Relatório de implementação

Disciplina de Linguagens e Compiladores

- 2018

Luiz Fernando Copetti

Abstract— This article relates the activities concerned to the NOCA assembly code generation in the context of LINGPON 2.0. The latter is a programming language and the first an computer architecture, both for the Notifications Oriented Paradigm – NOP. Generation of NOCA assembly is implemented with C++. This work is the final project of the Language and Compiler discipline.

Index Terms— NOCA, LINGPON, Assembly

1 Introdução

A implementação do projeto de um compilador para a arquitetura NOCA foi desenvolvida como projeto final para a disciplina de Linguagens e Compiladores. Este projeto difere dos demais devido à natureza particular da arquitetura contemplada. A seguir serão feitas algumas considerações a respeito de suas necessidades específicas. A arquitetura NOCA é especificamente destinada a aplicações do PON (Paradigma Orientado a Notificações). A geração de código assembly para esta arquitetura, a partir de um programa fonte escrito na linguagem de programação do PON, a LINGPON em sua versão 2.0, é o principal objetivo deste trabalho. Adicionalmente é desejada a utilização do grafo PON para consecução desta tarefa.

O item 2 deste trabalho descreve sucintamente os conceitos do PON, LINGPON, NOCA bem como uma motivação adicional para seu entendimento. Segue o próximo item (3) com a descrição da implementação. Testes, dificuldades encontradas, propostas de novos trabalhos e completam o artigo.

2 Conceitos básicos

2.1 PON

O paradigma orientado a notificações (PON) foi proposto por Simão [1] e consiste em uma nova abordagem para a computação em sistemas que possuam grande heterogeneidade em seus componentes.

O autor faz referência a sistemas de produção onde os recursos e produtos teriam uma certa "inteligência". A estas entidades "inteligentes" foi dado o nome de holon.

Estes sistemas de produção "holônicos", além de conterem os "hólons", precisam de um meio para controle da interação e cooperação entre estes componentes.

É proposta, então, a "holonificação" dos recursos do sistema em questão (*Resource-HL*). O controle passa a ser conhecido como HC (controle "holônico").

A proposta do PON reside em acoplar estas instâncias (hólons) à unidade de controle acima mencionada. Para tanto um sistema baseado em regras (RBS) foi utilizado. Cada hólon é tratado como um recurso virtual que emula, para fins de controle e cooperação, o recurso real.

O meta-modelo proposto pode ser assim resumido:

_ as relações de causa e efeito são tratadas por entidades denominadas *Rules*. As deduções ou inferências são baseadas em notificações. Estas são

geradas pelos *Resource-HLs*, baseadas em seu conhecimento factual, como por exemplo seus estados. Cada *Rule* notificada executa, no momento devido, uma ação especificada.

Estes fatores dão ao controlador características de alta reatividade e desacoplamento dos elementos. Na implementação física deste modelo é vislumbrada a eliminação de avaliações computacionais redundantes.

Termos importantes para o entendimento das implementações de programas baseados em PON são:

_***FBE*** (*Fact Base Element*) ou Elemento da Base de Fatos, que possui **atributos**, como por exemplo variáveis de estado, e métodos que podem ser funções ou operações;

_ ***Rule***: modela os elementos lógico-causais. Cada uma das *rules* agrega uma condição e uma ação;

_ ***Premise***: expressão relacional sobre um ou dois atributos (componentes do FBE, acima);

_ ***Condition*:** define o cálculo lógico para determinar se uma regra é ou não ativada.

_ ***Instigation***: elementos disparados pela aprovação das *Rules.* Ativam métodos (parte da FBE), fechando a cadeia de notificações.

A figura 1 esclarece, de forma pictórica, o relacionamento entre os principais elementos do PON.

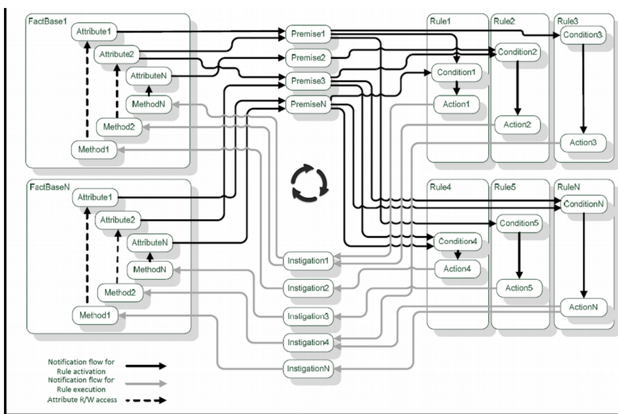

Figura 1 – Esquema de colaboração entre as entidades PON. Fonte: Linhares [3]

2.2 LINGPON

Para que o Paradigma Orientado a Notificações tivesse uma implementação facilitada houve a iniciativa da criação da Linguagem de Programação do PON, denominada de LingPON, cuja descrição pode ser encontrada em [2].

As primeiras implementações baseadas no PON utilizavam estruturas de dados demasiadamente custosas como descreve Ferreira [2].

Outra iniciativa para melhorar o porte das aplicações baseadas em PON para as diversas plataformas é a utilização do Grafo PON. Esta construção visa simplificar e padronizar as estruturas de dados derivadas de programas escritos em LINGPON. Ao invés de cada compilador destinado a plataformas específicas partir do processamento do código fonte PON, uma estrutura de dados intermediária é utilizada, o grafo PON.

Este também introduz o conceito de instâncias de FBEs e com estas a possibilidade de criação de regras de escopo, ao contrário das versões iniciais de implementações do PON.

2.3 NOCA - estrutura

Ao contrário das propostas de uso do PON em computadores baseados na arquitetura Von Neumann, a arquitetura NOCA [3] – *Notification-Oriented Computer Architecture* - propõe uma solução computacional diretamente associada às características do PON.

Antes de se fazer uma descrição mais detalhada da NOCA, o autor apresenta como provocação um experimento desenvolvido para implementação, em *hardware*, de um algoritmo ~~genético~~ baseado em enxame de partículas (PSO-Particle Swarm Optimization [?referência?]);

Desejava-se implementar, em lógica programável o seguinte algoritmo:

$$V_i^{t+1} = wV_i^t + c_1 r_{i1}^t \left(P_i^t - X_i^t \right) + c_2 r_{i2}^t \left(P_g^t - X_i^t \right) \quad (1)$$

$$X_i^{t+1} = X_i^t + \chi V_i^{t+1} \quad (2)$$

Como proposta de solução o seguinte circuito foi proposto:

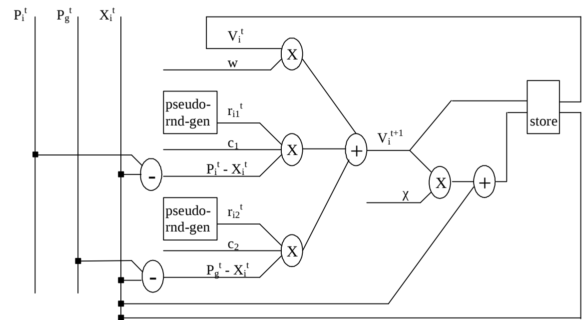

Figura 2 – PSO em *hardware*

Esta provocação foi usada por ter algumas similaridades com o PON:

Os valores P_i^t , P_g^t e X_i^t teriam um paralelo com os Atributos, a estrutura de multiplicadores e somadores bem como a realimentação através do elemento "store" seriam, em algum grau, um paralelo com a cadeia de premissas, condições, regras e instigações.

Apesar da analogia não ser perfeita, tal experimento denotou a extrema complexidade da realização física de uma proposta algorítmica.

Como veremos a seguir, a NOCA permite que alguns dos elementos que foram implementados “ad hoc” possam ter um metodologia de implementação.

Linhares, em [3] e [5], esquematiza a microarquitetura da NOCA, conforme figura 3.

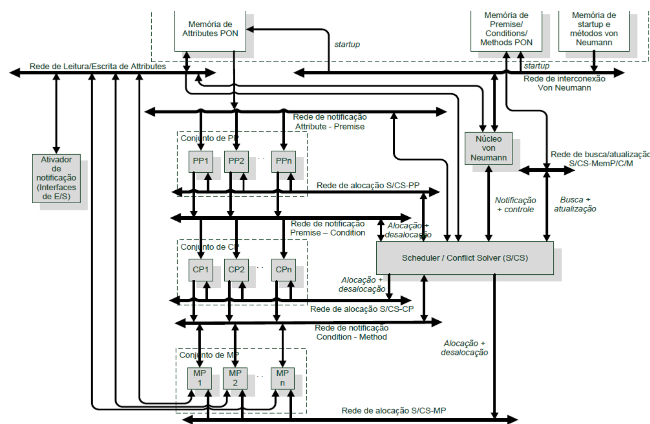

Figura 3: Organização microarquitetural da NOCA. Fonte: Linhares [5]

A realização em hardware do PON esbarra na baixa flexibilidade desta plataforma. Para aumenta-la, Linhares [3] propôs uma solução híbrida, em que a flexibilidade de modificação das soluções baseadas em *software* pudesse ser aliada ao desempenho das soluções em *hardware*.

Kerschbaumer [6] propôs a solução PON HD, mas esta tem uma limitação de tamanho de programa diretamente associada ao tamanho do dispositivo de *hardware* empregado.

Para tanto, foram implementados processadores associados aos principais elementos do PON:

_ Memória de atributos: guarda os valores iniciais dos atributos. Estes valores podem ser modificados ao longo da execução e novamente armazenados. Além disso podem ser usados como parâmetros do processador de métodos (MP)

_ Processador de Premissas (PP). Recebe notificações dos atributos e/ou das premissas e geram informações para que que os processadores de condições possam operar.

_ Processadores de condições (CP). Computam as notificações provenientes das premissas e também de condições. Como os processadores só podem ter dois operandos, muitas vezes é necessária a alocação de múltiplos CPs para processar condições/premissas em quantidade superior a duas.

_ Processadores de método: utilizam as notificações geradas em conjunto pelos PPs e CPs para processar os atributos / premissas. Notar que as *rules* não aparecem no diagrama. Elas existem apenas no programa PON e servem para montar a relação entre as premissas que ativariam determinada regra. Se tal ativação acontecer, isto será refletido no carregamento do PP com as informações correspondentes.

De acordo com Linhares [3] e [5], o número de processadores (PP, CP e MP) pode ser configurado de acordo com a capacidade do dispositivo programável em que a NOCA será instanciada. Isto permite que programas PON maiores sejam carregados no dispositivo.

A limitação passa a ser o tamanho das memórias do dispositivo (atributos, premissas, condições, métodos PON e startup/métodos von Neumann).

2.3 NOCA - funcionamento

Completando a base de conhecimentos necessária para compreensão da geração de código para a NOCA, passamos a uma descrição sucinta de sua operação.

Quando a NOCA é inicializada, precisa ter as memórias (atributos, premissas, condições, métodos PON) carrega-das com os valores necessários à consecução das ações relativas à “execução” de um determinado programa PON)

Nas implementações usuais do PON, todos oe elementos notificadores e recebedores de notificação estão presentes na plataforma alvo assim que o programa é carregado. “Programa” neste contexto é um conjunto de dados compatível com a plataforma alvo, que, ao ser carregado na mesma permite o acontecimento do ciclo de notificações PON.

Em se tratando da NOCA, há a limitação do número de premissas, atributos, condições e métodos PON dispo-níveis em cada instante. Desta forma, e com o auxílio da figura 3, depreende-se que a memória de startup deve ser configurada para carregar os atributos, condições, métodos e premissas necessárias à “execução” do programa PON em questão.

Caso um atributo, método, condição ou premissa não esteja carregado nos respectivos processadores, estes são carregados através das várias “redes de alocação”. Se um determinado atributo, método, condição ou premissa é frequentemente usado, é natural que o mesmo não seja desalocado do processador correspondente.

Isto teria o efeito similar ao das memórias “cache” em máquinas von Neumann. A existência de um número adequado de processadores faria com que esta alocação/desalocação fosse reduzida a um mínimo necessário para atendimento de requisitos de latência do sistema PON.

Os diversos processadores da NOCA precisam ser carregados com valores específicos de parâmetros

para poder realizar as operações de encadeamento de notificações.

Na próxima seção discutiremos como cada uma destas instruções foi gerada a partir de programas PON, escritos em LINGPON 2.0. Estes programas são processados para a geração do grafo PON correspondente. De posse do grafo PON passamos à geração das instruções do NOCA *assembly*.

A descrição sucinta de cada instrução será acompanhada do procedimento adotado para sua geração.

3 IMPLEMENTAÇÃO

3.1 Geração das instruções *assembly*

As instruções *assembly* para carregamentos dos processadores da NOCA estão definidas em Linhares [3], e um bom resumo está em Linhares [5]. Esta última serviu de base para a descrição abaixo.

ATTRIBUTE-DECL, composta pelo valor de um determinado *Attribute*, configurações específicas deste atributo com relação ao seu papel no mecanismo de notificações e uma lista de *Premises* a serem notificadas quando seu valor é alterado. Para cada atributo contido em determinado programa PON, uma instrução deste tipo é requerida. Esta instrução é armazenada na memória de *Attributes*, conforme mostrado na figura 3.

Os atributos são relativamente independentes de outros elementos do grafo PON e foram obtidos a partir da busca recursiva das instâncias do grafo PON. Abaixo o exemplo de geração de código *assembly*. O número 1, abaixo, representa que este atributo, ao ser modificado, produz uma notificação para 1 premissa. Esta premissa é listada na sequência, como seriam outras, caso o número citado fosse maior que 1.

```
sectorA.alarmA.atStatus:  ATTRIBUTE-DECL,,,N,,1,prAlarmAOn ;
```

Houve necessidade de modificação no mecanismo de recursão. Cada instância de nível mais alto passa à de nível mais baixo seu nome, para que o nome do atributo possa ser montado como no exemplo mostrado acima.

PREMISE-OP, define o comportamento do processador de premissas (PP) e define de quais atributos a premissa em questão depende. Para cada premissa é necessária uma instrução deste tipo. Outro componente da instrução é a operação relacional a ser realizada sobre os operandos e, de maneira similar à instrução específica dos atributos, uma lista das *Conditions* que devem ser notificadas em razão da mudança do valor lógico da premissa.

A seguir, um exemplo do estágio atual de geração das premissas é apresentado:

```
sectorA.prAlarmBOn                    PREMISE-OP,,,R,,,AA
```

CONDITION-OP, indica para o processador de condições (CP) quais as premissas que o afetam, a operação lógica que será efetuada sobre os valores destas premissas e a lisia de métodos que serão notificados quando o valor de saída do CP for alterado.

Uma particularidade da NOCA é que os processadores de condições só conseguem operar sobre duas premissas. Desta forma, no caso em que houver mais de duas premissas, estas devem ser combinadas de forma a produzir resultados intermediários que devem ser aplica-dos a outros processadores de condições.
Ainda não estão sendo geradas estas instruções.

METHOD-OP, esta instrução carrega o processador de métodos com os parâmtetros que definem: de qual condição o método depende, endereços dos operandos a serem lidos e o atributo onde o resultado será armazenado. Além disso é definida a operação lógico-aritmética que o MP executa e opcionalmente o endereço de um outro método que será executado em sequência, chamado de método dependente. A geração de métodos está na fase de percorrimento do grafo.

METHOD-VN-OP, em casos específicos onde é necessária a execução de código sequencial, esta instrução permite que programas diretamente escritos em linguagem de máquina do processador *softcore* NIOS utilizado na NOCA. Estes métodos são disparados pelo escalonador (S/CS). A classe "CodeBlock" pode carregar as informações.

3.2 Particularidades da geração de código

A NOCA precisa que as instruções de um mesmo tipo sejam agrupadas, pois devem ocupar endereços de memória sequenciais. Abaixo é mostrado um código *assembly* NOCA, com o agrupamento que foi citado.

```
#define VN.counter.mtEnd                        4029c end_method

counter.count:          ATTRIBUTE-DECL,,,N,,2,0,PrCounterEquals5,PrCounterLess5;

PrCounterEquals5:       PREMISE-OP,,,,,,AV,==,1,counter.count,5,CdCounterEquals50;
PrCounterLess5:         PREMISE-OP,,,R,,,AV,<,1,counter.count,5,CdCounterLess50;

@c00020
CdCounterEquals50:      CONDITION-OP,,,,,0,!!,2,PrCounterEquals5,,counter.mtEnd,
                        counter.mtRst;
CdCounterLess50:        CONDITION-OP,,,R,,0,!!,1,PrCounterLess5,,counter.mtInc;

@c00050
counter.mtInc:          METHOD-OP,,,,,AV,INC,IC=0,0,CdCounterLess50,counter.count,,
                        counter.count;
counter.mtRst:          METHOD-OP,,,,,VV,=,IC=0,0,CdCounterEquals50,0,,counter.count;
counter.mtEnd:          METHOD-VN-OP,VN.counter.mtEnd,0;
```

As instâncias são percorridas a partir da instância de mais alto nível, a partir da FBE main. Para que as instruções assembly fiquem organizadas de forma sequencial foi necessária a gravação das instruções em

cinco arquivos diferentes, que ao final do processo foram concatenados.

Os arquivos bison.y; flex.l; target.h e target.cpp foram modificados para a inclusão da variante de compilação 3 como sendo geração de código *assembly* para a NOCA.

Uma linha de compilação fica então:

./Compiler/NOPL 3 ./Applications/Sensor

(executado no diretório "compiler")

4 Testes

4.1 Sensors

A aplicação mais utilizada para teste de geração de código *assembly* foi a denominada de "Sensor". Foi escolhida por ter vários níveis de instâncias permitindo explirar uma das principais características da LINGPON 2.0 que é a segmentação dos elementos PON em instâncias.

4.1 NocaSim

O ambiente de testes NocaSim é um simulador da NOCA foi preparado para receber o código gerado. Foram feitas experiências com código gerado manualmente para LINGPON mas não foi possível a execução de testes com o compilador tratado neste artigo.

Além do simulador foi preparado o ambiente de compilação de arquivos NOCA e de configuração da NOCA (onde são escolhidos, entre outros o número de processadores utilizados na simulação)

5 Dificuldades

A premissa inicial da tarefa era de que haveria um mapeamento simples da estrutura do grafo PON para as instruções *assembly* da NOCA. Este mapeamento revelou-se bastante mais compilcado do que o pensado inicialmente.

O desconhecimento dos detalhes do paradigma PON impactaram grandemente na compreensão do problema. A inversão das disciplinas (compiladores e depois PON) influiu bastante neste quesito. Admito, porém, que ao final do ciclo das duas disciplinas, o total de conhecimentos apreendido pode ser maior do que na abordagem tradicional (PON seguido por compiladores).

A arquitetura da NOCA é razoavelmente complexa e demandou muito tempo para seu conhecimento, ainda que superficial em muitos aspectos.

Todos os ferramentais necessários tais como:

_ ferramentas de controle de versão, "*commit*", "*merge*"

_ compilador NASM,

_ simulador NocaSim

_ documentação das classes do grafo PON

não são acessíveis de uma maneira produtiva. Haveria necessidade de uma revisão bastante grande nas estruturas destes documentos / ferramentas.

O projeto final explorou mais a temática PON do que o objetivo, ao menos nominal, da disciplina que seria uma discussão sobre aspectos de linguagens e compiladores.

Houve uma dificuldade, em especial, no entendimento e codificação correta do algoritmo de varredura do grafo.

Outra barreira foi a necessidade de montar nomes de alguns dos elementos baseados em sua "ascendência", pelo fato de não haver métodos para que uma instância obtivesse o nome de seu "pai". Sugere-se, portanto, que tal método seja provido em uma próxima revisão do grafo PON.

6 Considerações Gerais

No caso específico da NOCA, em que toda a estrutura segmentada do grafo PON tem que ser "desmontada" há a necessidade de um trabalho posterior para verificar se a estrutura do grafo PON realmente traz ganhos de produtividade na programação para esta plataforma.

7 Conclusão

Como sugestão de trabalhos futuros:

_ Finalização da implementação do projeto de gerador de *assembly* para NOCA em LINGPON 2.0.

_ Elaboração de uma aplicação real tal como o Simulador de Controle de tráfego e teste da solução.

Referências

[1] J. M. Simão, "A Contribution To The Development Of A HMS Simulation Tool And Proposition Of A Meta-Model For Holonic Control," Tese. Doutorado em Engenharia Elétrica e Informática Industrial - CPGEI. Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba, p. 168, 2005

[2] C. A. Ferreira, *Linguagem e Compilador para o Paradigma Orientado a Notificações (PON): Avanços e Comparações*, Dissertação de Mestrado, PPGCA/UTFPR, 2015.

[3] R. R. Linhares. A Contribution to the Development of a Computer Architecture Proper to the Notification Oriented Paradigm",Tese. Doutorado em Engenharia Elétrica e informática Industrial – CPGEI. Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba

[4] L.F. Copetti. Relatório de implementação do algoritmo PSO (Particle Swarm Optimization) em hardware, utilizando-se lógica programável. CPGEI, outubro de 2005

[5] R. R. Linhares, J. M. Simao and P. C. Stadzisz, NOCA – A Notification-Oriented Computer Architecture. IEEE Latin Amarica Transactions, Vol. 13, No. 5, May 2015.

[6] R. Kerschbaumer, Paradigma Orientado a Notificações para síntese de lógica reconfigurável. 2017. 6 f. Texto de qualificação (Doutorado em Engenharia Elétrica e Informática Industrial). Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI). Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, 2017

[7] A. F. Ronszcka, G. Z. Valença; R. R. Linhares; P. C. Stadzisz, J. M. Simão. *Notification-Oriented Paradigm Framework 2.0: An Implementation Based On Design Patterns*. IEEE LA - IEEE Latin America Transactions, Vol. 15, Issue 11, Nov. 2017. ISSN: 1548-0992.

Luiz Fernando Copetti. Natural de Curitiba-PR, nascido em 1967. Mestre em Informática Industrial pelo CPGEI, Curso de Pós-graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná, no ano de 2008. Professor do ensino básico técnico e tecnológico da UTFPR, desde 1991. Tem experiência como engenheiro de desenvolvimento em empresas como a Siemens, além de vivência internacional na Alemanha, Finlândia e Itália, tanto como desenvolvedor quanto como líder de projetos.
http://lattes.cnpq.br/.