Mecanismos de Comunicação entre *Clusters* para Lightweight Manycores no Nanvix OS

João Vicente Souto
UFSC
Florianópolis, Brasil
joao.vicente.souto@grad.ufsc.br

Pedro Henrique Penna
UGA, PUC Minas
Grenbole, França
pedro.penna@sga.pucminas.br

Márcio Castro
UFSC
Florianópolis, Brasil
marcio.castro@ufsc.br

Henrique Freitas
PUC Minas
Belo Horizonte, Brasil
cota@pucminas.br

***Resumo*—Ambientes de desenvolvimento para *lightweight manycores* são onerosos e suscetíveis a erros. Eles pecam, principalmente, em prover uma boa relação entre programabilidade e portabilidade. Neste contexto, este trabalho propõe mecanismos de comunicação entre clusters para um sistema operacional distribuído projetado para essa classe de processadores. Estes mecanismos buscam ser precisos, fáceis de usar, escalonáveis e facilmente portáveis. Os resultados mostraram ser possível suportar algoritmos de comunicação coletiva de forma eficiente utilizando os mecanismos propostos em um *lightweight manycore* real.**

***Index Terms*—HAL, Sistema Operacional Distribuído, *Lightweight Manycore*, Kalray MPPA-256**

I. Introduction

ABC.

II. Fundamentação Teórica

Esta seção apresenta uma breve introdução aos conceitos e objetos de estudo utilizados neste trabalho. Primeiro, abordamos o Sistema Operacional (OS) base para implementação dos mecanismos propostos. E por fim, detalhamos a arquitetura do *lightweight manycore* utilizado nos experimentos.

A. Processadores Lightweight Manycore

A classe de *lightweight manycores* é constituído de processadores que buscam prover um alto nível de paralelismo e baixo consumo energético. Eles se diferem de outras arquiteturas em diversos pontos como:

i. integram centanas de núcleos de baixo potência em um único chip organizados em clusters;
ii. são projetados para lidar com workloads Multiple Instruction Multiple Data (MIMD);
iii. dependem de uma Rede-em-Chip (NoC) para uma comunicação através de troca de mensagens rápida e confiável;
iv. possuem sistemas de memória restritivos; e
v. integram componentes heterogêneos.

Junto de uma escalabilidade superior, *lightweight manycores* trouxeram um novo conjunto de desafios no desenvolvimento de software proveniente de suas particularidades arquiteturais. Precisamente, eles introduziram as seguintes dificuldades:

Agradecimentos ao CNPq e a CAPES pelo auxílio financeiro através de bolsa IC para a realização deste estudo.

Modelo de Programação Híbrida advém da natureza paralela e distribuída da arquitetura. Requer frequentemente que seja adotado o padrão de troca de mensagens para comunicação entre *clusters* e um modelo de memória compartilhada dentro do *cluster* [**?**].

Falta de suporte em hardware para coerência de cache é utilizada para reduzir o consumo de energia. Sem essa obrigatoriedade, os programadores precisam lidar explicitamente com a coerência e frequentemente precisam reprojetar suas aplicações [1].

Sistema de memória restritivo apresentam multiplos espaços de endereçamento e pequenas memórias locais, requisitando o particionamento e *prefetching* dos dados para manipulação [**?**].

Configuração heterogênea requer a programação de componentes distintos tornando o *deploy* de aplicações mais complexa [2].

O MPPA-256 é um *lightweight manycore* projetado pela empresa francesa Kalray. A Figura **??** ilustra a versão *Bostan* utilizada neste trabalho. Esta versão integra 288 núcleos agrupados em 16 Compute Clusters e 4 I/O Clusters. De um lado, os Compute Clusters possuem 16 Processing Element (PE), 1 , 2 MB de SRAM e 1 interface NoC e são dedicados a carga de trabalho do usuário. Por outro lado, os I/O Clusters possuem 4 , 4 MB de SRAM e 4 interfaces NoC destinados para comunicação com periféricos. O espaço de endereçamento de cada *cluster* é privado, forçando a comunicação de trocas de mensagens por duas diferentes 2-D Torus NoCs. De um lado, a Control NoC (C-NoC) é destinada a troca de pequenas mensagens de controle. Do outro lado, a Data NoC (D-NoC) permite a troca de quantidades arbitrárias de dados. Adicionalmente, todos os *clusters* possuem uma Direct Memory Access (DMA) habilitando a comunicação assíncrona.

B. Sistemas Operacionais para Lightweight Manycores

Um OS é uma peça fundamental na constituição de um sistema computacional moderno. Sua principal função é prover serviços que abstraem a arquitetura física a baixo deles. Isto aumenta significativamente a programabilidade, produtividade e confiabilidade no desenvolvimento de aplicações computacionais. Além disso, existem padrões que permitem a compatibilidade entre OSs, permitindo o *deploy* de aplicações

instâncias de OSs distintas. Um exemplo de padrão é o Portable Operating System Interface (POSIX).

Apesar da grande facilidade a programação de sistemas trazida pelos OSs, a grande maioria deles não se adequa as novas arquiteturas que andam surgindo atualmente. Por exemplo, os OSs mais comuns são monolíticos e projetados para trabalhar em um sistema com memória compartilhada e com uma quantidade pequenas de núcleos de processamento. Este fato levou a pesquisa de projetos alternativos para esta nova era de processadores ([**?**]). Entretanto, tais alternativas são focadas em extrair o melhor desempenho destas plataformas, deixando uma lacuna para uma solução que equilibre custos de desenvolvimento com desempenho, principalmente para *lightweight manycores*.

Neste contexto, Nanvix OS nasce com o foco nos desafios de portabilidade e programabilidade surgidos juntos com os *lightweight manycores* ([**?**], [**?**]). A proposta do Nanvix é repensar o design do OS a partir de seus principios básicos sem perder a compatibilidade com outros OSs ([**?**], [**?**]). Para isso, o Nanvix adaptará o conceito de um *multikernel* que seja compatível com o padrão POSIX. Ao todo, o Nanvix é constituído de três camadas, *multikernel*, *microkernel* e Camada de Abstração de Hardware (HAL).

A Figura **??** ilustra o funcionamento do *multikernel*. Sobre a pespectiva do OS, os *clusters* são nós que troca mensagens entre si para solicitar ou prover serviços. Neste ponto, é possível notar que os serviços do OS agora são processos ativos e ficam isolados dos processos do usuário. Dentro de cada *cluster*, temos um *microkernel* assimétrico. A Figura **??** apresenta a estrutura do *microkernel* e da HAL. O *microkernel* isola a manipulação e controle das estruturas internas do em um único núcleo, eliminando os problemas de coerência de cache os *lightweight manycores*. Por fim, a HAL agrupa os *lightweight manycores* sobre uma perspectiva comum abstraindo as arquiteturas em seus principais componentes. Este trabalho está localizado no *microkernel* e na HAL.

III. Mecanismos de Comunicação entre *Clusters*

Os mecanismos de comunicação entre *clusters* propostos sintetizam três comportamentos regulares em sistemas distribuídos, i.e. sincronização, trocas de pequenas mensagens de controle e grandes transferência de dados. Denominadas, *sync*, *mailbox* e *portal*, eles exportam uma visão abstrata e padronizada dos recursos existentes em *lightweight manycores*. Esta seção aborda detalhes semânticos e de implementação de cada das abstrações mencionadas.

A abstração *Sync* provê a criação de barreiras distribuídas. Similar ao POSIX *Signals*, um *cluster* pode emitir uma notificação a um outro *cluster*. Entretanto, as notificações não transmitem valores, servem apenas para sincronização. A Figura **??** ilustra os modos de operação de um ponto de sincronização. O modo `ALL_TO_ONE` define que um *cluster* mestre (`ONE`) deve aguardar bloqueado N sinais de N *clusters* escravos. Contrariamente, o modo `ONE_TO_ALL` é responsabilidade do mestre sinalizar os N escravos, liberando-os do bloqueio. A criação de um *sync* requer apenas três parâmetros, i.e. a lista de *clusters* envolvidos, o identificador do *cluster* mestre e qual o modo de operação. Desta forma, é possível exportar um comportamento padronização sem requisitar detalhes do hardware. Por exemplo, a implementação no Kalray MPPA-256 utilizou apenas recursos da C-NoC que são alocados relativos aos parâmetros mas é invisível para a aplicação cliente.

A abstração *Mailbox* permite a troca de mensagens de tamanho fixo. Similar ao ix *Message Queue*, o *cluster* receptor aloca espaço suficiente para receber pelo menos uma mensagem de cada possível emissor. Para simplificar a implementação do lado do receptor, o emissor é responsável por emitir a mensagem para um local pré-determinado. Para garantir que o emissor não sobrecarregue o receptor, ou sobrescreva mensagens não lidas, o algoritmo da *mailbox* implementa um controle de fluxo onde o repector deve notificar o emissor depois de cada mensagem consumida. O emissor, por sua vez, aguardará uma notificação caso tenha uma mensagem prévia pendente. A implementação no Kalray MPPA-256 utiliza a D-NoC para emitir a mensagem e a C-NoC para emitir as notificações de controle. Como no *sync*, a *mailbox* relativisa os recursos utilizados nos conforme nos identificadores dos receptores e emissores.

A abstração *Portal* habilita a transferência de quantidades arbitrárias de dados. Definindo uma comunicação unidirecional entre dois *clusters*, o *portal* implementa um comportamento similar ao POSIX *Pipe*. Para garantir que o emissor não envie os dados antes que o receptor esteja apto a receber, o *portal* define um controle de fluxo que força o emissor esperar um sinal de permissão do receptor. O *portal* utiliza recursos da D-NoC para enviar os dados e C-NoC para emitir as permissões. A configuração desses recursos é relativa a quais *clusters* estão se comunicando.

Todas os mecanismos propostos exportam uma interface de baixo nível que abstream todos os detalhes de hardware dos *lightweight manycores*. A natureza distribuída é sintetizada através de identificadores parâmetros lógicos que simplificam a interface. Eles baseiam-se fortemente em soluções largamente utilizadas nos sistemas operacionais modernos trazendo conceitos familiares a desenvolvedores de forma geral.

IV. Experimental Results

In this section, we present our experimental results. First, we analyze outcomes for *Microbenchmarks*, and next we move to a discussion on the results for the *Synthetic Benchmarks*. For a detailed description about these experiments, see Section **??**.

A. Microbenchmarks

Figure 1 pictures the breakthrough of execution time for the *Local Kernel Call (L-Kcall) Benchmark*. Before proceeding with our analysis, it is important to note that hardware events that are depicted are not exclusive with each other. For example, an *I-Cache Stall* may incur in a *Register Stall*, thus the bar labelled *Total Cycles* is not the aggregation of the other bars. Noteworthy, this observation applies to the other plots that follow.

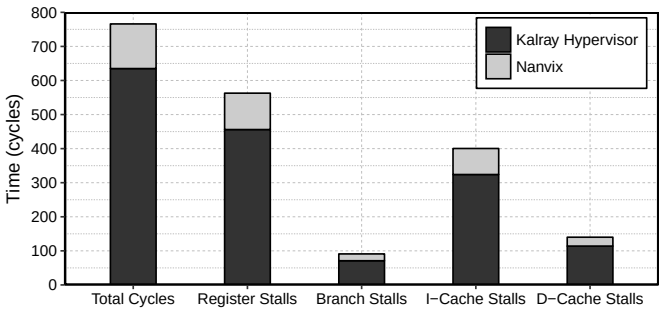

Figura 1. Execution breakthrough for the *L-Kcall Benchmark*.

V. Trabalhos Correlatos

ABC.

VI. Conclusão

ABC.

Referências

[1] E. Francesquini, M. Castro, P. H. Penna, F. Dupros, H. Freitas, P. Navaux, and J.-F. Méhaut, "On the Energy Efficiency and Performance of Irregular Application Executions on Multicore, NUMA and Manycore Platforms," *Journal of Parallel and Distributed Computing*, vol. 76, no. C, pp. 32–48, Feb. 2015.

[2] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran, "Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms," in *European Conf. on Computer Systems*, Bordeaux, France, Apr. 2015, pp. 1–16.