Avaliação de alternativas para parametrização de simuladores: Um estudo de caso aplicado ao SiNUCA3

Gabriel G. de Brito[1], **Marco A. Z. Alves**[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)

{ggb23,mazalves}@inf.ufpr.br

Resumo. *Simuladores de sistemas computacionais empregam soluções de parametrização para definir características do sistema a ser simulado. Para permitir e auxiliar no trabalho dos pesquisadores que utilizarão a ferramenta, tal solução deve ser desenvolvida de acordo com seus casos de uso esperados. Considerando isso e, comparando soluções existentes, desenvolvemos o mecanismo de parametrização para um novo simulador, baseado em C++ e YAML.*

1. Introdução

A simulação em software é uma técnica importante para a pesquisa moderna em arquitetura de computadores. Ela permite o estudo de sistemas computacionais existentes ou novos, sem o custo de desenvolver hardware específico. Para tal, diversos simuladores com diferentes características e focos estão em uso atualmente [Akram and Sawalha 2019]. Cada simulador emprega uma solução para definir o sistema computacional a ser estudado. Simuladores como Multi2Sim [Ubal et al. 2012] e Simulator of Non-Uniform Cache Architectures (SiNUCA) [Alves et al. 2015] utilizam arquivos de texto em linguagens de configuração simples (INI [Corporation 2013] e libconfig [Lindner 2021]), enquanto simuladores como gem5 [Binkert et al. 2011] utilizam linguagens de programação completas (Python).

Neste trabalho visamos estudar e estabelecer a melhor forma para parametrização do simulador em desenvolvimento Simulator of Non-Uniform Cache Architectures, Version 3 (SiNUCA3). Para isso, focaremos nos objetivos básicos do novo simulador, justificando a escolha da linguagem de configuração YAML Ain't Markup Language (YAML) [Ben-Kiki et al. 2009] junto da programação de componentes de simulação em C++.

2. Linguagens de Configuração Consideradas Para o SiNUCA3

JavaScript Object Notation (JSON) [International 2017] é uma linguagem de troca de dados. Por ter sido desenvolvida para uso em aplicações JavaScript e na *World Wide Web*, é conhecida entre programadores e cientistas de dados. Sua sintaxe simples contém apenas valores primitivos, objetos e vetores. Porém, a linguagem não possui expressividade para adequadamente definir as relações entre componentes microarquiteturais, pois não há sintaxe para criar referências a objetos.

Lua [Ierusalimschy 2006] é uma linguagem de programação criada para ser integrada em aplicações, possibilitando sua extensão e configuração. Por ser uma linguagem completa, possui expressividade para definir as relações necessárias e a configuração de componentes arquiteturais. Em contrapartida, exige um certo aprendizado por parte do pesquisador, além de aumentar a complexidade do simulador.

libconfig [Lindner 2021] é uma biblioteca C++ para leitura de arquivos de configuração. Devido à experimentos anteriores, incluindo o seu uso no SiNUCA, sua sintaxe foi considerada mais complexa que o necessário e pouco adequada aos casos de uso do novo simulador, principalmente pelo fato de a criação de referências ser realizada por meio da inclusão de arquivos, o que forçaria qualquer configuração com mais de um componente a ser divida.

YAML é uma linguagem de troca de dados semelhante a JSON porém possui sintaxe para criar referências. Na linguagem, objetos são chamados de *mappings*, e referências com nome de *alias*.

YAML foi escolhida para uso no SiNUCA3 pela facilidade, disponibilidade e versatilidade da biblioteca LibYAML [Project 2020], além da capacidade de expressar referências.

3. Casos de uso do SiNUCA3

O SiNUCA3 é baseado no SiNUCA, um simulador validado com precisão de ciclo e orientado a traços [Alves et al. 2015]. Baseado na arquitetura x86, simula tanto aplicações *single* e *multithread*. O novo simulador está sendo desenvolvido principalmente para: 1) Avaliar a eficácia de novos componentes microarquiteturais propostos bem como; 2) Efetuar exploração de espaço de projeto de componentes existentes quando seus parâmetros de funcionamento são modificados. Os dois casos são diretamente conflitantes quanto à escolha da solução de configuração.

Para adequadamente cobrir o primeiro caso de uso, deve ser possível programar o comportamento de um componente microarquitetural inédito ou derivado de um pré-existente. Isso inclui a descrição de algoritmos que serão executados a cada ciclo de relógio, possivelmente modificando o estado interno do componente e comunicando-se com o resto do sistema simulado. Linguagens de programação, em especial as com suporte à orientação a objeto são, portanto, uma escolha natural para cobrir esse caso. Por exemplo, o simulador gem5 usa a linguagem Python para descrever e programar os componentes simulados, dessa maneira cobrindo esse caso de uso.

Para o segundo caso de uso, o uso de linguagens de programação apresentam complexidade excessiva, aumentando a curva de aprendizado do simulador. Por outro lado, linguagens de configuração mais simples facilitam o uso. O simulador gem5 mantém esse caso de uso coberto por meio da configuração em Python, enquanto simuladores como Multi2Sim e SiNUCA utilizam linguagens de configuração simplificadas.

Enquanto as linguagens de programação cobrem ambos os casos de uso (de maneira não ótima no segundo). As linguagens de configuração não tem expressividade para descrever o funcionamento de componentes inteiros. No caso do SiNUCA, não há alternativa suportada para cobrir o primeiro caso de uso exceto a reprogramação do simulador em si, que necessita de conhecimento acerca do seu funcionamento interno, edição do sistema de compilação (*Makefile*), recompilação de partes significativas do simulador e a religação do programa inteiro.

Para abranger ambos os casos de maneira ótima, o SiNUCA3 possui a noção de classes e componentes análoga à adotada pelo paradigma de programação orientada a objetos. Enquanto as classes simulam o comportamento de componentes arquiteturais,

o arquivo de configuração descreve as instâncias de cada classe, a relação entre elas e possíveis parâmetros que os componentes receberão. Dessa maneira, é possível criar configurações complexas de maneira simples e sem repetição. Além disso, esse paradigma funciona para simular sistemas computacionais semelhantes aos modernos e sistemas radicalmente diferentes, facilitando assim o estudo de ideias inovadoras na área.

4. Programação de Componentes no SiNUCA3

A programação dos componentes no novo simulador é feita em C++. O sistema de compilação utilizado é um *Makefile* que, por meio do utilitário de linha de comando *find*, detecta todos os arquivos no código-fonte. Ele permite assim que qualquer arquivo adicionado no diretório *custom_components* seja automaticamente compilado e ligado ao programa sem intervenção manual. Cada componente deve ser uma classe, que herda da classe *Component*.

Para o simulador encontrar os componentes durante a execução, eles devem ser declarados no arquivo *custom_components/custom.cpp*. A declaração é feita com dois macros *COMPONENTS(...)* e *COMPONENT(type)*, onde passa-se o nome da classe do componente por meio do parâmetro *type*. A figura 1 mostra o conteúdo do arquivo com apenas o componente exemplo *CustomExample* declarado.

Com essa abordagem, para adicionar ou modificar um componente, apenas o código do próprio componente e do arquivo *custom_components/custom.cpp* precisarão ser recompilados. O usuário não precisa ter conhecimento acerca do sistema de compilação ou funcionamento interno do simulador exceto quanto aos métodos da classe *Component*. Além disso, espera-se uma boa performance devido ao fato de que o código dos componentes é compilado diretamente no simulador, ao invés de ser executado em um interpretador de linguagem de mais alto nível.

```
#include "../sinuca3.hpp"
#include "custom_example.hpp"

COMPONENTS(
    COMPONENT(CustomExample);
)
```

Figura 1. Conteúdo do arquivo *custom_components/custom.cpp*

```
l2Cache: &sharedl2Cache
  class: SimpleCache

l1Cache:
  class: SimpleCache
  memory: *sharedl2Cache
```

Figura 2. Configuração em YAML com duas definições e um *alias*.

5. Configuração de Parâmetros no SiNUCA3

A configuração dos parâmetros irá utilizar a linguagem YAML como descrito na seção 2. No arquivo de configuração, serão declarados quais componentes serão instanciados e quais parâmetros serão passados para cada componente. Todos os componentes implementam o método virtual *SetConfigParameter*, que o simulador chama para passar parâmetros ao componente. Dessa maneira, os componentes tem acesso a toda sua configuração, e podem tratar os parâmetros de maneira customizada.

Configurações de componentes (no simulador chamadas de "definições") são descritas como objetos na linguagem. Caso o nome da definição seja passado como parâmetro em outra parte da configuração, uma nova instância dessa classe será criada

e terá os mesmos parâmetros passados. Um ponteiro para essa instância é passado ao componente (ou ao próprio simulador) cuja configuração referenciou a definição. No simulador, isso é chamado de "referência à definição". Na definição, *class* é utilizado pelo simulador para saber qual classe C++ deve ser instanciada, e não passado para o componente.

Opcionalmente, por meio do recurso de *alias* (referências nomeadas), é definido um nome para uma instância única junto da definição. Caso a definição possua *alias*, uma instância da mesma será criada. Caso esse *alias* seja referenciado em outra parte do arquivo, um ponteiro para essa instância única é passado. No simulador, isso é chamado de "referência à instância".

Na figura 2, são feitas duas definições: *l2Cache* e *l1Cache*. Sempre que o nome de uma definição for referenciado, uma nova instância dela será criada. No caso da definição *l2Cache*, existe uma instância única com o *alias sharedl2Cache*. Na definição de *l1Cache*, esse *alias* é referenciado por meio do parâmetro *memory*. Dessa maneira, todas as instâncias de *l1Cache* terão um ponteiro para a mesma instância única de *l2Cache*.

Com a abordagem de configurar todos os componentes microarquiteturais com a mesma abstração, o simulador se torna versátil. Enquanto o SiNUCA, por exemplo, presume uma organização com memória principal e cache separadas, e a presença de um *prefetcher*, o SiNUCA3 permite a configuração de hierarquias e componentes de memória incomuns e inéditos.

6. Conclusão

A parametrização de simuladores pode ser feita com diferentes tecnologias. A escolha da solução utilizada em uma ferramenta deve ser feita considerando-se seus casos de uso. Dessa maneira e, comparando a experiência de simuladores anteriores, foi criada a solução de parametrização do SiNUCA3.

Referências

Akram, A. and Sawalha, L. (2019). A survey of computer architecture simulation techniques and tools. *IEEE Access*.

Alves, M. A. Z., Villavieja, C., et al. (2015). Sinuca: A validated micro-architecture simulator. In *Int. Conf. on High Performance Computing and Communications*.

Ben-Kiki, O., Evans, C., and Ingerson, B. (2009). Yaml ain't markup language (yaml™) version 1.1. *Working Draft*.

Binkert, N., Beckmann, B., et al. (2011). The gem5 simulator. *ACM SIGARCH Comp. Arch. News*.

Corporation, M. (2013). Configure an ini file item.

Ierusalimschy, R. (2006). *Programming in lua*. Roberto Ierusalimschy.

International, E. (2017). The json data interchange syntax. Standard, ECMA Int.

Lindner, M. (2021). Libconfig.

Project, T. Y. (2020). Libyaml.

Ubal, R., Jang, B., et al. (2012). Multi2sim: A simulation framework for cpu-gpu computing. In *Int. Conf. on Parallel architectures and compilation techniques*.