e por uma lista de cópias das variáveis. E por fim, a classe *IOPack* realiza as operações de entrada e saída em arquivos com códigos: em C, intermediários e em PDL. Todas as classes serão detalhadas mais adiante.

Figura 23 - Diagrama de classes do *back-end* do compilador



Fonte: AUTORIA PRÓPRIA (2015)

Na classe *Translator* (definição na Figura 24), a Clang é chamada através do método *createIR* e realiza a tradução do arquivo de entrada em código intermediário LLVM-IR. Esse código é utilizado como entrada para a classe *Preprocessor*, que aplicará otimizações gerando um código transformado. O código após as modificações é enviado para *Translator* para a tradução linha a linha em linguagem PDL, através do método *translation.*

Translator é a única classe que precisa ser instanciada na função principal *main*. Ela possui as seguintes variáveis privadas: *fontName*, que corresponde ao nome do código fonte em C; *pathName*, corresponde ao diretório onde o código fonte está salvo; *mainCode* armazena o código em PDL, sem as declarações das variáveis; e *varCode* guarda as declarações das variáveis. O objeto *io* corresponde a uma instância da classe *IOPack*, que é utilizada para entrada e saída em arquivos. Visando a diminuição da utilização das instruções *load*s, rótulos para as variáveis são criados para serem utilizados posteriormente e armazenados juntamente com essas variáveis na árvore *copies*. Por fim, a variável *cmp* é um ponteiro para a função que irá comparar duas estruturas *VariableCell*.

Figura 24 – Cabeçalho da classe *Translator*

```
class Translator
{
 private:
    char fontName[50];
    char pathName[300];
    stringstream mainCode;
    stringstream varCode;
    int nTabs;
    IOPack * io;
    set < VariableCell*, bool(*)(VariableCell*, VariableCell *)> copies;
    bool(*cmp) (VariableCell *, VariableCell *);

 public:
    Translator(char name[], char path[]);
    ~Translator();

    void createIR();
    void translation();
};
```

Fonte: AUTORIA PRÓPRIA (2015)

Na classe *Preprocessor*, são realizadas varreduras no código buscando possíveis locais de otimização, a definição dessa classe é mostrada na Figura 25. Ela é formada por: uma instância de *IOPack* chamada de *io*, *nameProgram* que armazena o nome do programa em C e *code* que guarda o código após a execução da função *optimize*, e é enviado para a classe *Translator*.

Figura 25 – Cabeçalho da classe *Preprocessador*

```
class Preprocessor
{
 private:
    IOPack * io;
    char nameProgram[100];
    stringstream code;

 public:

    Preprocessor(char name[]);
    ~Preprocessor();

    void optimize();
};
```

Fonte: AUTORIA PRÓPRIA (2015)

Nessa classe, o código é preparado para a geração dos pacotes dos laços de repetição. É realizada também a otimização de eliminação de desvios desnecessários, como visto na subseção 2.3.1.1 deste Capítulo, ao gerar código para desvios de fluxo de execução, a Clang gera desvios para instruções imediatamente seguintes. Para remover esses desvios desnecessários, conhecido na literatura como otimização *peephole*, no COMPILADOR1 foi

desenvolvido um pré-processamento para o código intermediário gerado pelo Clang, que consiste em duas etapas: eliminação de *jumps* e eliminação de *labels* sem referência.

Na primeira etapa são removidos do código intermediário as instruções de *jump* que apontam para instruções seguintes (sendo omitidos apenas os *jumps*, os *labels* para onde estão apontando permanecem).

Já na segunda etapa, são removidos os *labels* sem instrução que aponte para eles (ocasionado pela etapa anterior). Uma varredura no código é realizada e todos os *labels* encontrados são anotados em uma árvore de busca binária, cuja chave do nó corresponde ao próprio *label*. Quando um *branch* é encontrado, é feita uma busca na árvore a procura dos *labels* para os quais a instrução aponta e é anotado que o *label* possui uma referência no código. Todos os *labels* que possuem referência são retirados da árvore, cujos *labels* restantes correspondem aos que precisam ser eliminados do código. Durante outra varredura no código, ao encontrar um *label* é realizada uma busca na árvore, em caso verdadeiro será removido do código, caso contrário, permanecerá. Ao final da varredura os *labels* são removidos.

Como a ferramenta COMPILADOR1 tem como alvo a arquitetura IPNoSys, uma particularidade dela também motivou uma outra otimização: a redução de instruções *load* e *store.* Instruções desse tipo apresentam um custo extra para a arquitetura, como apresentado na seção 2.2.1.

Quando tem-se uma instrução *store* realizando o armazenamento em uma determinada variável e em seguida tem-se uma instrução de *load* dessa variável, as instruções de armazenamento e carregamento são desnecessárias, uma vez que o valor dessa variável será utilizada nas próximas instruções. Para evitar esse problema, a solução é a criação e armazenamento de duas cópias da variável em uma árvore binária de busca, cuja chave é o nome da variável. Para a eliminação das instruções de *load*, é realizada uma busca na árvore objetivando encontrar uma cópia da variável presente na instrução. Caso o resultado da busca seja verdadeiro, uma das cópias é removida e utilizada, caso contrário, a instrução *load* é inevitável. Quando a instrução *ret* (retorno) é encontrada, é necessário o *store* de uma cópia de cada chave que pertence a árvore, já que ela apresenta o resultado de todas as computações realizadas.

Dessa forma, a utilização de instruções de armazenamento, para todas as variáveis, somente é gerada ao final da tradução do código, evitando assim a repetição do carregamento de uma determinada variável. Já para as instruções de carregamento, é realizada a troca dessa instrução pela instrução de cópia (*copy*). O caso em que isso não ocorre é quando não existe

mais cópias armazenadas dessa variável e a instrução *load* é utilizada. Gerando assim, código contendo instruções de *copys* desnecessárias.

A classe *VariableCell* é responsável por inserir um *label* a sua lista de cópias. Sua definição está mostrada na Figura 26. Como pode-se observar, a classe possui duas variáveis: *key*, que armazena o nome do *label*; e *copies*, que corresponde a lista de cópias disponíveis. Seu método *lessThan* realiza a comparação entre suas chaves para que uma instância da classe possa ser inserida e buscada na árvore. O método *nCopies* retorna o número de cópias disponíveis para o *label*. E o método *getCopy* remove uma cópia da lista e retorna-a para ser utilizada.

Figura 26 – Cabeçalho da classe *VariableCell*

```
class VariableCell
{
public:
    string key;
    list<string> * copies;

public:
    VariableCell(string k, list<string> * c);
    VariableCell(string k);
    ~VariableCell();

    static bool lessThan (VariableCell * a, VariableCell * b );
    int nCopies();
    string getCopy();
};
```

Fonte: AUTORIA PRÓPRIA (2015)

Por fim, a classe *IOPack* faz a manipulação de arquivos. Sua definição é apresentada na Figura 27. Nessa classe, seus atributos geram instâncias para os objetos de manipulação de arquivo - *fin* e *fout*. A variável *name* corresponde ao nome do arquivo que está sendo manipulado. A variável *code* armazena o código lido e as *flags* de controle (*flagFilesRead* e *flagFilesWrite*) são usadas para fechar os arquivos abertos. Essa classe apresenta os métodos *openRead* e *openWrite*, que controlam a abertura do arquivo para a leitura e escrita, respectivamente. O método *readLn* faz a leitura de uma linha do arquivo e guarda seu conteúdo na variável de entrada *line*. Por fim, o método *writeCode* salva em arquivo a *string* de entrada que contém o código em PDL.

Figura 27 – Cabeçalho da classe *IOPack*

```
class IOPack
{
 private:
    ifstream * fin;
    ofstream * fout;
    char name[50];
    string code;
    bool flagFilesRead;
    bool flagFilesWrite;

    void closeReadFile();
    void closeWriteFile();
 public:
    int LINE;

 public:
    IOPack(char n[]);
    IOPack();
    ~IOPack();

    void openRead(char n[]);
    void openWrite(char n[]);
    void readLn(char * line);
    void writeCode(string c);
};
```

Fonte: AUTORIA PRÓPRIA (2015)

O código gerado pelo compilador pode ser usado pelo montador de IPNoSys para a geração de código objeto, que é usado pelo simulador dessa arquitetura para execução.

A ferramenta COMPILADOR1 apesar de possuir melhor desempenho em relação as outras propostas anteriores, visto que considera a característica da arquitetura alvo em relação as instruções de *load* e *store*, possui como deficiências: a geração de instruções *copys* desnecessárias, visando a redução de *loads*, que corresponde a erro para a IPNoSys, visto que sempre que uma variável temporária é criada é necessária sua utilização; e ao reduzir *stores*, há a geração de instruções de *stores* desnecessárias para todas as variáveis, visto que o compilador não considera se determinada variável não foi modificada.

2.5 Considerações

Visando a utilização dos novos recursos das arquiteturas paralelas atuais surge a necessidade da geração de um código melhorado que considere as características dessas arquiteturas. Para isso, é necessária a aplicação de otimizações a esses códigos. Na literatura,

há uma variedade de técnicas de otimização disponíveis e algumas foram descritas neste capítulo. A escolha de qual otimização utilizar depende da finalidade do compilador e da arquitetura alvo.

Este trabalho consiste em uma segunda versão da ferramenta COMPILADOR1, a qual é desenvolvida uma etapa de otimização nela. COMPILADOR1 faz uso da ferramenta automática Clang para a geração de seu *front-end* e traduz um código em C para a linguagem *assembly* da Arquitetura IPNoSys, a PDL. A ferramenta Clang foi escolhida por gerar código intermediário semelhante ao código PDL. Esse código guarda os resultados intermediários das instruções em variáveis temporárias, o mesmo ocorre com a PDL. O conjunto de otimizações escolhido para esse trabalho é específico da IPNoSys e foi implementado manualmente. Entre as otimizações escolhidas, uma explora o paralelismo da IPNoSys e para ela foi utilizada a ideia da técnica de desenrolamento de laço, descrita na seção 2.1.7.4.

3 COMPILADOR OTIMIZANTE C2PDL

O compilador otimizante C2PDL (lê-se C *to* PDL) proposto como objeto de estudo e testes desta Dissertação é apresentado neste capítulo. Ele foi chamado assim porque possibilita ao usuário a tradução otimizada de seus programas escritos na linguagem C para a linguagem *assembly* da arquitetura IPNoSys, a PDL. Este trabalho consiste no desenvolvimento da etapa de otimização da ferramenta de compilação COMPILADOR1, apresentada no Capítulo 2. A característica da arquitetura IPNoSys em relação as instruções de *load* e *store* e as deficiências da ferramenta COMPILADOR1, expostos na seção 2.4, são considerados neste trabalho.

Beneficiando-se da geração de código para a arquitetura IPNoSys, a ferramenta COMPILADOR1 foi escolhido. E em seu código, foi implementada uma etapa de otimização oferecendo três níveis de otimização. O desenvolvimento desta ferramenta não teve o auxílio de ferramenta automática, sendo implementada na linguagem de programação C++. Optou-se por não utilizar ferramenta automática pelos seguintes pontos:

- A implementação das otimizações desenvolvidas em COMPILADOR1 foi aproveitada e melhorada.
- O conjunto de otimizações desenvolvido nesta ferramenta é específico para a arquitetura IPNoSys, com exceção da otimização de instruções desnecessárias, que será detalhando na seção 3.1.
- O código PDL é semelhante ao código LLVM-IR, uma vez que a IPNoSys não possui registradores, os valores intermediários das instruções são armazenados em variáveis temporárias. Essas variáveis somente podem ser utilizadas uma única vez. Característica que também ocorre no código LLVM-IR.

Visando a geração de códigos que melhore o desempenho de uma aplicação, é necessária a escolha da melhor sequência de otimizações para ser aplicada a um programa, porém isso não é trivial. Dessa forma, essa escolha fica a cargo do usuário por meio de opções da ferramenta (PEREIRA, 2011). Assim, a escolha de qual nível, disponibilizado por esta ferramenta, será aplicado ao código é de responsabilidade do usuário e ele fará essa opção de acordo com qual critério de desempenho – tempo de execução ou tamanho do código – deseja melhorar em sua aplicação.

Além da etapa de otimização foi implementada a geração de código contendo matrizes e/ou vetores, que será detalhada na seção 3.6.

Para utilização desta ferramenta, o usuário deverá salvar seu código em um arquivo com extensão ".c" e utilizar o comando ".\compilador prog otim" para a sua execução. 'prog' corresponde ao nome do arquivo que será traduzido e 'otim' o nível de otimização (O, O1 ou O2) que será aplicado ao código.

Os três níveis de otimização desenvolvidos são:

- O nível O consiste na tradução linha a linha do código intermediário LLVM-IR gerado pela Clang, ou seja, código gerado por esse nível não apresenta otimização.
- O nível O1 é composto por quatro otimizações: eliminação de instruções desnecessárias, diminuição de instruções *LOADs*, eliminação de instruções *STOREs* desnecessárias e eliminação de instruções *COPYs* desnecessárias.
- O nível O2 é formado pelas otimizações do nível anterior acrescido da geração de código paralelo. A paralelização de código escolhida para esse nível consiste na segunda forma de paralelismo exposta na subseção 2.2.6.3 do Capítulo 2. Foi escolhido paralelizar somente código que apresenta laço de repetição, objetivando um maior ganho de desempenho. Visto que para a IPNoSys, os laços de repetição são os locais onde o paralelismo é principalmente explorado (ARAUJO, 2012).

3.1 Otimização de eliminação de instruções desnecessárias

Para essa otimização, uma instrução é considerada desnecessária quando existem duas instruções de carregamento para uma mesma variável e entre essas duas instruções seu valor não é utilizado/lido como operando, apenas substituído, como ilustra o trecho de código da Figura 28 (a) para a variável 'a'. Seu código intermediário gerado pela Clang (Figura 28 (b)) conterá essas duas instruções.

Figura 28 – Trecho de código contendo instruções desnecessárias em: (a) C e (b) LLVM-IR

(a)

```
4  int main() {
5      int a = 0, b = 0, e = 1;
6      a = 1 + e;
7      e = e + b;
8      a = e + 20;
9      return 0;
10 }
```

(b)

```
6  define i32 @main() #0 {
7    %1 = alloca i32, align 4
8    %a = alloca i32, align 4
9    %b = alloca i32, align 4
10   %e = alloca i32, align 4
11   store i32 0, i32* %1
12   store i32 0, i32* %a, align 4
13   store i32 0, i32* %b, align 4
14   store i32 1, i32* %e, align 4
15   %2 = load i32* %e, align 4
16   %3 = add nsw i32 1, %2
17   store i32 %3, i32* %a, align 4
18   %4 = load i32* %e, align 4
19   %5 = load i32* %b, align 4
20   %6 = add nsw i32 %4, %5
21   store i32 %6, i32* %e, align 4
22   %7 = load i32* %e, align 4
23   %8 = add nsw i32 %7, 20
24   store i32 %8, i32* %a, align 4
25   ret i32 0
26 }
```

Como pode-se observar na Figura 28 (a), o valor para a variável ‘a’ está sendo armazenado nas linhas 6 e 8. Porém, entre essas duas linhas não existe instrução que utilize o valor de ‘a’ da linha 6. Tornando essa instrução (linha 6) totalmente dispensável e podendo ser eliminada. Contudo, como a tradução de Clang é fiel ao código do programa na linguagem de alto nível inserido como entrada, seu código intermediário gerado apresentará a tradução para todas as instruções, como mostra a Figura 28 (b).

Com isso, foi desenvolvida a otimização de eliminação de instruções desnecessárias. Essa é a única otimização pertencente no conjunto de otimizações desta ferramenta que é independente das características da arquitetura IPNoSys e pode ser utilizada em uma outra ferramenta que use a Clang para gerar seu *front-end*. Ela consiste na varredura do código intermediário com a finalidade de eliminar instruções cujo conteúdo de determinada variável é sobrescrita e não lida (não utilizada como operando) em instruções seguintes antes de ser sobrescrita. Sua implementação foi realizada utilizando o algoritmo da Figura 29.

Figura 29 – Algoritmo da otimização de eliminação de instruções desnecessárias

```
1  faça
2      code_teste2 = code_teste, apaga conteúdo de code_teste
3        se cont = 1
4          aux1 = aux
5        ler linha, faz uma cópia para linhaCopia e quebra linha quando
   encontra um delimitador de espaço e armazena em aux
6      se aux = store
7          code_teste = linhaCopia
8          incrementa cont
9          se cont = 2
10           se aux = aux1
11             se o conteúdo de code_teste3 for diferente de vazio
12               apaga conteúdo de code_teste3
13             code_teste3 = code_teste, apaga conteúdo de code_teste
14           senão
15             se o conteúdo de code_teste3 for diferente de vazio
16               mainCode = code_teste3, apaga conteúdo de code_teste3
17             mainCode = code_teste2, apaga conteúdo de code_teste2
18           cont = 1
19     senão se aux = load
20         quebra linha duas vezes
21         se aux = aux1
22           mainCode = code_teste2, apaga o conteúdo de code_teste2
23           mainCode = code_teste3, apaga o conteúdo de code_teste3
24         code_teste = linhaCopia
25         ler linha, faz uma cópia para linhaCopia e quebra linha quando
   encontra um delimitador de espaço e armazena em aux
26         enquanto aux for diferente de store
27           code_teste = linhaCopia
28           ler linha, faz uma cópia para linhaCopia e quebra linha quando
   encontra um delimitador de espaço e armazena em aux
29         se aux = store
30           code_teste = linhaCopia
31           incrementa cont
32           se cont = 2
33             se aux = aux1
34               se o conteúdo de code_teste3 for diferente de vazio
35                 apaga conteúdo de code_teste3
36               code_teste3 = code_teste, apaga conteúdo de code_teste
37             senão
38                se o conteúdo de code_teste3 for diferente de vazio
39                 mainCode = code_teste3, apaga conteúdo de code_teste3
40                mainCode = code_teste2, apaga conteúdo de code_teste2
41             cont = 1
42     se não tiver passado por nenhuma condição anterior
43       mainCode = linhaCopia
44 enquanto linha != final do arquivo
```

Fonte: AUTORIA PROPRIA (2016)

O algoritmo da Figura 29 realiza uma varredura no código intermediário. A leitura de sua primeira linha é realizada e armazenada em ‘linhaCopia’ (linha 5). Se a instrução for de armazenamento, a linha é armazenada em ‘code_teste’ (linha 7), a variável ‘cont’, que guarda a ocorrência da instrução de armazenamento, é incrementada (linha 8). Caso essa seja a segunda ocorrência da instrução de armazenamento (linha 9), é verificado se o operador (aux) da instrução corrente é igual a ‘aux1’ (linha 10), que guarda o conteúdo do operando de uma instrução de armazenamento (linha 4). Em caso de verdade, é verificado se existe conteúdo em ‘code_teste3’ (linha 11), que corresponde a uma variável auxiliar que guarda trechos do código intermediário assim como ‘code_teste2’. Se tiver, esse conteúdo é apagado (linha 12). O

conteúdo de ‘code_teste’ é armazenado em ‘code_teste3’ e apagado (linha 13). No caso de ‘aux’ e ‘aux1’ serem diferentes, é testado se existe conteúdo em ‘code_teste3’ (linha 15). Em caso positivo esse conteúdo é armazenado em ‘mainCode’, que contém o código resultante dessa otimização, e apagado (linha 16). Em seguida, o conteúdo de ‘code_teste2’ também é armazenado em ‘mainCode’ e apagado (linha 17).

Caso contrário, se a instrução for de carregamento (linha 19) seu operando (aux) é comparado com ‘aux1’. Caso seja verdade, todo conteúdo das variáveis ‘code_teste2’ e ‘code_teste3’ é armazenado na variável ‘mainCode’ (linhas 22 e 23). O conteúdo das variáveis ‘code_teste2’ e ‘code_teste3’ são apagados. Em seguida, o conteúdo de ‘linhaCopia’ será armazenado em ‘code_teste’ (linha 24). A próxima linha é lida e armazenada em ‘linhaCopia’ (linha 25). Enquanto não for uma instrução de carregamento (linha 26), linha é armazenada em ‘code_teste’ (linha 27) e é feita a leitura da próxima linha (linha 28). Se a instrução for de armazenamento, o processo é o mesmo que o apresentado no parágrafo anterior.

Se o conteúdo da linha lida não é uma instrução de armazenamento nem de carregamento, seu conteúdo é inserido em ‘mainCode’ (linha 43). E esse processo é repetido enquanto o fim do aquivo não for encontrado.

Para exemplificar a aplicação dessa otimização foi gerado o código intermediário da Figura 28, que corresponde a Figura 30 (a), cujo conteúdo equivale a função ‘*main*’, tendo seu cabeçalho e rodapé omitidos. A Figura 30 (b) apresenta o código otimizado após a aplicação da otimização de eliminação de instruções desnecessárias. Como explicado anteriormente, a linha 6 da Figura 28 é desnecessária e seu código intermediário corresponde as linhas 14 a 16 da Figura 30 (a). Observe que na Figura 30 (b) essas linhas foram eliminadas gerando um código com a ausência de instruções desnecessárias.

Figura 30 - (a) LLVM-IR da Figura 28 (b) Após a eliminação de instruções desnecessárias

```
5 define i32 @main() #0 {
6   %1 = alloca i32, align 4
7   %a = alloca i32, align 4
8   %b = alloca i32, align 4
9   %e = alloca i32, align 4
10  store i32 0, i32* %1
11  store i32 0, i32* %a, align 4
12  store i32 0, i32* %b, align 4
13  store i32 1, i32* %e, align 4
14  %2 = load i32* %e, align 4
15  %3 = add nsw i32 1, %2
16  store i32 %3, i32* %a, align 4
17  %4 = load i32* %e, align 4
18  %5 = load i32* %b, align 4
19  %6 = add nsw i32 %4, %5
20  store i32 %6, i32* %e, align 4
21  %7 = load i32* %e, align 4
22  %8 = add nsw i32 %7, 20
23  store i32 %8, i32* %a, align 4
24  ret i32 0
25 }
```

(a)

```
5 define i32 @main() #0 {
6   %1 = alloca i32, align 4
7   %a = alloca i32, align 4
8   %b = alloca i32, align 4
9   %e = alloca i32, align 4
10  store i32 0, i32* %1
11  store i32 0, i32* %a, align 4
12  store i32 0, i32* %b, align 4
13  store i32 1, i32* %e, align 4
14  %4 = load i32* %e, align 4
15  %5 = load i32* %b, align 4
16  %6 = add nsw i32 %4, %5
17  store i32 %6, i32* %e, align 4
18  %7 = load i32* %e, align 4
19  %8 = add nsw i32 %7, 20
20  store i32 %8, i32* %a, align 4
21  ret i32 0
22 }
```

(b)

Fonte: AUTORIA PROPRIA (2016)

3.2 Otimização de diminuição de instruções *LOADs*

Em COMPILADOR1 foi desenvolvida uma otimização que visa a diminuição de instruções *LOADs*. Essa otimização substitui as instruções *LOADs* por instruções *COPYs*, como foi apresentado na seção 2.4 do Capítulo 2. Porém, ela ainda gera códigos contendo instruções de carregamento de variáveis declaradas no código de entrada.

Em vista disso e objetivando a eliminação das instruções de carregamento desnecessárias, foi desenvolvida a otimização de diminuição de instruções *LOADs*. Nessa otimização, todas as instruções de carregamento também são substituídas por instruções *COPY*.

Se uma variável é lida apenas uma vez em todo o código, seu valor é inserido na posição da instrução em que a variável é lida. Evitando assim que seja inserida uma instrução *COPY* desnecessária. Caso uma variável seja lida mais de uma vez, será inserido no código PDL uma instrução *COPY* contendo dois temporários e o valor dessa variável será copiado para esses temporários.

Com esta otimização, a instrução *LOAD* é totalmente eliminada em códigos que não contenham operações com vetor e/ou matriz. Pois no caso em que o código possui operação com vetores e/ou matrizes, é necessário carregar o valor de uma determinada posição e isso só é possível com a utilização da instrução *LOAD*.

Para sua implementação foi necessária uma varredura no código para o armazenamento da ocorrência de cada variável do programa. Essa varredura foi desenvolvida segundo o algoritmo da Figura 31.

Figura 31 – Algoritmo para o armazenamento da variável e sua ocorrência

```
1  faça
2      ler linha, quebra linha três vezes quando encontra um delimitador de
       espaço e armazena em aux
3      se aux = icmp ou aux = add ou aux = sub ou aux = mul ou aux = sdiv
4          quebra linha três vezes quando encontra um delimitador de espaço e
           armazena em aux
5          se aux não for um número
6              insere aux numa fila e o valor de sua ocorrência é incrementado
7              quebra linha quando encontra um delimitador de espaço e armazena em
                aux
8              se aux não for um número
9                  insere aux numa fila e o valor de sua ocorrência é incrementado
10 enquanto linha != final do arquivo
```

Fonte: AUTORIA PROPRIA (2016)

No algoritmo da Figura 31, é realizada uma varredura no código do programa. Ao encontrar uma instrução de comparação (*icmp*) ou aritmética (*add* ou *sub* ou *mul* ou *sdiv*) (linha 3), a *linha* é quebrada três vezes para encontrar o primeiro operando dessa instrução (linha 4). Caso esse operando não seja um número (linha 5), a variável é armazenada numa estrutura de dados e o valor de sua ocorrência é incrementado (linha 6). Em seguida, a *linha* é quebrada novamente na busca do segundo operando (linha 7). Se a variável não for um número, ela é inserida e o valor de sua ocorrência incrementado (linha 9). Esse processo se repete até encontrar o final do arquivo (linha 10).

O algoritmo da otimização de diminuição de instruções *LOADs* é apresentado na Figura 32. O código intermediário gerado pela Clang é lido. Ao encontrar uma instrução *LOAD* (linha 3), o operando dessa instrução é inserido numa fila (linha 5). Porém, caso a instrução seja de comparação ou aritmética (linha 6), é verificado se o primeiro operando dessa instrução é um número (linha 8) ou uma variável (linha 10) criada pela Clang. Se o operando for um número, esse valor é inserido no código PDL. Caso contrário, é desenfileirada a variável (linha 11) que corresponde a esse operando e realizada uma busca pela ocorrência dessa variável (linha 12). Se o retorno dessa busca for um (linha 13), ou seja, a variável aparece somente uma vez no código, é retornada uma cópia dessa variável que está armazenada numa árvore binária de busca. Essa árvore já era utilizada em COMPILADOR1. Para o caso em que a ocorrência da variável for maior que um, é chamada a função '*criar_copys*' (linha 16). Essa função insere no

código PDL uma instrução *COPY* contendo dois temporários. O valor que será copiado nessa instrução será uma cópia da variável que está armazenada na árvore binária de busca. E em seguida é gerada a instrução correspondente em PDL, cujo primeiro operando será uma cópia da árvore (linha 17). Para o segundo operando, o processo é o mesmo do primeiro. Esses passos se repetem até encontrar o final do arquivo.

Figura 32 - Algoritmo da otimização da diminuição de instruções *LOADs*

```
1  faça
2      ler linha, quebra linha três vezes quando encontra um delimitador de espaço e
       armazena em aux
3      se aux = lood
4          quebra linha duas vezes quando encontra um delimitador de espaço e
           armazena em aux
5          insere aux numa fila
6      senão se aux = icmp ou aux = add ou aux = sub ou aux = mul ou aux = sdiv
           quebra linha três vezes quando encontra um delimitador de espaço e
7          armazena em aux
8          se aux for um número
9              insere esse valor na instrução PDL
10         senão
11             desenfileira e armazena em aux1
12             busca a ocorrência da variável armazenada em aux1
13             se ocorrência = 1
14                 retorna a cópia armazenada na árvore binária de busca
15             senão
16                 chama a função criar_copys
17                 insere a cópia armazenada na árvore binária de busca na instrução PDL
18         quebra linha quando encontra um delimitador de espaço e armazena em aux
19         se aux for um número
20             insere esse valor na instrução PDL
21         senão
22             desenfileira e armazena em aux1
23             busca a ocorrência da variável armazenada em aux1
24             se ocorrência = 1
25                 retorna a cópia armazenada na árvore binária de busca
26             senão
27                 chama a função criar_copys
28                 insere a cópia armazenada na árvore binária de busca na instrução PDL
29 enquanto linha != final do arquivo
```

Fonte: AUTORIA PROPRIA (2016)

Para ilustrar a aplicação da otimização de diminuição de instruções *LOADs*, o código da Figura 33 (a) foi traduzido para PDL na ferramenta COMPILADOR1 mostrado em (b) e neste compilador C2PDL, exposto em (c). Observe que o tamanho do código gerado por COMPILADOR1 é bem maior que o código da ferramenta aqui proposta (Figura 33 (c)). Isso ocorre porque COMPILADOR1 faz uma tradução direta do código intermediário gerado pela Clang.

Figura 33 – (a) Programa em C. Correspondente em PDL gerado por: (b) COMPILADOR1 e (c) C2PDL

(a)

```
1 int main()
2 {
3         int a = 0, b = 0, e = 1;
4         b = e + e;
5         a = e + a;
6         return 0;
7 }
```

(b)

```
1 PROGRAM adicao
2 DATA
3         z1 = 0
4         a = 0
5         b = 0
6         e = 0
7 PACKAGE main
8 ADDRESS MAU_0
9         COPY c2 c1;
10                0;
11        COPY c4 c3;
12                0;
13        COPY c6 c5;
14                1;
15        COPY %2;
16                c5;
17        COPY %3;
18                c6;
19        ADD %4;
20                %2
21                %3;
22        COPY c9 c8;
23                %4;
24        LOAD MAU_0 c10;
25                e;
26        COPY c11 c12;
27                c10;
28        COPY %5;
29                c11;
30        COPY %6;
31                c3;
32        ADD %7;
33                %5
34                %6;
35        COPY c14 c13;
36                %7;
37        STORE MAU_0;
38                c13
39                a;
40        STORE MAU_0;
41                c9
42                b;
43        STORE MAU_0;
44                c12
45                e;
46        STORE MAU_0;
47                c1
48                z1;
49 END
50 END_PROGRAM
```

(c)

```
1 PROGRAM adicao
2 DATA
3         a = 0
4         b = 0
5         e = 1
6 PACKAGE main
7 ADDRESS MAU_0
8         COPY c2 c1;
9                 1;
10        COPY c4 c3;
11                c2;
12        ADD %4;
13                c1
14                c3;
15        ADD %7;
16                c4
17                0;
18        STORE MAU_0;
19                %7
20                a;
21        STORE MAU_0;
22                %4
23                b;
24 END
25 END_PROGRAM
```

Fonte: AUTORIA PROPRIA (2016)

Observe que na seção DATA do código PDL da Figura 33 (b) todas as variáveis são declaradas e inicializadas com o valor 0, já em (c) as variáveis são inicializadas com seus valores reais declarados no seu código em C. Note que na Figura 33 (b) há a declaração da variável ‘z1’, ela corresponde a variável criada pela Clang quando a função principal é declarada como *int*. Essa variável foi eliminada da geração de código deste compilador otimizante, visto que para IPNoSys ela é desnecessária.

São criadas instruções *COPY* contendo dois temporários para as inicializações das variáveis do programa, localizadas da linha 9 a 14 da Figura 33 (b). O valor que é copiado para os temporários corresponde ao valor da variável. Para exemplificar, a inicialização da variável ‘e’ (linha 13 e 14) corresponde a uma instrução *COPY*, que realiza uma cópia do valor um (1) para os temporários ‘c5’ e ‘c6’.

Na Figura 33 (b), da linha 15 a 48 tem o código correspondente ao corpo da função *main*. Quando uma variável é lida, é inserida uma instrução *COPY* para uma variável temporária criada pela Clang inicializando com ‘%’. Por exemplo, para a primeira leitura da variável ‘e’, o código gerado está ilustrado nas linhas 15 e 16. Nele, há a instrução *COPY* para a variável ‘%2’ e o valor copiado é ‘c5’, que corresponde a uma das cópias geradas durante a inicialização dessa variável. E quando todas as cópias da variável são consumidas, há a inserção da instrução *LOAD*, como é o caso das linhas 24 e 25. Nessas linhas, as cópias da variável ‘e’ (c5 e c6) foram consumidas anteriormente e essa variável é lida novamente. Para esse caso, é gerada uma instrução de carregamento dessa variável para a variável temporária ‘c10’. Em seguida, a variável temporária é copiada para dois temporários (linhas 26 e 27).

Já no código da Figura 33 (c), quando uma variável é lida, não é inserida uma instrução para ela, como ocorria em COMPILADOR1. Antes da variável ser utilizada é criada a quantidade de cópias necessárias para a instrução corrente. E sempre que a variável for utilizada é removida uma cópia armazenada na árvore. Para exemplificar, considere a instrução de adição da linha 4 da Figura 33 (a), nela a variável ‘e’ é lida duas vezes, somada e o resultado armazenado em ‘b’. O código correspondente dessa adição na Figura 33 (c) está nas linhas 12 a 14. Observe que são removidas duas cópias (c1 e c3) dessa variável, que foram realizadas anteriormente nas linhas 8 a 11.

Quando uma determinada variável apresenta apenas uma ocorrência no código, como é o caso da variável ‘a’ desse exemplo, o seu valor é inserido na instrução que utiliza a variável. Observe as linhas 15 a 17 da Figura 33 (c). Nesse código, há a adição da variável ‘e’ com ‘a’,

para a variável ‘e’ é inserida nessa instrução sua cópia, já para ‘a’, é inserido o valor 0, que corresponde ao seu valor declarado para o programa.

3.3 Otimização de eliminação de instruções *STORE* desnecessárias

Para a otimização da eliminação de instruções *STORE* desnecessárias, foi melhorada a versão já existente em COMPILADOR1. A versão de COMPILADOR1 faz o armazenamento para todas as variáveis declaradas somente ao final do código, evitando assim que instruções de armazenamento intermediárias sejam utilizadas, isto é, se no código intermediário existem dois ou mais armazenamentos para uma determinada variável, somente no último será gerada uma instrução *STORE*. Assim, a versão da otimização deste trabalho considera se o valor de determinada variável foi modificado no decorrer do código. Dessa forma, nessa otimização são geradas instruções de carregamentos somente para as variáveis, cujo valor foi modificado.

Na sua implementação foi necessária uma varredura no código intermediário a fim de conhecer quais as variáveis foram alteradas. Dessa forma, sempre que uma instrução de carregamento é encontrada, a *flag* ‘modificado’ correspondente a variável na tabela de dispersão é marcada como verdadeira.

O algoritmo da Figura 34 foi utilizado para o desenvolvimento da otimização de eliminação de instruções *STORE* desnecessárias. Quando uma instrução de carregamento é encontrada (linha 3), o valor que está sendo armazenado na variável é inserido na árvore binária de busca juntamente com sua respectiva variável (linha 6). No caso em que o fim do programa, que corresponde ao fecha chaves, ou do bloco básico é encontrado (linha 7), para todas as variáveis armazenadas na árvore são verificadas se seus valores foram modificados no decorrer do código. Em caso de verdade, é inserido ao código PDL uma instrução *STORE* para essa variável, cujo valor que será armazenado nela será uma cópia da árvore.

Figura 34 - Algoritmo da otimização de eliminação de instruções *STORE* desnecessárias

```
1  faça
2      ler linha, quebra linha três vezes quando encontra um delimitador de
       espaço e armazena em aux
3      se aux = store
4          quebra linha duas vezes quando encontra um delimitador de espaço para
           encontrar o valor que será armazenado na variável
5          quebra linha mais duas vezes quando encontra um delimitador de espaço
           para encontrar a variável
6          insere o valor armazenado na árvore binária de busca
7      se aux encontrar o fim do bloco básico ou fim do programa
8              para todas as variáveis armazenada na árvore binária de busca faça
9                  se a variável foi modificada
10                     insere instrução STORE para a variável contendo uma cópia da
                       árvore
11 enquanto linha != final do arquivo
```

Fonte: AUTORIA PROPRIA (2016)

Tomando como exemplo o código da Figura 33, observe que em (b) são geradas quatro instruções *STOREs* (linha 37 a 48) para as quatro variáveis declaradas na seção DATA desse código. Porém, a variável ‘e’ não foi modificada, gerando uma instrução de armazenamento desnecessária para ela, além da instrução para a variável ‘z1’. Observe que em (c) isso não ocorre, apenas são geradas instruções de armazenamento (linha 18 a 23) para as variáveis que tiveram seus valores alterados no decorrer do código.

3.4 Otimização de eliminação de instruções *COPY* desnecessárias

O trabalho COMPILADOR1 faz uma tradução direta do código intermediário gerado pela Clang para a linguagem PDL, gerando instruções *COPY* cujos operandos não são utilizados posteriormente, visto que em PDL os temporários criados somente podem ser utilizados uma única vez. Caso um determinado temporário seja criado e não utilizado, ao executar esse código, o montador mostrará um erro semântico. Para solucionar esse problema foi criada a otimização de eliminação de *copys* desnecessários.

A implementação dessa otimização consiste inicialmente em uma varredura no código gerado em PDL, no qual todos os temporários criados são armazenados juntamente com o número de vezes que aparecem no código. E seus temporários são enfileirados.

Em seguida, outras varreduras são realizadas na busca pelo temporário que aparece uma única vez, conforme o algoritmo da Figura 35. O temporário cuja ocorrência é um (1) é proveniente de uma instrução *COPY*. Quando esse temporário é encontrado (linha 7), o valor do outro temporário da instrução *COPY* é armazenado na variável ‘temp’ (linha 8) e o valor a

ser copiado é armazenado em ‘temp1’ (linha 9). Realizando assim eliminação dessa instrução. Em seguida é feita uma busca no código pela variável temporária armazenada em ‘temp’, que estará localizada em alguma instrução aritmética, de armazenamento ou de envio (*SEND*) (linha 12). Se um dos operandos da instrução for ‘temp’ (linha 13), ele será substituído pelo valor de ‘temp1’ (linha 14) gerando a instrução com o operando atualizado. Caso contrário, a linha contendo a instrução é copiada para a variável ‘code’.

Figura 35 - Algoritmo da otimização de eliminação de instruções *COPY* desnecessárias

```
1  desenfileira temporário e armazena em temp
2  enquanto temp for diferente de vazio faça
3   se temp possui uma ocorrência
4      faça
5       ler linha
6       se linha for uma instrução COPY
7        se um de seus temporários é igual a temp
8           temp recebe o valor do outro temporário
9           ler próxima linha, quebra linha quando encontra um delimitador de
             espaço e armazena em temp1
10       senão
11          code recebe linha, próxima linha é lida e armazenada em code
12      senão se encontrar uma instrução aritmética ou armazenamento ou envio
13       se um dos operandos é iguais a temp
14          o operando é substituído por temp1
15       senão
16          code recebe linha
17   enquanto linha é diferente do fim do arquivo
18 desenfileira temporário
```

Fonte: AUTORIA PROPRIA (2016)

Retornando ao exemplo do código da Figura 33 (b), observe que as variáveis temporárias ‘c2’, ‘c4’, ‘c8’ e ‘c14’, criadas em instruções *COPY*, aparecem apenas uma vez em todo o código. Caso esse código seja submetido ao montador, ele apontará um erro semântico para cada uma dessas variáveis. E o código de (c), há a utilização de todas as variáveis temporárias criadas.

3.5 Paralelização de código

Esta otimização somente é aplicada a programas que contenham em seus códigos laços de repetição. Nela, todos os laços são replicados em quatro fluxos de execução, cada um para um endereço de MAU diferente (MAU_0, MAU_1, MAU_2 e MAU_3), visto que a IPNoSys possui quatro MAUs, como foi apresentado no Capítulo 2. Além disso, nessa otimização a quantidade de iterações de determinado laço é dividida em quatro pacotes, para que cada MAU possa executar uma parte do laço.

Para sua implementação, o código PDL gerado pelo nível O1 foi utilizado como entrada para este nível. Inicialmente, é realizada uma varredura no código. Ao longo dessa varredura, o código é dividido em partes para a sua posterior utilização. E cada parte é armazenada em um arquivo texto diferente. A primeira parte consiste do início do programa até encontrar a declaração do primeiro pacote, já as partes seguintes compreendem os códigos dos pacotes do programa. Durante essa varredura, os valores das iterações dos laços são armazenados e divididos por 4, caso esse valor seja um número e seja maior ou igual a 4. O resultado dessa divisão será chamado de "*valor_iterador*" e corresponde a quantidade de iterações que cada fluxo de execução terá.

O arquivo texto, que corresponde ao pacote inicial chamado de '*main*', é lido. Nele é aplicado o algoritmo descrito na Figura 36. Ao encontrar uma instrução END (linha 3), é gerada uma instrução *SYNEXEC* com o endereço de MAU 0. Essa instrução contém uma lista de pacotes, cujos sinais de sincronismo são necessários esperar para que o pacote que finalizará o programa seja injetado (linha 4). Essa lista é composta pelos quatro pacotes que correspondem aos quatro fluxos da divisão do laço de repetição. Para o caso de uma instrução de envio (linha 5) ser lida, são criadas 4 instruções *SEND* para uma MAU diferente e modificados os valores que são enviados para as variáveis. Para o primeiro pacote, o valor não é modificado e geralmente corresponde ao valor 0, mas para os pacotes posteriores, esse valor será a soma do valor do pacote anterior com *valor_iterador*. Essa modificação no valor só ocorrerá para o caso do iterador ser um número, caso contrário essa instrução é apenas replicada. Quando uma instrução EXEC é encontrada são geradas quatro instruções *EXEC* para as quatro MAUs. Cada instrução faz a injeção de um dos pacotes que corresponde aos quatro fluxos.

Figura 36 - Algoritmo da paralelização de código para o pacote *main*

```
1  ler linha
2  faça
3  | se linha for uma instrução END
4  | | insere no arquivo a instrução SYNEXEC para os quatro pacotes
   | | correspondentes aos laços e o pacote que finaliza o programa
5  | senão se linha for a instrução SEND
6  | |  insere quatro instruções SEND para as quatro MAUs
7  | senão se linha for a instrução EXEC
8  | |  insere quatro instruções EXEC para as quatro MAUs
9  | ler linha
10 enquanto linha é diferente do final do arquivo
```

Fonte: AUTORIA PROPRIA (2016)

Para os pacotes que correspondem aos laços de repetição, o algoritmo da Figura 37 é aplicado. Esse algoritmo faz a replicação do código em quatro pacotes. Cada pacote apresentará

o mesmo conteúdo com a diferença nos endereços das MAUs e nos valores dos iteradores do laço. Quando uma instrução de comparação (linha 3) for lida, será verificada se um de seus iteradores é um número (linha 4). Para o caso de ser verdade, essa instrução é inserida em quatro pacotes com o valor do iterador substituído. No primeiro pacote, o valor do iterador será substituído por 'valor_iterador'. Já para os pacotes seguintes, o valor do iterador de determinado pacote será a soma de *valor_iterador* com o novo valor do iterador do pacote anterior (linha 5). No caso do valor do iterador não ser um número (linha 6), essa instrução é apenas replicada para os quatro pacotes (linha 7). Já se uma instrução *SEND*, *LOAD* ou *STORE* for lida (linha 8), a instrução é inserida nos quatro pacotes, modificando apenas o endereço da MAU (linha 9). Porém, se a instrução lida for *EXEC* (linha 10), é verificado se o nome do pacote que essa instrução injetará inicia com 'end_' (linha 11). Em caso de verdade, são inseridas instruções *SYNC* para os pacotes (linha 12), que corresponde ao sinal de sincronismo que a instrução SYNEXEC espera para injetar o pacote que encerrará o programa. Caso contrário, a instrução é replicada nos pacotes, apenas modificando os endereços das MAUs (linha 14). Para o caso da instrução lida não ser nenhum dos casos citados anteriormente (linha 15), a instrução é replicada para os quatro pacotes (linha 6). Esse processo se repete até encontrar o final do arquivo.

Figura 37 - Algoritmo da paralelização de código dos pacotes com laços de repetição

```
1  ler linha
2  faça
3    se linha for uma instrução de comparação
4      se iterador for um número
5        instrução é inserida em quatro pacotes com o valor do iterador
         substituído pela soma do valor do iterador do pacote anterior
         com valor_iterador
6      senão
7        instrução é replicada em quatro pacotes
8    senão se linha for a instrução SEND ou LOAD ou STORE
9      instrução é inserida em quatro pacotes com o endereço da MAU diferente
10   senão se linha for a instrução EXEC
11     se o nome do pacote a ser injetado iniciar com 'end_'
12       insere instrução SYNC para o pacote
13     senão
14       instrução é inserida em quatro pacotes com o endereço da MAU diferente
15   senão
16     instrução é replicada para os quatro pacotes
17   ler linha
18 enquanto linha é diferente do final do arquivo
```

Fonte: AUTORIA PROPRIA (2016)

Nessa otimização, também é feita uma modificação nos nomes dos pacotes que correspondem aos laços, sendo acrescido os valores de 1 a 4, nessa ordem, para cada replicação do laço.

Ao final da aplicação dos algoritmos aos seus respectivos arquivos textos, é realizada a combinação desses arquivos em apenas um e gerado a saída dessa otimização, o código paralelo e otimizado do programa inserido como entrada.

O código em C da Figura 38 (a) corresponde a aplicação de descompressão de dados RLE para um vetor com 8 posições. Seu código correspondente em PDL, gerado como saída do nível O1 desta ferramenta, é apresentado na Figura 38 (b). Observe que nessa aplicação seu código PDL é composto por 4 pacotes: *main*, *loop2*, *loop1* e *end_loop2*. No pacote '*main*', é realizado o envio de '0' para a variável '*index*' do pacote '*loop2*', que é chamado em seguida. Essa variável corresponde a uma posição do vetor de entrada 'b'. O pacote '*loop2*' equivale ao laço mais externo. Nele, um elemento do vetor de entrada 'b' é enviado para a variável '*length*' do pacote '*loop1*'. Essa variável consiste na quantidade de vezes que determinado elemento deverá se repetir. A variável '*index*' é incrementada na busca do próximo elemento do vetor que será enviado para a variável '*run*' do pacote *loop1*. '*run*' corresponde ao elemento que será replicado. Os valores das variáveis '*i*' e '*index*' são enviados para '*loop1*'. Em '*loop2*' também é testado se o vetor de entrada chegou ao fim. Em caso de verdade, as operações anteriores são executadas e o pacote '*loop1*' é chamado. Caso contrário, o programa chamará o pacote '*end_loop2*' que finalizará o programa através da instrução *EXIT*. O pacote '*loop1*' corresponde ao laço mais interno. Nesse pacote, a variável '*run*' será replicada a quantidade de vezes que '*length*' armazena. A cada iteração do laço, '*run*' é guardada em uma posição do vetor de saída 'a', '*length*' é decrementada e a variável 'i', que corresponde a posição do vetor 'a', é incrementada. Os valores das variáveis '*length*', '*run*', 'i' e '*index*' são enviados para o próprio pacote '*loop1*'. Quando '*length*' for 0, ou seja, o elemento já estiver sido replicado a quantidade de vezes que essa variável guardava, '*index*' é incrementada e enviada juntamente com o valor de 'i' para o pacote '*loop2*', que é chamado em seguida.

Figura 38 – Código da aplicação de descompressão RLE, em: (a) C e (b) PDL

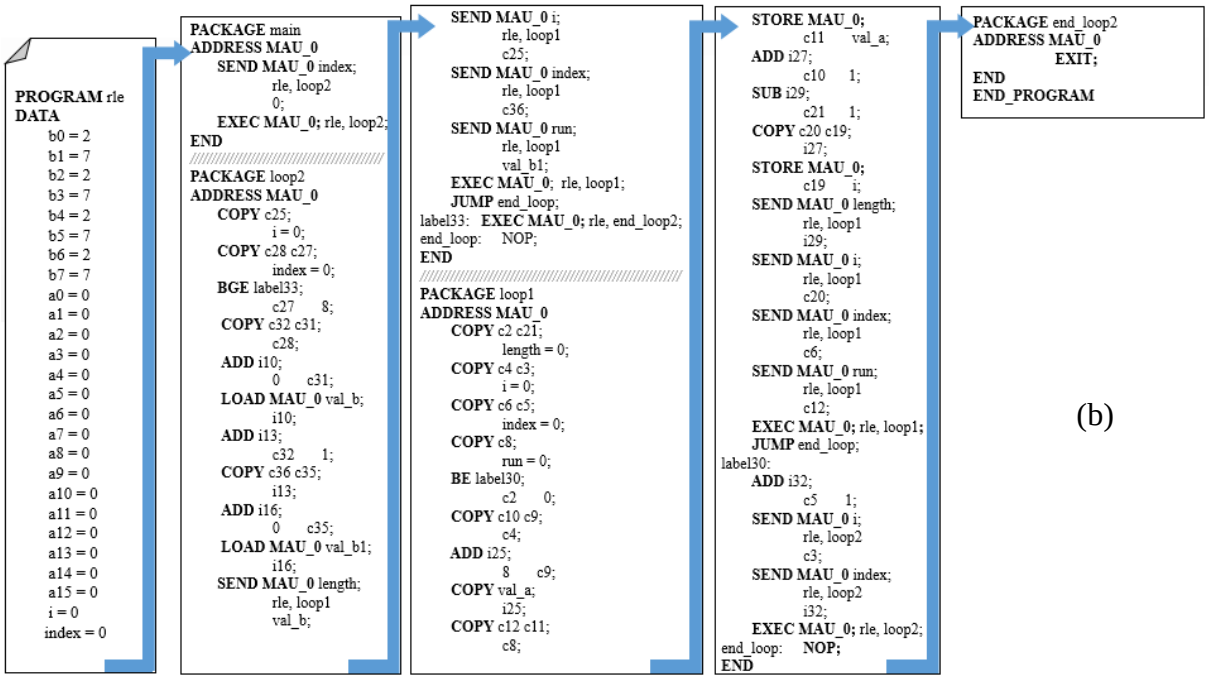
```c
int main()
{
    int b[8] = {2, 7, 2, 7, 2, 7, 2, 7};
    int a[16];
    int i = 0;
    int index = 0;
    int run, length;

    while(index < 8)
{
    length = b[index];
    index++;
    run = b[index];

        while (length != 0){
            a[i] = run;
            i++;
            length = length - 1;
        }
        index++;

}
return 0;
}
```

(a)



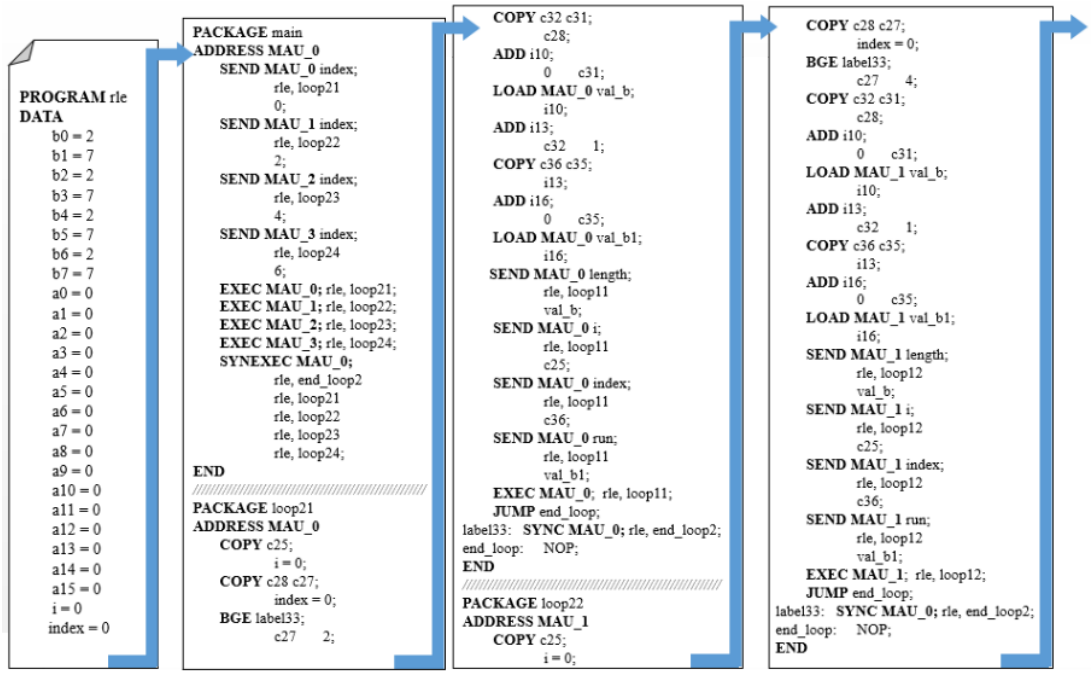
(b)

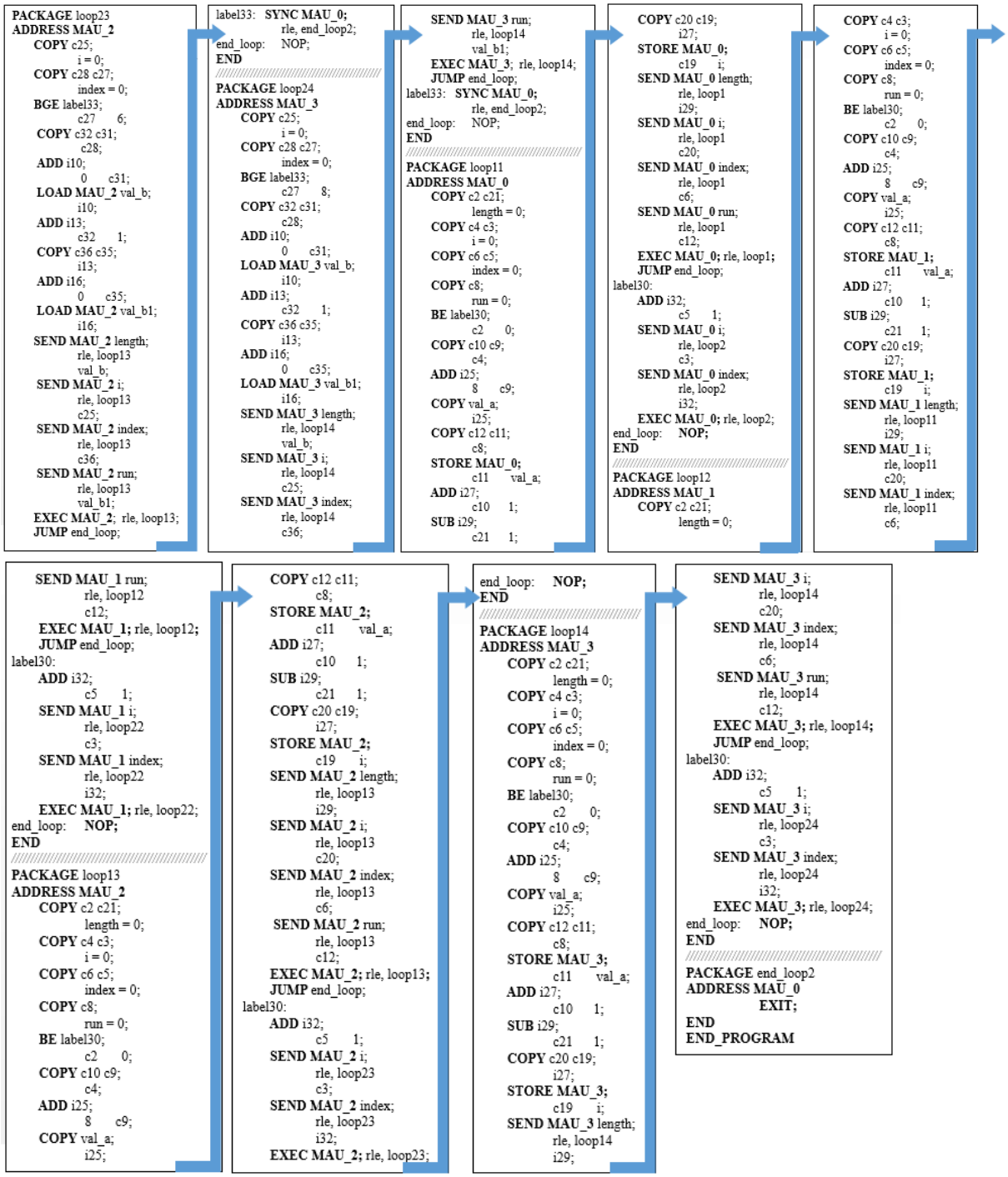
Fonte: AUTORIA PROPRIA (2016)

Com a aplicação da otimização de paralelização de código ao programa da Figura 39 é gerado o código da Figura 39, que corresponde ao seu equivalente paralelo. Observe que esse programa apresenta 10 pacotes: *main*, *loop21*, *loop22*, *loop23*, *loop24*, *loop11*, *loop12*, *loop13*, *loop14* e *end_loop2*. No pacote *main*, há a replicação da instrução de envio com valores ‘0’, ‘2’, ‘4’ e ‘6’ para a variável ‘*index*’. Esses valores correspondem a divisão do laço de repetição do pacote ‘*loop2*’. Cada pacote realizará duas iterações. Em seguida, os quatro pacotes que correspondem a replicação do laço mais externo são injetados. Por fim, tem-se uma instrução *SYNEXEC* contendo a lista de pacotes que o pacote ‘*end_loop2*’ esperará os sinais de sincronismo para finalizar o programa. Observe ainda que o conteúdo dos pacotes *loop21*,

loop22, *loop23* e *loop24* é o mesmo do pacote *loop2* da Figura 38, com a diferença nos endereços das MAUs, nomes dos pacotes que são acrescidos os valores de 1 a 4 e o valor do iterador que corresponde a soma do valor do iterador do pacote anterior com a divisão do valor do iterador original por 4. Ao final de cada replicação do pacote '*loop2*', a instrução *EXEC* que injetava o pacote '*end_loop2*' é substituída pela instrução *SYNC*, para o envio dos sinais de sincronismos que a instrução *SYNEXEC* espera. Percebe-se também que o conteúdo dos pacotes *loop11*, *loop12*, *loop13* e *loop14* também corresponde ao mesmo do pacote *loop1* da Figura 38, sendo modificados os endereços das MAUs e nomes dos pacotes. O valor do iterador é o mesmo, visto que esse valor corresponde a 0, e para esse caso a instrução é apenas replicada.

Figura 39 - Código paralelo e otimizado da aplicação RLE





Fonte: AUTORIA PROPRIA (2016)

3.6 Geração de código para matrizes e vetores

Este compilador/otimizador também realiza a geração de código PDL para vetores e matrizes, uma vez que em COMPILADOR1 essa funcionalidade não foi desenvolvida. Para isso, foi necessário realizar uma modificação no código intermediário gerado pela Clang para que fosse possível a geração de código em PDL. Como apresentado no Capítulo 2, vetores e

matrizes em PDL são implementados seguindo fórmulas. Com isso, o código foi adequado a essas fórmulas.

Na Figura 40 (a) é apresentado um exemplo de código em C contendo uma operação com o vetor ‘vet1’. Nesse código, a cada iteração do laço um elemento do vetor é atribuído a variável ‘g’. A Figura 40 (b) contém o fragmento de código gerado pela Clang para a declaração e inicialização do vetor ‘vet1’ com duas posições, contendo como elementos o valor 3. Na linha 5 é feita a declaração dos elementos de ‘vet1’, o primeiro trecho entre colchetes indica que o vetor possui tamanho 2 e seu tipo é inteiro. Já no segundo trecho, são apresentados os elementos desse vetor. Na linha 9 é realizada a declaração do vetor, também indicando seu tipo e seu tamanho. Em (c) é apresentado o correspondente código PDL de (b), cuja declaração e inicialização ocorre na seção DATA, onde cada elemento do vetor corresponderá a uma variável nessa seção. Cada variável inicia com o nome do vetor seguida da posição do elemento no vetor. Para a primeira posição do vetor ‘vet1’ o nome da variável correspondente é ‘vet10’, como pode-se observar em (c).

Figura 40 – (a) Operação com vetor vet1 em C. (b) Declaração e inicialização do vetor e (c) correspondente em PDL.

```c
int main() {

  int vet1[2]={3,3};
  int i;
  int g;

  for(i=0; i<2; i++){
     g = vet1[i];
  }
  return 0;
}
```

(a)

```
5 @main.vet1 = private unnamed_addr constant [2 x i32] [i32 3, i32 3], align 4
6
7 ; Function Attrs: nounwind uwtable
8 define i32 @main() #0 {
9   %vet1 = alloca [2 x i32], align 4
```

(b)

```
DATA
        vet10 = 3
        vet11 = 3
```

(c)

Fonte: AUTORIA PROPRIA (2016)

O código gerado pela Clang para a leitura de um elemento do vetor ‘vet1’ é mostrado na Figura 41 (a). Nele, é realizada a leitura da variável ‘i’ e armazenada em ‘%7’. ‘i’ corresponde a posição do vetor que se deseja ler. Em seguida, a instrução ‘*sext*’ estende o valor inteiro de 32 bits armazenado em ‘%7’ para um inteiro de 64 bits. A instrução ‘*getelementptr*’ retorna o endereço do elemento da posição que está armazenada no rótulo ‘%8’. E por fim, o

elemento é lido. Esse código foi adequado à fórmula: endereço_do_elemento [i] = endereço_base + i, onde ‘endereço_base’ corresponde ao endereço da primeira posição do vetor na seção DATA e ‘i’ a posição do vetor. Com isso, o código da Figura 41 (a) foi substituído pelo código de (b). Nesse código, também é realizada a leitura da variável ‘i’ e seu armazenamento em ‘%7’. Em seguida, é inserida a adição do valor 0, que corresponde a primeira posição de ‘vet1’, com a variável ‘i’ e esse valor é armazenado em ‘%9’, que conterá o endereço do elemento que se deseja ler. Em seguida, uma instrução *LOAD* é inserida para ler o conteúdo do endereço guardado em ‘%9’ que será carregado para a variável ‘val_vet1’. Essa variável armazenará o elemento lido. Em (c) tem-se o código PDL correspondente ao código de (b).

Figura 41 – Leitura de um elemento do vetor vet1: (a) gerado pela Clang, (b) código intermediário modificado e (c) em PDL

```
%7 = load i32* %i, align 4
%8 = sext i32 %7 to i64
%9 = getelementptr inbounds [2 x i32]* %vet1, i32 0, i64 %8
%10 = load i32* %9, align 4
```

(a)

```
%7 = load i32* %i, align 4
%9 = add nsw i32 0, %7
          LOAD MAU_0 val_vet1;
                     %9;
```

(b)

```
ADD i9;
          0
          i;
LOAD MAU_0 val_vet1;
          i9;
```

(c)

Fonte: AUTORIA PROPRIA (2016)

Na Figura 42 (a) é ilustrado um exemplo de código em C contendo uma operação com a matriz ‘mat1’. Nele, os elementos da matriz são atribuídos a variável ‘g’. A Figura 42 (b) apresenta o fragmento de código gerado pela Clang com a declaração e inicialização da matriz ‘mat1’[2][2], cujos elementos são 3, 3, 2 e 2. Na linha 5 é feita a declaração dos elementos de ‘mat1’, o primeiro trecho entre colchetes indica que a matriz possui 2 como o número de linhas e colunas, e seu tipo é inteiro. O segundo e quarto trecho indica a quantidade de elementos na linha e seu tipo. Já no terceiro e quinto trecho, são mostrados os elementos da matriz. Na linha 9, a matriz é declarada. A Figura 42 (c) corresponde a declaração e inicialização de cada elemento da matriz, que assim como no vetor, cada elemento corresponderá a uma variável na seção DATA. Os nomes das variáveis da matriz também inicia com o nome da matriz seguida da posição do elemento. Por exemplo, para a posição cuja linha é 0 e coluna 1, o nome da variável correspondente é ‘mat101’.

Figura 42 - (a) Operação com a matriz mat1 em C. (b) Declaração e inicialização da matriz e (c) correspondente em PDL.

```c
int main() {

  int mat1[2][2]={{3,3}, {2,2}};
  int i, j, g;

  for(i=0; i<2; i++){
     for (j = 0; j<2; j++){
        g = mat1[i][j];
     }
  }
  return 0;
}
```

(a)

```
@main.mat1 = private unnamed_addr constant [2 x [2 x i32]] [[2 x i32] [i32 3, i32 3], [2 x i32] [i32 2, i32 2]], align 16

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
  %mat1 = alloca [2 x [2 x i32]], align 16
```

(b)

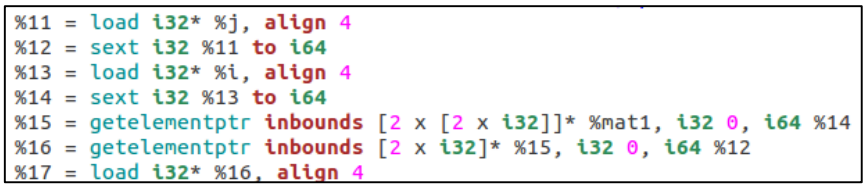
```
DATA
        mat100 = 3
        mat101 = 3
        mat110 = 2
        mat111 = 2
```

(c)

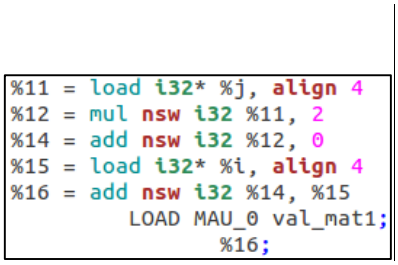
Fonte: AUTORIA PROPRIA (2016)

Para matrizes, o código gerado pela Clang da leitura de um elemento na matriz ‘mat1’ é ilustrado na Figura 43 (a). Nesse código, as variáveis ‘j’ e ‘i’ são carregadas e seus valores estendidos para o inteiro de 64 bits. As duas instruções ‘*getelementptr*’ realizam o retorno do endereço do elemento, cuja posição é o valor das variáveis ‘i’ e ‘j’ correntes. E esse endereço é armazenado em ‘%16’, que é lido em seguida. Esse trecho de código foi adequado a fórmula: endereço_do_elemento [i][j] = j*m + endereço_base + i, onde ‘j’ corresponde ao valor da coluna corrente, ‘m’ ao valor total das colunas da matriz, ‘endereço_base’ a posição do primeiro elemento da matriz e ‘i’ ao valor da linha corrente. Em (b) é apresentado esse trecho de código modificado. Nele, o valor da coluna corrente identificado pela variável ‘j’ é lida, em seguida é realizada a multiplicação de ‘j’ com o valor 2, que corresponde ao número total de colunas da matriz. O resultado dessa multiplicação é adicionado a 0, que compreende a primeira posição da matriz, e armazenado na variável ‘%14’. Logo em seguida, a linha corrente é lida, indicada pela variável ‘i’, é carregada e seu valor é somado com o conteúdo da variável ‘%14’ e armazenado em ‘%16’. Por fim, é inserida a instrução *LOAD* para realizar a leitura do conteúdo do endereço guardado em ‘%16’ e seu elemento será carregado para a variável ‘val_mat1’. Em (c) apresenta-se o código PDL correspondente ao código intermediário de (b).

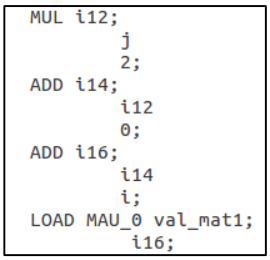
Figura 43 - Leitura de um elemento da matriz mat1: (a) gerado pela Clang, (b) código intermediário modificado e (c) em PDL



(a)



(b)



(c)

Fonte: AUTORIA PROPRIA (2016)

3.7 Considerações

O compilador C2PDL possibilita ao usuário a tradução de códigos escritos na linguagem de programação C para a linguagem PDL. Esse compilador disponibiliza três níveis de otimização para serem aplicados aos programas. Sendo constituídos por 5 otimizações ao todo, além da possibilidade da geração de código contendo matrizes e vetores. Os códigos gerados são melhores que os códigos produzidos por COMPILADOR1, uma vez que C2PDL considera características da Arquitetura IPNoSys. Como pontos fracos esse compilador não realiza a geração de código que contenham matrizes com mais de duas dimensões, macros e funções.

4 RESULTADOS DE VALIDAÇÃO

Para validação do compilador/otimizador C2PDL, foram utilizadas cinco aplicações reais já utilizadas como *benchmark* da arquitetura, porém implementadas diretamente em PDL, ou seja, sem a utilização de compilador. O que as cinco aplicações possuem em comum são laços de repetição em seus códigos. Essa característica foi escolhida em virtude de que o nível de otimização O2 atua somente sobre eles. As aplicações são: contador acumulador, multiplicação de matrizes, soma de matrizes, RLE e DCT-2D. Para cada aplicação foram gerados códigos nos três níveis de otimização oferecidos pelo C2PDL. Os critérios de desempenho designados para analisar o impacto das otimizações propostas neste trabalho foram: tempo de execução da aplicação no simulador e tamanho do código da aplicação. Será apresentada também a quantidade de diminuição das instruções de *LOAD* e *STORE* em cada aplicação comparando os níveis de otimização O e O1. Em seguida, são comparados os códigos das aplicações gerados pelo C2PDL com os códigos desenvolvidos diretamente em PDL utilizando os dois critérios de desempenho. Por fim, ainda visando a validação desta ferramenta para aplicações que não utilizam laços de repetição e o impacto da aplicação das otimizações de eliminação de instruções *COPY* e *STORE* desnecessárias foram utilizadas mais três aplicações, são elas: acumulador, média aritmética e média ponderada.

A primeira aplicação consiste em um contador acumulador. Nela, o valor da variável acumulador é acrescido de um (1) a cada iteração do laço, que apresenta 256 iterações, totalizando 256 somas. A segunda aplicação é a multiplicação de duas matrizes quadradas de dimensão 20x20. A terceira aplicação escolhida consiste na soma de duas matrizes de dimensão 20x20. A penúltima aplicação é o algoritmo da DCT-2D para 192 blocos. Cada bloco corresponde a uma matriz quadrada de dimensão 8 x 8. E por fim, a última aplicação consiste no algoritmo de descompressão de dados RLE em um vetor composto por 256 elementos.

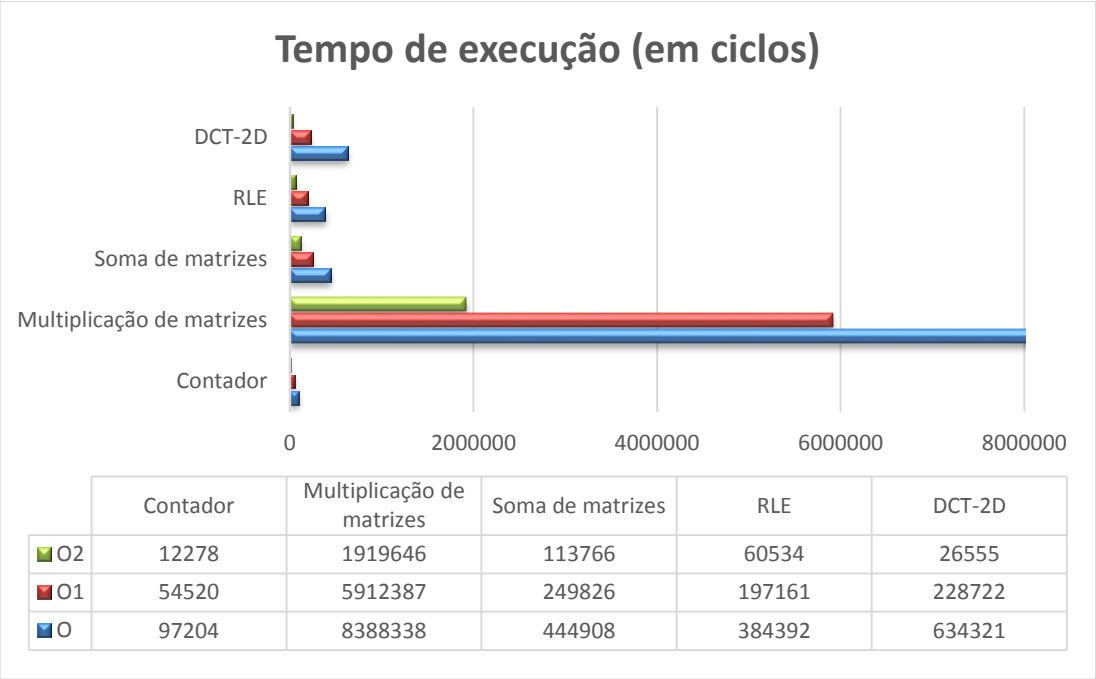
Os códigos das aplicações em linguagem de programação C foram inseridos como entrada no compilador C2PDL que produziu programas na linguagem PDL da IPNoSys. Foram gerados programas PDL para os três níveis de otimização. Em seguida, os programas PDL foram montados no *assembler* padrão de IPNoSys e executados em seu simulador. O simulador calculou o tempo de execução em ciclos e o tamanho do código em *bytes* de cada programa bem como a quantidade de cada instrução executada.

Durante a execução do nível O2, os laços de repetição das aplicações foram divididos em 4 fluxos de execução. Na aplicação contador acumulador, cada fluxo contém 64 iterações.

Na soma de matrizes, cada laço de repetição foi dividido em 5 iterações. Na multiplicação de matrizes, cada laço também foi dividido em 5 iterações. Na RLE cada fluxo contém 64 iterações. E na DCT-2D, cada fluxo contém 2 iterações.

O gráfico da Figura 44 ilustra o tempo de execução de cada aplicação nos três níveis de otimização. Ao analisar esse gráfico, observa-se um comportamento semelhante nas barras que representam cada nível. Nota-se que para todas as aplicações o nível O gastou o maior tempo de execução, enquanto que o nível O2 necessitou de um tempo menor.

Figura 44 – Cinco aplicações em relação ao tempo de execução



	Contador	Multiplicação de matrizes	Soma de matrizes	RLE	DCT-2D
O2	12278	1919646	113766	60534	26555
O1	54520	5912387	249826	197161	228722
O	97204	8388338	444908	384392	634321

Fonte: AUTORIA PROPRIA (2016)

O nível O não apresenta otimizações. O código para esse nível é gerado linha a linha do código intermediário criado pela Clang, dessa forma apresenta muitas instruções de *LOAD* e *STORE*. E como apresentado no Capítulo 2, essas instruções necessitam de um maior tempo de espera, especialmente a instrução *LOAD* que precisa de um tempo de espera dobrado. Com isso, um código que contenha essas instruções sempre apresentará um tempo de execução elevado comparado a um código que apresente poucas ou nenhuma dessas instruções.

Já o nível O2 apresenta o menor tempo de execução dos três. Isso acontece porque esse nível beneficia-se tanto das otimizações do nível O1 (especialmente da eliminação de instruções *LOAD*s e *STORE*s desnecessárias) quanto do paralelismo da arquitetura. Visto que nesse nível, o código é dividido em quatro fluxos, fragmentando a execução do laço de repetição. Esses

quatro fluxos são executados simultaneamente e cada um é responsável pela execução de uma parte do laço.

Nesse gráfico, nota-se a influência da ausência e/ou diminuição de instruções *LOAD* e *STORE* nos códigos das aplicações para a arquitetura IPNoSys. Justificando assim que os códigos em PDL devem conter apenas as instruções de *LOAD* e *STORE* indispensáveis, ou seja, instruções que não podem ser substituídas por outras que não são executadas na MAU.

Ainda nesse gráfico observa-se que há um grande ganho de desempenho em tempo de execução ao realizar a paralelização do código. Esse ganho é diretamente proporcional ao número de iterações do laço de repetição. Assim, quanto maior o número de iterações do laço maior também será o ganho do desempenho, pois há uma diminuição do tempo de execução que a aplicação necessita. Não há um ganho de desempenho padrão para todas as aplicações. Dentre as aplicações analisadas, a DCT-2D possui o maior ganho de desempenho por apresentar um número maior de instruções e iterações do laço de repetição. No seu pior caso, nível O, é gasto 23 vezes mais tempo que no nível O2. E no caso médio, nível O1, é gasto 8 vezes mais que o nível O2. Dessa forma, para o critério tempo de execução o melhor caso é o nível O2.

A Tabela 1 apresenta a quantidade de instruções de carregamento e armazenamento contidas no código de cada aplicação nos níveis O e O1. Os valores para o nível O2 foram omitidos, pois deseja-se identificar a quantidade de instruções eliminadas com a otimização de *LOAD* e *STORE*.

Tabela 1 - Quantidade de instruções *LOADs* e *STOREs* nos níveis de otimização O e O1.

Aplicação	Nível	Quantidade de *LOADS*	Quantidade de *STORES*
Contador acumulador	O	2	2
	O1	0	1
Multiplicação de matrizes	O	11	7
	O1	2*	2
Soma de matrizes	O	12	4
	O1	2*	1
RLE	O	15	7
	O1	2*	2
DCT-2D	O	88	39
	O1	17*	8

Fonte: AUTORIA PROPRIA (2016)

Observando a Tabela 1, verifica-se que ocorreu uma diminuição tanto da quantidade de instruções de carregamento quanto de armazenamento comparando os valores dos níveis O e O1. Percebe-se que a diminuição na quantidade de instruções *LOAD* corresponde a pelo menos 5 vezes a quantidade gerado no nível O. E que para as instruções *STORE* corresponde a pelo

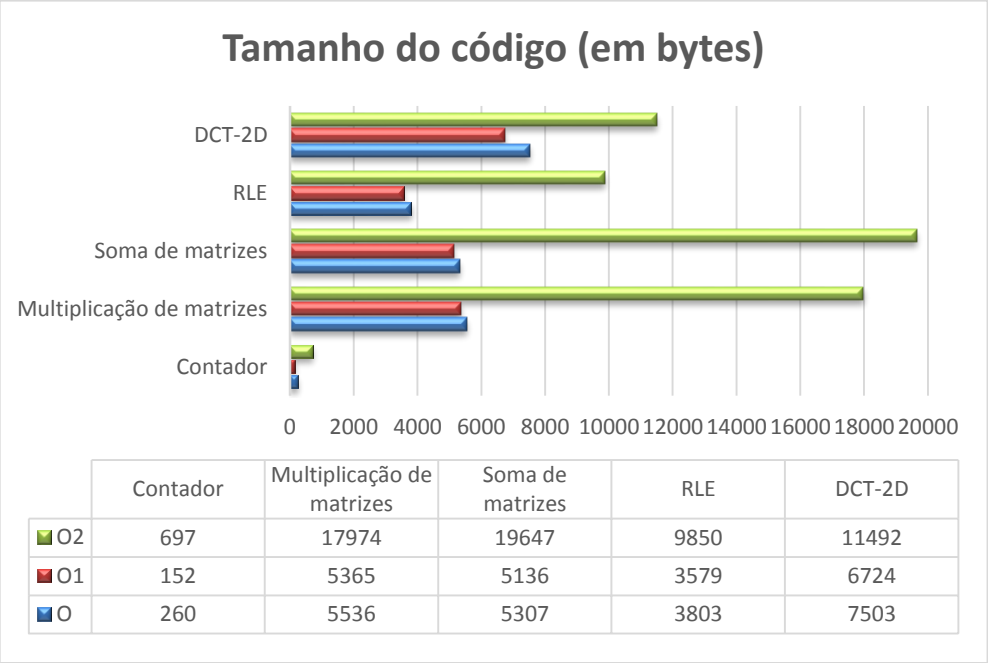
menos a metade. O símbolo asterisco (*) presente na coluna “Quantidade de *LOADS*” indica que essas aplicações realizam operações com matrizes e/ou vetores, o qual esse é o único caso em que instruções *LOAD* não podem ser substituídas por outras. Caso exista a possibilidade da substituição, a instrução *LOAD* é totalmente eliminada como acontece com a aplicação do contador acumulador.

A diminuição da quantidade de instruções de *STORE* acontece, porque o nível O1 gera somente uma instrução de armazenamento para uma determinada variável, enquanto o nível O gera a quantidade de vezes que aparece no código intermediário.

Essas duas otimizações afetam diretamente o desempenho da aplicação em relação ao tempo de execução.

Na Figura 45, é apresentado o gráfico do tamanho do código em cada aplicação nos três níveis de otimização. Percebe-se que para todas as aplicações o nível O2 necessitou de um tamanho maior enquanto que o nível O1 um menor tamanho comparando os três níveis.

Figura 45 - Cinco aplicações em relação ao tamanho do código



	Contador	Multiplicação de matrizes	Soma de matrizes	RLE	DCT-2D
O2	697	17974	19647	9850	11492
O1	152	5365	5136	3579	6724
O	260	5536	5307	3803	7503

Fonte: AUTORIA PROPRIA (2016)

Os códigos gerados pelo nível O2 apresentam quatro cópias do código produzido pelo nível O1 com o acréscimo de um pacote, uma instrução SYNEXEC e quatro instruções SYNC, como apresentado no Capítulo 2. Dessa forma, os códigos do nível O2 podem ocupar um tamanho até quatro vezes maior do que os códigos gerados pelo nível O1. Assim como para o tempo de execução, não há um percentual igual do aumento ou diminuição do tamanho do

código gerado para as aplicações. Dentre as aplicações testadas, a aplicação contador acumulador apresentou o maior aumento no tamanho do código produzido no nível O2. Esse tamanho foi 4 vezes maior que o código no nível O1. E o dobro do tamanho em relação ao código gerado pelo nível O.

O menor tamanho do código dos três níveis é requerido em O1. Isso se dar pelo fato de que no nível O sempre que uma variável é lida, é inserida uma instrução *LOAD* para ela. Já no nível O1, para realizar a leitura de uma variável, uma instrução de cópia para dois temporários é gerada, diminuindo assim o número de instruções de carregamento e economizando em tamanho do código.

E ainda no nível O, sempre que uma instrução de armazenamento aparecer no código intermediário, será gerada uma instrução *STORE*. Dessa forma, se houver, por exemplo, dois armazenamentos para uma determinada variável e o seu valor não for lido em uma outra instrução antes do segundo armazenamento, serão geradas duas instruções de armazenamento para essa variável. Assim, nesse nível, o número de instruções *STORE* geradas é igual ao número de instruções de armazenamento que aparece no código intermediário. Já no nível O1, as instruções de *STORE* são geradas ao final do código e apenas uma vez para cada variável modificada no programa.

Após as análises dos gráficos, percebe-se que não existe um percentual padrão de melhoria de um nível de otimização para o outro, mas é notável que o nível O2 apresenta uma melhoria em relação ao tempo de execução da aplicação e corresponde ao pior caso em relação ao tamanho do código. Já no nível O1 há uma diminuição no tamanho do código da aplicação.

Os valores gerados para cada aplicação variam pelas instruções utilizadas e pelo número de iterações do laço de repetição. Em códigos que apresentam um menor número de instruções que são executadas pelas MAUs, seu tempo de execução será menor que em um código com uma quantidade maior dessas instruções. Por fim, quanto maior o número de iterações do laço, maior também será o tempo exigido para sua execução. Uma vez que, ao replicar o código, dividir as iterações de cada laço e executar em uma MAU diferente seu tempo diminui significativamente.

Além da geração automática de código otimizado, a ferramenta desenvolvida contribui com uma melhora no desempenho tanto no tempo de execução quanto no tamanho do código gerado. O usuário é o responsável pela escolha de qual nível de otimização utilizar na aplicação.

A Tabela 2 apresenta o tempo de compilação da geração de código de cada aplicação nos três níveis de otimização. Nota-se que a geração de código no nível O1 gasta aproximadamente o dobro de tempo que a geração no nível O. Isso acontece porque no nível

O1, além do tempo gasto na geração de código desse nível, é acrescido o tempo para gerar o código no nível O. Ainda observa-se que a geração no nível O2 gasta em torno de 1,05 de tempo a mais que o nível O1, visto que esse tempo a mais corresponde a replicação do código gerado pelo nível O1 em quatro fluxos de execução.

Tabela 2 - Tempo de compilação das cinco aplicações nos três níveis de otimização

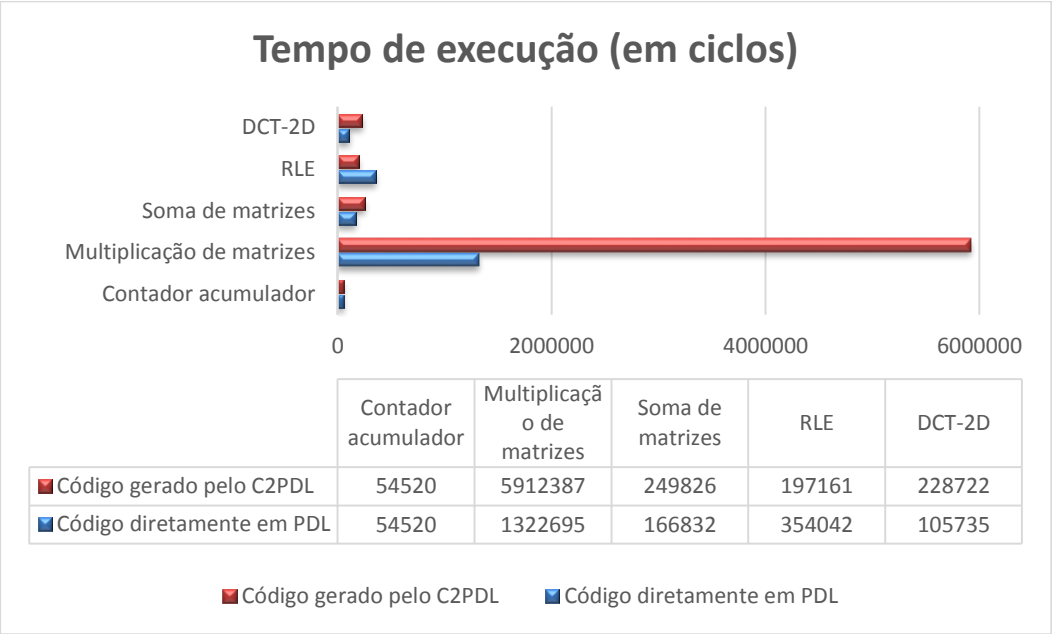
Aplicação	Nível	Tempo de compilação
Contador acumulador	O	0,006 s
	O1	0,011 s
	O2	0,021 s
Multiplicação de matrizes	O	0,104 s
	O1	0,175 s
	O2	0,186 s
Soma de matrizes	O	0,091 s
	O1	0,179 s
	O2	0,193 s
RLE	O	0,051 s
	O1	0,099 s
	O2	0,103 s
DCT-2D	O	0,163 s
	O1	0,321 s
	O2	0,343 s

Fonte: AUTORIA PROPRIA (2016)

Os códigos das cinco aplicações geradas pelo nível O1 do C2PDL foram comparados aos códigos gerados diretamente em PDL utilizando também os critérios: tempo de execução e tamanho do código.

Para o tempo de execução, o gráfico da Figura 46 foi gerado. Observa-se que para esse critério, as aplicações que apresentam operações com matrizes em seus códigos (multiplicação de matrizes, soma de matrizes e DCT-2D) obtiveram tempo de execução maior nos códigos gerados pelo C2PDL comparados aos seus códigos implementados diretamente em PDL. Isso acontece porque suas implementações foram realizadas de maneiras distintas.

Figura 46 – Tempo de execução de código PDL e gerado pelo C2PDL



Fonte: AUTORIA PROPRIA (2016)

Nos códigos desenvolvidos diretamente em PDL é realizado o cálculo da posição do primeiro elemento da matriz. E para o cálculo das posições seguintes, é incrementada a posição do elemento anterior, aproveitando assim o cálculo inicial. Em seguida, são realizadas as leituras de cada posição retornando os respectivos elementos para que sejam realizadas as operações sobre eles. E dependendo da aplicação, todos os elementos são enviados para o próximo pacote ou são calculadas as novas posições para serem armazenados. Como exemplo, observe a Figura 47, em (a) nas linhas 54 a 58, é apresentado o cálculo do endereço para a leitura do primeiro elemento da matriz A e armazenado nas variáveis ‘ind_A1’ e ‘ind_A2’. ‘ind_A1’ é utilizada para carregar o conteúdo dessa posição para a variável ‘val_A’. Nas linhas 60 e 61, é realizada a adição da variável ‘ind_A2’ com o valor 4, que corresponde ao deslocamento da matriz B, aproveitando assim o cálculo realizado anteriormente para a matriz A. O mesmo ocorre para a matriz C como se pode observar nas linhas 66 e 67. Em (b), é aproveitado o cálculo da primeira posição de determinada matriz e para as posições seguintes é incrementado o valor 1, como pode-se observar nas linhas 99 a 101, 104 a 106 e 109 a 111. Em seguida, são realizadas as leituras (linhas 102 e 103, 107 e 109 e 112 e 113) e envio dessas posições (linhas 114 a 125) para o pacote ‘pac_4’.

Figura 47 – Códigos PDL contendo: (a) cálculo do primeiro elemento das matrizes e (b) leitura e envio dos elementos

```
40        PACKAGE laco_coluna
41        ADDRESS MAU_0
42                COPY l1; //linha atual (do laco anterior)
43                        linha = 0;
44                COPY c1  c2; //coluna atual (controlador laco)
45                        coluna = 0;
46                COPY c3  c4; //coluna atual
47                        c1;
48                ADD cont1; //incrementa controlador
49                        c2    1;
50                BE fim_laco;
51                        c3   2;  //qtd de colunas
52                SEND MAU_0 coluna;      //atualiza proprio controlador
53                        prog, laco coluna    cont1;
54                MUL l2;
55                        l1  2; //2 quantidade de elementos por linha
56                ADD ind_A1  ind_A2;       //indice de A = l*2 + c
57                        l2   c4;
58                LOAD MAU_0   val_A;
59                        ind_A1;  //carrega o valor A[l][i]
60                ADD ind_B1  ind_B2;  //indice de B
61                        ind_A2   4;  //4 = quantidade de elementos de A
62                LOAD MAU_0   val_B;
63                        ind_B1;  //carrega o valor B[l][i]
64                ADD val_C;
65                        val_A  val_B;
66                ADD ind_C;  //indice de C
67                        ind_B2   4;  //4 = quantidade de elementos de B
68                STORE MAU_0;
69                        val_C  ind_C;
70                EXEC MAU_0;
71                        prog, laco_coluna;
72                JUMP prox_itera;
73        fim_laco:       EXEC MAU_0; // volta para o laco mais externo (linha)
```

(a)

```
 95        COPY t3 t7;
 96                t2; //endereco inicial
 97        LOAD MAU_1 r0;
 98                t3;
 99        ADD t4 t8;
100                t7
101                1;
102        LOAD MAU_1 r1;
103                t4;
104        ADD t5 t9;
105                t8
106                1;
107        LOAD MAU_1 r2;
108                t5;
109        ADD t6;
110                t9
111                1;
112        LOAD MAU_1 r3;
113                t6;
114        SEND MAU_1 a_0;
115                prog_0, pac_4
116                r0;
117        SEND MAU_1 a_1;
118                prog_0, pac_4
119                r1;
120        SEND MAU_1 a_2;
121                prog_0, pac_4
122                r2;
123        SEND MAU_1 a_3;
124                prog_0, pac_4
125                r3;
126        EXEC MAU_1;
127                prog_0, pac_3;
```

(b)

Fonte: FERNANDES; SILVA (2016)

Já em C2PDL, para cada leitura da posição de um elemento é utilizada a fórmula mostrada no Capítulo 2, e isso ocorre a cada iteração do laço, ou seja, somente é realizada a leitura de um elemento de uma determinada matriz a cada iteração, aumentando assim o tempo de execução da aplicação. A Figura 48 ilustra um trecho de código gerado por C2PDL para a

leitura de elementos das matrizes ‘mat1’ (linhas 48 a 60) e ‘mat2’ (linhas 63 a 75). Observe que para cada matriz é realizado um cálculo diferente, não sendo aproveitado o cálculo da primeira matriz como ocorre nos códigos gerados diretamente em PDL. Dentre essas três aplicações, a multiplicação de matrizes apresentou um tempo 4 vezes maior, uma vez que essa aplicação é composta por três laços de repetição com 20 iterações cada e são calculadas 400 posições de cada matriz. Já a aplicação contador acumulador, obteve o mesmo tempo de execução nos dois códigos, uma vez que os códigos gerados são semelhantes. Por fim, a aplicação RLE foi a única das cinco que apresentou um tempo de execução menor no código gerado pelo C2PDL, houve uma redução de 44,3% no tempo. Isso aconteceu porque sua implementação gerou somente dois pacotes enquanto que no seu código diretamente em PDL foram gerados cinco pacotes e a injeção e execução de cada pacote consome um determinado tempo.

Figura 48 – Trecho de código gerado por C2PDL para a leitura de um elemento das matrizes

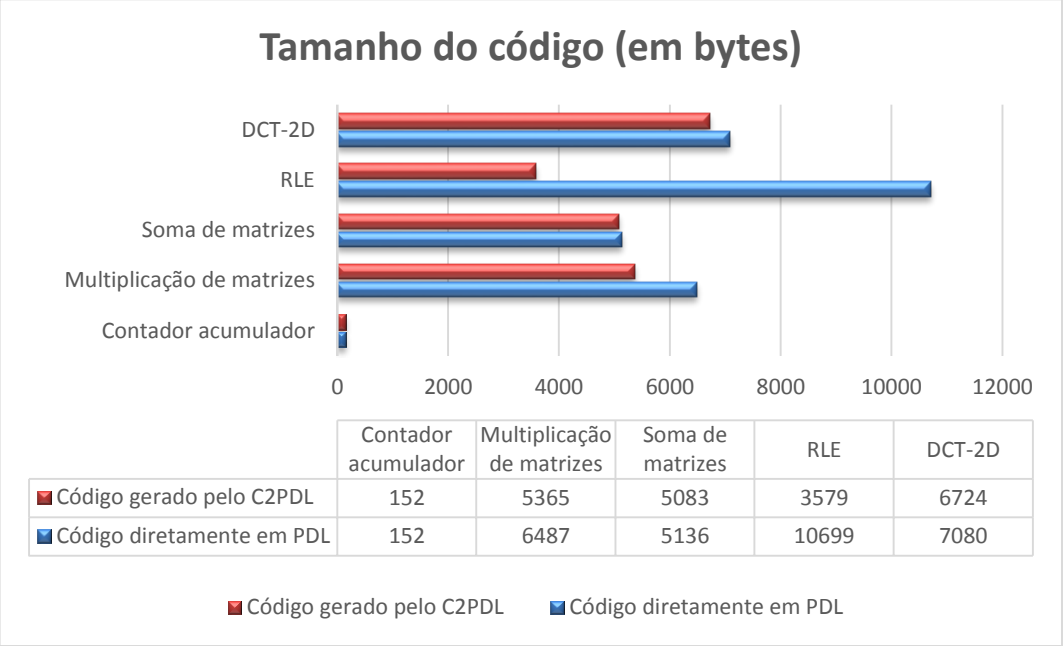
```
37 PACKAGE loop1
38 ADDRESS MAU_0
39         COPY c2 c1;
40                 j = 0;
41         COPY c4 c3;
42                 i = 0;
43         BGE label37;
44                 c1
45                 2;
46         COPY c6 c5;
47                 c2;
48         MUL i14;
49                 c5
50                 2;
51         ADD i16;
52                 i14
53                 0;
54         COPY c8 c7;
55                 c4;
56         ADD i18;
57                 i16
58                 c7;
59         LOAD MAU_0 val_mat1;
60                 i18;
```

```
63         MUL i21;
64                 c9
65                 2;
66         ADD i23;
67                 i21
68                 4;
69         COPY c12 c11;
70                 c8;
71         ADD i25;
72                 i23
73                 c11;
74         LOAD MAU_0 val_mat2;
75                 i25;
```

Fonte: AUTORIA PROPRIA (2016)

A avaliação do critério tamanho do código é ilustrada na Figura 49. Nela, observa-se que, como os códigos gerados para a aplicação contador acumulador são semelhantes, o tamanho do código também corresponde ao mesmo. Já os códigos das demais aplicações gerados pelo C2PDL apresentaram um tamanho menor comparado aos seus códigos implementados diretamente em PDL. Nas aplicações que realizam operações com matrizes foi diminuído o número de instruções *SEND* geradas, os incrementos de uma determinada posição da matriz e a quantidade de instruções *LOAD* e *STORE* geradas.

Figura 49 – Tamanho do código de código PDL e gerado pelo C2PDL



	Contador acumulador	Multiplicação de matrizes	Soma de matrizes	RLE	DCT-2D
Código gerado pelo C2PDL	152	5365	5083	3579	6724
Código diretamente em PDL	152	6487	5136	10699	7080

Fonte: AUTORIA PROPRIA (2016)

Apesar de C2PDL gerar códigos contendo operações com matrizes com um tempo de execução maior que seus correspondentes implementados diretamente em PDL, os códigos produzidos pelo C2PDL apresentam um tamanho de código menor. A utilização desta ferramenta para a geração de código contribui para a diminuição do esforço de implementação das aplicações.

Para essas cinco aplicações, não é possível a geração de código por COMPILADOR1, uma vez que esse compilador possui restrições e não gera código para laços de repetição e nem para matrizes ou vetores. Por esse motivo seus dados não foram utilizados na comparação.

Os códigos gerados pelo nível O de C2PDL são diferentes dos códigos produzidos por COMPILADOR1. Uma vez que COMPILADOR1 já gera código contendo alguma otimização, enquanto que o nível O não possui, consiste no código produzido por um desenvolvedor que não conhece a arquitetura IPNoSys e sua característica em relação as instruções *LOAD* e *STORE*.

Contribuindo também para a validação do compilador C2PDL, mais três aplicações foram utilizadas, uma vez que somente programas contendo laços de repetição haviam sido testados. Essas aplicações não fazem uso de laços de repetição. A primeira aplicação consiste em um acumulador, nela são realizadas 256 adições do valor 1 a variável ‘a’, em seguida seu resultado é armazenado na variável ‘x’. A segunda aplicação corresponde a uma média aritmética de 30 elementos, cujo resultado é inserido na variável ‘a’. Por fim, a última aplicação

consiste na média ponderada da nota de 10 alunos, calculada a partir da seguinte fórmula: media_ponderada = ((n1 * 2) + (n2 * 3) + (n3 * 4))/9, onde ‘n1’, ‘n2’ e ‘n3’ correspondem as notas de determinado aluno.

Objetivando o conhecimento da quantidade de instruções *COPY* e instruções *STORE* que foram eliminadas comparando o código gerado por COMPILADOR1 e por C2PDL no nível de otimização O1, essas três aplicações foram inseridas nos compiladores e a partir de suas execuções foi gerada a Tabela 3. Observe que para todas as aplicações a quantidade de instruções *COPY* foi diminuída em 100% nos códigos gerados pelo compilador otimizante. E também houve uma diminuição na quantidade de instruções *STORE* de 67%, 97% e 76%, respectivamente, para as aplicações acumulador, média aritmética e média ponderada.

Tabela 3 – Quantidade de instruções nos códigos gerados por COMPILADOR1 e por C2PDL

Aplicação	Compilador	Quantidade de *COPY*	Quantidade de *STORE*
Acumulador	COMPILADOR1	5	3
	C2PDL	0	1
Média Aritmética	COMPILADOR1	63	32
	C2PDL	0	1
Média Ponderada	COMPILADOR1	81	41
	C2PDL	0	10

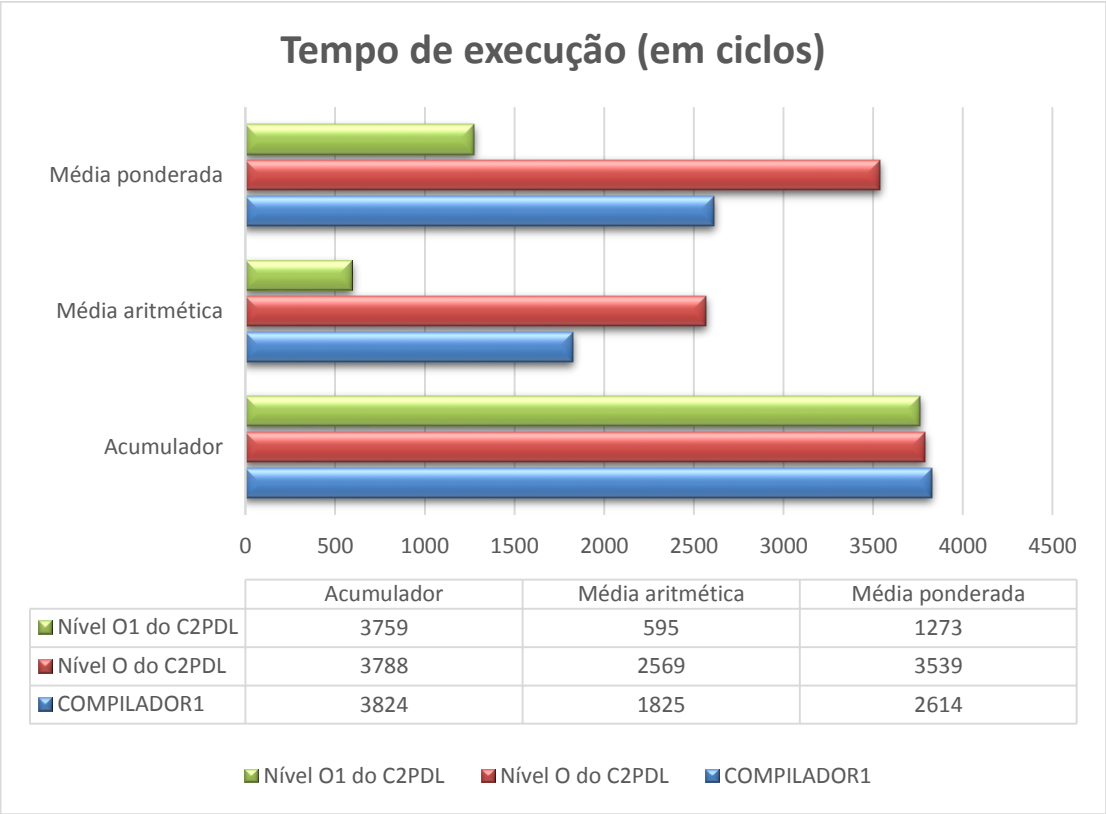
Fonte: AUTORIA PROPRIA (2016)

Foram gerados também códigos no nível de otimização O desta ferramenta para serem comparados com os códigos gerados pelo compilador COMPILADOR1 e pelo compilador otimizante no nível O1. Os códigos PDL dessas aplicações também foram inseridos no montador e simulados na IPNoSys para a identificação e comparação dos critérios de desempenho: tempo de execução e tamanho do código.

Na Figura 50, ilustra-se essa comparação em relação ao tempo de execução das três aplicações. Observa-se que para todas as aplicações, o menor tempo de execução pertence aos códigos produzidos por esta ferramenta no nível de otimização O1. Isso acontece, pois nesses códigos há a ausência de instruções de carregamento e somente são utilizadas as instruções de armazenamento necessárias, contribuindo para a diminuição do tempo de execução gasto pelos códigos. Já, para as aplicações média aritmética e média ponderada, um maior tempo de execução foi gasto pelo nível O desta ferramenta, uma vez que nesses códigos há a utilização de instruções *LOAD* e *STORE*. Enquanto que para a aplicação acumulador, o maior tempo foi gasto pelo compilador COMPILADOR1. Isso ocorreu porque esse compilador gera instruções de armazenamento para todas as variáveis declaradas em seu código, porém nem todas as

variáveis são modificadas no decorrer do código, sendo inseridas instruções *STORE* desnecessárias.

Figura 50 – Três aplicações em relação ao tempo de execução



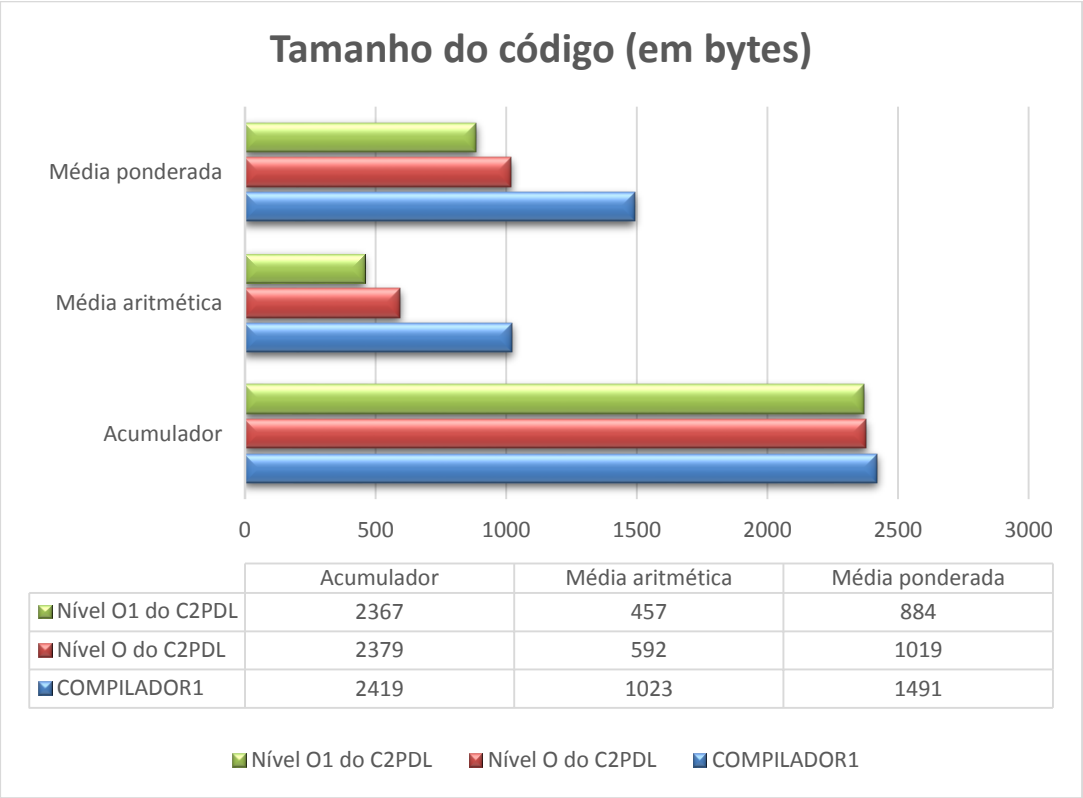
	Acumulador	Média aritmética	Média ponderada
Nível O1 do C2PDL	3759	595	1273
Nível O do C2PDL	3788	2569	3539
COMPILADOR1	3824	1825	2614

Fonte: AUTORIA PROPRIA (2016)

A porcentagem de diminuição do tempo de execução dos códigos produzidos por C2PDL no nível de otimização O1 em relação aos códigos gerados por COMPILADOR1 foi de 2%, 67% e 51%, para as aplicações acumulador, média aritmética e média ponderada, respectivamente. Essa diminuição ocorreu devido à redução da quantidade de instruções *COPY* e especialmente instruções *STORE* utilizadas.

O gráfico da Figura 51 ilustra a comparação dos códigos produzidos por esta ferramenta nos níveis O e O1 e por COMPILADOR1 em relação ao tamanho do código. Como pode-se observar, em todas as aplicações COMPILADOR1 gerou um código maior, devido a quantidade de instruções *COPY* e *STORE* desnecessárias que são inseridas. Já um código menor foi produzido pelo nível O1 desta ferramenta. Que para essas aplicações eliminou todas as instruções *COPY* inseridas em COMPILADOR1 e instruções de carregamento desnecessárias.

Figura 51 - Três aplicações em relação ao tamanho do código



	Acumulador	Média aritmética	Média ponderada
Nível O1 do C2PDL	2367	457	884
Nível O do C2PDL	2379	592	1019
COMPILADOR1	2419	1023	1491

Fonte: AUTORIA PROPRIA (2016)

A diminuição no tamanho do código produzido pelo nível O1 desta ferramenta em relação a COMPILADOR1 foi de 2%, 55% e 41% para as aplicações acumulador, média aritmética e média ponderada, respectivamente. Isso se dá pela redução das instruções *COPY* e *STORE*.

Ao analisar os dados gerados e os gráficos produzidos, percebe-se que o compilador C2PDL contribuiu com uma melhor qualidade nos códigos gerados em relação aos critérios tempo de execução e tamanho do código comparada aos códigos gerados por COMPILADOR1. Uma vez que para todas as aplicações utilizadas, esses dois critérios foram diminuídos, gerando somente instruções indispensáveis ao programa.

5 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Este trabalho foi desenvolvido devido à necessidade da geração automática de código para a arquitetura IPNoSys a partir da linguagem de alto nível C, aumentando a produtividade do programador. A ferramenta desenvolvida C2PDL considera as características da arquitetura como a cautela no uso das instruções de *LOAD* e *STORE* e a exploração de paralelismo em laços de repetição.

Dessa forma, este trabalho propôs a criação da etapa de otimização na ferramenta COMPILADOR1 já existente. Esse compilador utiliza a ferramenta Clang para a geração de seu *front-end*. A ferramenta proposta beneficia-se da geração de código PDL e melhora o código gerado por COMPILADOR1 .

A implementação dessa etapa de otimização foi realizada na linguagem de programação C++, sem o auxílio de ferramentas automáticas. Ferramentas não foram utilizadas, pois este trabalho beneficiou-se da implementação de otimizações desenvolvidas na ferramenta COMPILADOR1. Além de que as otimizações escolhidas para essa etapa são específicas para a arquitetura IPNoSys. E também devido a semelhança do código LLVM-IR gerado pela Clang com o código PDL.

A ferramenta proposta apresenta três níveis de otimização (O, O1 e O2). A escolha de qual nível utilizar é realizada pelo usuário do compilador. Essa escolha é feita pela opção de qual critério de desempenho o usuário deseja melhorar em seu código.

Para a validação do C2PDL, foram escolhidas cinco aplicações reais e gerados seus códigos em cada nível de otimização, para que os critérios de desempenho tempo de execução das aplicações no simulador e tamanho da aplicação fossem obtidos e comparados. Para todas as aplicações, os resultados de simulação mostram que o nível O2 gera um código com melhor tempo de execução comparando os três níveis, no entanto suas aplicações requerem um tamanho maior. Já o nível O apresenta o pior tempo de execução dos três níveis. E por fim, o nível O1 necessita de um tamanho menor para seus códigos. Foi comparado ainda a quantidade de instruções *LOAD* e *STORE* geradas nos níveis O e O1, percebe-se que nos códigos gerados pelo nível O1 há uma diminuição na quantidade dessas instruções comparados aos códigos no nível O. E ainda foi realizada uma comparação entre os códigos gerados pelo C2PDL e os códigos desenvolvidos diretamente em PDL dessas cinco aplicações. Notou-se uma diminuição no tamanho do código gerado por esta ferramenta. Já em relação ao tempo de execução, as aplicações que realizavam operações com matrizes gastaram um tempo maior.

Por fim, visando validar o C2PDL para códigos sem a utilização de laços de repetição, foram utilizadas mais três aplicações e seus dados confrontados com os dados de seus códigos gerados por COMPILADOR1. Percebeu-se que esta ferramenta gerou tanto um código com o tamanho menor quanto um menor tempo de execução.

Os resultados obtidos deixam evidentes o ganho tanto em relação ao tempo de execução quanto ao tamanho do código gerados com a aplicação dos níveis de otimização disponibilizados por esta ferramenta aos códigos inseridos como entrada. Além da melhora no código gerado em COMPILADOR1.

Este trabalho de dissertação gerou um artigo que foi submetido ao periódico IEEE *Latin America* e aguarda resultado.

Os trabalhos futuros incluem a substituição da manipulação de arquivos por estruturas de dados, visando a diminuição do tempo de compilação para a geração de código desta ferramenta. A geração de códigos PDL para programas contendo matrizes com mais de duas dimensões, macros, chamadas de funções e a posterior implementação da otimização *inlining*. Essa otimização consiste na substituição do código da função pela sua chamada, minimizando assim o tempo gasto na chamada. A utilização da otimização *inlining* acarretará em uma melhora no desempenho de códigos que contenham funções, uma vez que no modelo atual há uma paralisação do pacote à espera do resultado da função.

REFERÊNCIAS

ABDERAZEK, B. A.; ARSENJI, M.; SHIGETA, S.; YOSHINAGA, T.; SOWA, M. Queue Processor Architecture for Novel Queue Computing Paradigm Based on Produced Order Scheme. In: *Proceedings of the High Performance Computing and Grid in Asia Pacific Region, Seventh International Conference on (HPCAsia'04)* - Volume 00, 2004. IEEE Computer Society, 2004.

ABDERAZEK, B. A.; YOSHINAGA, T.; SOWA, M. Scalable Core-Based Methodology and Synthesizable Core for Systematic Design Environment in Multicore SoC (MCSoC). In: *International Conference on Parallel Processing Workshops* (ICPPW'06), 2006. ACCELLERA, Systems Initiative. SystemC. Disponível em: <http://www.accellera.org/>. Acesso em: 25 jun. 2016.

ADAMATZKY, A. **Unconventional computing**. International Journal of General Systems, 2014. Vol. 43, No. 7, 671–672.

AHO, A. V., SETHI, R.; ULLMAN, J. D. **Compiladores: Princípios, técnicas e ferramentas**. São Paulo: Pearson, 2007. 344 p.

ARAUJO, S. R. F. **Estudo da Viabilidade do Desenvolvimento De Sistemas Integrados Baseados em Redes em Chip Sem Processadores: Sistema IPNoSys.** 2008. Dissertação (Mestrado em Sistemas e Computação) - Programa de Pós-Graduação em Sistemas e Computação, Universidade Federal do Rio Grande do Norte, Natal, 2008. Disponível em: <http://repositorio.ufrn.br:8080/jspui/bitstream/123456789/17969/1/SilvioRFA.pdf>. Acesso em: 16 mai. 2014.

ARAUJO, S. R. F. **Projeto de Sistemas Integrados de Propósito Geral Baseados em Redes em Chip Expandindo as Funcionalidades dos Roteadores para Execução de Operações: A plataforma IPNoSys.** 2012. Tese (Doutorado em Sistemas e Computação) - Programa de Pós-graduação em Sistemas e Computação, Universidade Federal do Rio Grande do Norte, Natal, 2012. Disponível em: <http://repositorio.ufrn.br:8080/jspui/bitstream/123456789/17948/1/SilvioRFA_TESE.pdf>. Acesso em: 23 jun. 2014.

ARAUJO, S.; OLIVEIRA, B. C.; SILVA, I. S. Using NoC routers as processing elements. In: SBCCI2009, *22nd Symposium on Integrated Circuits and Systems Design*, 2009, Natal.

CANEDO, A.; ABDERAZEK, B.; SOWA, M. A GCC-based Compiler for the Queue Register Processor (QRP-GCC). In: *IWMST2006, The 2006 International Workshop on Modern Science and Technology*, 2006, Wuhan. **Anais Eletrônicos...** Wuhan, 2006, p. 250 - 255. Disponível em: <http://web-ext.u-aizu.ac.jp/~benab/publications/conferences/co/canedo06iwmst.pdf>. Acesso em: 24 jun. 2014.

CANEDO, A.; ABDERAZEK, B.; SOWA, M. Queue Register File Optimization Algorithm for QueueCore Processor. In: *19th International Symposium on Computer Architecture and High Performance Computing*, Rio Grande do Sul. **Anais Eletrônicos…** Rio Grande do Sul, 2007, p. 169-176. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4384055&newsearch=true&query

Text=Queue%20Register%20File%20Optimization%20Algorithm%20for%20QueueCore%20Processor>. Acesso em: 10 ago. 2016.

CAVAZOS, J.; O'BOYLE, M. F. P. Method-specific dynamic compilation using logistic regression. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, 2006, New York. **Anais Eletrônicos…** New York, USA: ACM, 2006, p. 229-240. Disponível em: <http://www.anc.ed.ac.uk/machine-learning/colo/oopsla06.pdf >. Acesso em: 23 mar. 2015.

CID, G. S. V. **Uma plataforma de desenvolvimento de software baseada no LLVM para sistemas embarcados com processador RISCO**. 2010. TCC (Bacharelado em Ciência da Computação) - Universidade Federal do Rio Grande do Norte, Natal, 2010. Disponível em: <https://risco-llvm.googlecode.com/files/monografia.pdf>. Acesso em: 25 mai. 2014.

CLANG. **Site oficial do Clang**, 2012. Disponível em: <http://clang.llvm.org/>. Acesso em: 27 mar. 2013.

COOPER, K. D.; SCHIELKE, P. J.; SUBRAMANIAN, D. Optimizing for reduced code space using genetic algorithms. In: *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, New York. **Anais Eletrônicos…** New York, USA: ACM, 1999, p. 1- 9. Disponível em: <www.cs.rice.edu/~devika/papers/lctes99.pdf.gz>. Acesso em: 30 mar. 2015.

COUTO, J. V. **Implementação de um compilador para a arquitetura SIC.** 2013. Monografia (Graduação em Ciência da Computação), Universidade Federal Rural do Semi-Árido, Mossoró, 2013. Acesso em: 08 jun. 2016.

DOWN, K.; SEVERANCE, C. **High performance computing: RISC Architectures, Optimization and Benchmarking**. Sebastopol, CA: O'Reilly, 1998. 466 p.

DUARTE, L. K. **Análise de Métricas de Código Fonte: Além das Métricas de *Design* de Código**. 2014. TCC (Bacharelado em Engenharia de *Software*) - Universidade de Brasília, Brasília, 2014. Disponível em: <http://fga.unb.br/articles/0000/7978/TCC1_Lucas_Kanashiro.pdf>. Acesso em: 28 mai. 2015.

FERNANDES, S. R; SILVA, I. S.; KREUTZ, M. Packet-driven General Purpose Instruction Execution on Communication-based Architectures. **Journal Integrated Circuits and Systems**, v. 5, n.1, p. 53 – 66, abr. 2010. Disponível em: <http://www.sbmicro.org.br/jics/html/artigos/vol5no1/06.pdf>. Acesso em: 13 jun. 2016.

FERNANDES, S. R.; SILVA, I. S. Programação paralela utilizando o modelo IPNoSys. In: *Escola Regional de Informática do Piauí*. **Anais Eletrônicos…** Teresina, PI: ERIPI, p. 145 - 176, 2016. Disponível em: <http://www.eripi.com.br/2017/anais-2016/minicursos>. Acesso em: 02 jul. 2016.

FUJII, S. Y. **Utilização de Técnicas de Otimização de Desempenho em Bioinformática. Estudo de Caso: SRNASCANNER.** 2012. Dissertação (Mestrado em Bioinformática) - Programa de Pós-Graduação em Bioinformática, Universidade Federal do Paraná, Curitiba, 2012. Disponível em: <http://dspace.c3sl.ufpr.br/dspace/bitstream/handle/1884/28955/R%20-

%20D%20-%20SERGIO%20YOSHIMITSU%20FUJII.pdf?sequence=1>. Acesso em: 02 mai. 2015.

FURSIN, G. G.; O'BOYLE, M. F. P.; KNIJNENBURG, P. M. W. Evaluating iterative compilation. In: *Proceedings of the 15th International Conference on Languages and Compilers for Parallel Computing*, Berlin. **Anais Eletrônicos…** Berlin, Heidelberg: Springer-Verlag, 2005, p. 362-376. Disponível em: <http://link.springer.com/chapter/10.1007%2F11596110_24>. Acesso em: 11 mai. 2015.

GADELHA, M. A.; CORREA, E. F.; KREUTZ, M. E. A tool for developing IPNoys programs using a high-level programming language. In: *Workshop on Circuits and System Design*, João Pessoa. **Anais Eletrônicos…** João Pessoa - PB, 2011. Disponível em: <http://www.sbmicro.org.br/index.php?option=content&task=view&id=261&Itemid=102>. Acesso em: 05 jul. 2015.

GCC. GCC, the GNU Compiler Collection. **Site oficial do GCC**, 2015. Disponível em: <http://gcc.gnu.org/>. Acesso em 16 abr. 2015.

GPROF. GNU gprof. **Site oficial do GPROF**, 2016. Disponível em: <https://sourceware.org/binutils/docs-2.20/gprof/index.html#Top>. Acesso em 10 set. 2016.

GOMES, U. M; COUTO, J. V; ARAUJO, S. R. F. Usando Clang para Construção de um Compilador. In: ESCOLA POTIGUAR DE COMPUTAÇÃO E SUAS APLICAÇÕES - EPOCA, Santa Cruz. **Anais Eletrônicos...** Santa Cruz, 2014. Disponível em: <http://www.natal.uern.br/~epoca14/>. Acesso em: 15 out. 2014.

GONÇALVES, C. O., SPOLON, R.; LOBATO, R. S.; MANACERO, A.; LOBATO, D. C. Paralelização Automática de Laços. In: *Information Systems and Technologies (CISTI), 9th Iberian Conference on*, Barcelona. **Anais Eletrônicos...** Barcelona: IEEE, 2014. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6876985&queryText%3DAutomatic+loops+parallelization>. Acesso em: 26 mai 2015.

GOUGH, B. **An Introduction to GCC**. Bristol: Network Theory Limited, 2004. Disponível em: <http://www.tunl.duke.edu/documents/public/root/material/5/An_Introduction_to_GCC-Brian_Gough.pdf>. Acesso em: 03 mai. 2015.

HENNESSY, J., PATTERSON, D. **Arquitetura de Computadores – Uma Abordagem Quantitativa**. 6. ed. Rio de Janeiro: CAMPUS, 2014. 744 p.

HOSTE, K.; EECKHOUT, L. Cole: Compiler optimization level exploration. In: *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, New York. **Anais Eletrônicos…** New York, USA: ACM, 2008, p. 165-174. Disponível em: <http://users.elis.ugent.be/~leeckhou/papers/cgo08.pdf >. Acesso em: 03 abr. 2015.

KREMENEK, T. **Finding software bugs with the clang static analyzer**. In: Apple, 2009. Disponível em: <http://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf>. Acesso em: 28 mai. 2015.

LATTER, C.; ADVE, V. LLVM: A compilation framework for lifelong program analysis and transformation. In: *International Symposium on Code Generation and Optimization – Cgo*, California. **Anais Eletrônicos…** California, 2014. Disponível em: <http://dl.acm.org/citation.cfm?id=977673>. Acesso em: 08 mar. 2015.

LEE. S.; JOHNSON, T. A.; EIGENMANN, R. Cetus An Extensible Compiler Infrastructure for Source-to-Source Transformation. In: *Languages and Compilers for Parallel Computing - LCPC*, Texas **Anais Eletrônicos...** Texas, USA, p. 539-553, 2003. Disponível em: <http://link.springer.com/chapter/10.1007%2F978-3-540-24644-2_35>. Acesso em: 30 ago. 2014.

LLVM. The LLVM Compiler Infrastructure. **Site oficial do LLVM**, 2015. Disponível em: <http://llvm.org/>. Acesso em 12 out. 2015.

LOUDEN, K. C. **Compiladores: Princípios e práticas**. 1ª ed. São Paulo: Thomson Pioneira, 2004. 351 p.

MORGAN, R. **Building an Optimizing Compiler**. 1ª ed. Estados Unidos: Digital Press, 1997. 451 p. Disponível em: <http://turbo51.com/download/Building-an-Optimizing-Compile-Book-Preview.pdf>. Acesso em: 28 mai. 2015.

NAROFF, S. **Clang intro**. In: Apple, 2009. Disponível em: <http://llvm.org/devmtg/2008-08/>. Acesso em: 28 mai. 2015.

OPENMANDRIVA. OpenMandriva. **Site oficial do OpenMandriva**, 2016. Disponível em: <https://www.openmandriva.org>. Acesso em: 04 ago. 2016.

PARIZI, R. B. **Implementação e Avaliação da Técnica ACCE para Detecção e Correção de Erros de Fluxo de Controle no LLVM**. 2013. Dissertação (Mestrado em Ciência da Computação) - Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2013. Disponível em: <https://www.lume.ufrgs.br/bitstream/handle/10183/72923/000891172.pdf?sequence=1>. Acesso em: 18 mai. 2015.

PEREIRA, M. M. **Análises e Otimizações no LLVM – Resultados Experimentais.** Campinas, 2011. Disponível em: <http://pt.slideshare.net/MarcioMachadoPereira/otimizacoes-llvm>. Acesso em: 19 mai. 2015.

PINTO, J. P. F.; BANDEIRA, T. F. L.; FERNANDES, S. Construção de um Compilador para arquitetura IPNoSys. In: Escola Potiguar de Computação e suas Aplicações (EPOCA), Mossoró. **Anais Eletrônicos...** Mossoró, RN, p. 73–79, 2010. Disponível em: <http://www.di.uern.br/epoca2010/artigos/78448_1.pdf>. Acesso em: 19 jun. 2014.

PINTO, P. **Suggesting Loop Unrolling Using a Heuristic-guided Approach.** 2012. Dissertação (Mestrado em Informática e Engenharia da Computação) - Programa de Pós-Graduação em Informática e Engenharia da Computação, Faculdade de Engenharia da Universidade do Porto, Porto, Portugal 2012. Disponível em: <https://web.fe.up.pt/~ei07073/downloads/thesis.pdf>. Acesso em: 08 jun. 2015.

RANGEL, J. L. **Compiladores: Otimização de código**. Rio de Janeiro, 2000. Disponível em: <http://www.cin.ufpe.br/~mvpm/Compiladores/otim.pdf >. Acesso em: 20 abr. 2015.

RANGEL, J. L. **Compiladores: Tradução Dirigida pela Sintaxe**. Rio de Janeiro, 1999. Disponível em: <http://www.cin.ufpe.br/~mvpm/Compiladores/COMP5.zipd>. Acesso em: 28 mai. 2015.

RAUBER, T.; RÜNGER, G. **Parallel Programming: For Multicore and Cluster Systems**. Springer, New York, 2010.

REGO, R. S. L. S. **Projeto e Implementação de uma Plataforma MP-Soc usando SystemC.** 2006. Dissertação (Mestrado em Sistemas e Computação) - Programa de Pós-Graduação em Sistemas e Computação, Universidade Federal do Rio Grande do Norte, Natal-RN, 2006. Disponível em: < http://www.repositorio.ufrn.br:8080/jspui/bitstream/123456789/18031/1/RodrigoSLSR.pdf>. Acesso em: 07 jul. 2015.

SCHMIT, H.; LEVINE, B.; YLVISAKER, B. Queue Machines: Hardware Compilation in Hardware. In: *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Alexandria. **Anais Eletrônicos…** Alexandria, USA: IEEE Computer Society, p. 152 - 160, 2002. Disponível em: <http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=1106670&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D1106670>. Acesso em: 02 jul. 2015.

SILVA, L. C. **Algoritmos para Escalonamento de Instruções e Alocação de Registradores na Infraestrutura LLVM.** 2013. Dissertação (Mestrado em Ciência da Computação) - Programa de Pós-Graduação em Ciência da Computação, Universidade Federal do Mato Grosso do Sul, Campo Grande, 2013. Disponível em: <http:// http://lscad.facom.ufms.br/wiki/images/7/75/Presentation_quali_lucas.pdf>. Acesso em: 20 mai. 2015.

SOFTPEDIA. Disponível em: <http://news.softpedia.com/news/openmandriva-we-are-the-only-gnu-linux-distro-to-use-clang-as-the-main-compiler-504011.shtml>. Acesso em: 04 ago. 2016.

SPENCER, R.; Gordon HENRIKSEN, G. **LLVM Analysis and Transform Passes, Documentation for the LLVM System**, 2009. Disponível em: <http://www.llvm.org/docs/Passes.html>. Acesso em: 14 mai. 2015.

STALLMAN, R. M. **Using and Porting the GNU Compiler Collection (GCC)**. Boston: Free Software Foundation, 2001. Disponível em: <http://www.skyfree.org/linux/references/gcc-v3.pdf>. Acesso em: 30 abr. 2015.

STALLMAN, R. M. **Using GCC: The GNU Compiler Collection Reference Manual**. Boston: Free Software Foundation, 2003. Disponível em: <http://dl.acm.org/citation.cfm?id=1593499>. Acesso em: 25 abr. 2015.

WADLEIGH, K.; CRAWFORD, I. **Software optimization for high-performance computing. HP Professional Series**. New Jersey: Prentice Hall PTR, 2000. 377 p.

XAVIER, T. C. S. **Solução Integrada para os Problemas de Seleção e Ordenação de Fase**. 2014. Dissertação de Mestrado (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Ciência da Computação, Universidade Estadual de Maringá, Maringá, 2014. Disponível em: <http://www.din.uem.br/~mestrado/diss/2014/souzaxavier.pdf>. Acesso em: 12 mar. 2015.

ANEXO A – CONJUNTO DE INSTRUÇÕES DA ARQUITETURA IPNoSys

Codop	Instrução	Tipo	# Operandos	Descrição
0	ADD	Aritmética	2	Soma 2 inteiros
1	SUB	Aritmética	2	Subtrai 2 inteiros
2	MUL	Aritmética	2	Multiplica 2 inteiros
3	DIV	Aritmética	2	Divide 2 inteiros
4	NOT	Lógica	1	Negação de 1 valor
5	AND	Lógica	2	Conjunção de 2 valores
6	OR	Lógica	2	Disjunção de 2 valores
7	XOR	Lógica	2	Ou-exclusivo de 2 valores
8	RSHIFT	Deslocamento	2	Desloca *n* bits de um valor à direita
9	LSHIFT	Deslocamento	2	Desloca *n* bits de um valor à esquerda
10	LOAD	Acesso à Memória	1	Solicita um valor da memória
11	STORE	Acesso à Memória	Vários	Armazena um valor na memória
12	EXEC	Sincronização	1	Ordena a injeção imediata de um pacote
13	SYNEXEC	Sincronização	Vários	Ordena a injeção de um pacote após sincronização
14	SYNC	Sincronização	1	Sinal de sincronização para um pacote
15	RELOAD	Acesso à Memória	1	Retorna um valor carregado da memória
16	BE	Condicional	2	Desvia se igual
17	BNE	Condicional	2	Desvia se diferente
18	BL	Condicional	2	Desvia se menor
19	BG	Condicional	2	Desvia se maior
20	BLE	Condicional	2	Desvia se menor ou igual
21	BGE	Condicional	2	Desvia se maior ou igual
22	JUMP	Incondicional	0	Desvia incondicionalmente
23	COPY	Auxiliar	1	Copia 1 valor para outra instrução no mesmo pac
24	NOP	Auxiliar	0	Sem operação
25	SEND	Sincronização	2	Envia um valor para um ser inserido em um pac.
26	IN	Entrada/Saída	3	Recebe bytes do controlador de E/S
27	OUT	Entrada/Saída	3	Envia bytes ao controlador de E/S
28	WAKEUP	Sistema	1	Ordena a reinjetar um pacote antes interrompido
29	NOTIFY	Sistema	1	Notifica estado de um pacote
30	CALL	Procedimento	Vários	Faz a chamada de uma função/pacote
31	RETURN	Procedimento	2	Retorna o resultado de uma função para o chamador

Fonte: ARAÚJO (2012)