UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

PAULO JOSÉ BASTOS LEITÃO

AVALIAÇÃO DA TÉCNICA DO PREBAKING PARA REDUÇÃO DO COLD START EM SERVIÇOS FAAS PARA A RUNTIME DE PYTHON

CAMPINA GRANDE - PB

2021

PAULO JOSÉ BASTOS LEITÃO

AVALIAÇÃO DA TÉCNICA DO PREBAKING PARA REDUÇÃO DO COLD START EM SERVIÇOS FAAS PARA A RUNTIME DE PYTHON

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. Thiago Emmanuel Pereira da Cunha Silva.

CAMPINA GRANDE - PB

2021

PAULO JOSÉ BASTOS LEITÃO

AVALIAÇÃO DA TÉCNICA DO PREBAKING PARA REDUÇÃO DO COLD START EM SERVIÇOS FAAS PARA A RUNTIME DE PYTHON

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Thiago Emmanuel Pereira da Cunha Silva

Orientador – UASC/CEEI/UFCG

Professor Dr. Andrey Elisio Monteiro Brito

Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 06 de abril de 2021.

CAMPINA GRANDE - PB

ABSTRACT

Serverless platforms became a very attractive business model by abstracting the server's infrastructure from the developer and letting them focus on their application's functionalities. We adopted FaaS (function as a service), a type of serverless platform in which the client implements stateless functions that run on a cloud. However, due to this platform's billing model, which charges only for the time the function is actually running, it is not favorable for the provider to have idle functions taking up the cloud's resources. This model brings forth the cold start problem: a function's startup time is often very slow, which ends up driving clients away from FaaS. Previously, the Prebaking technique, based on creating snapshots from running functions and restoring them later, in order to minimize cold starts, has proven itself very effective for the Java runtime. In this work, we verified that for Python's runtime the gains are also significant: we managed to reduce a function's startup time by up to 1000%.

Avaliação da técnica do *Prebaking* para redução do *cold start* em serviços FaaS para a *runtime* de Python

Paulo José Bastos Leitão
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
paulo.leitao@ccc.ufcg.edu.br

Thiago Emmanuel Pereira
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
temmanuel@computacao.ufcg.edu.br

RESUMO

Plataformas *serverless* são um modelo de negócio bastante atrativo por abstrair do desenvolvedor a infraestrutura do servidor e permití-lo focar na lógica da sua aplicação. Adotamos aqui o FaaS (funções como serviço), que é uma modalidade de plataformas *serverless* em que o cliente implementa funções que são executadas na nuvem de forma *stateless*. Porém, devido ao modelo de cobrança dessas plataformas, em que se paga apenas pelo tempo em que a função de fato está sendo executada, não é vantajoso para o provedor que funções permaneçam ociosas ocupando os recursos da nuvem. Esse modelo traz consigo o problema do *cold start*: muitas vezes, o tempo de instanciação de uma função pode ser muito lento, o que acaba por afastar os clientes do FaaS. Previamente, a técnica do *Prebaking*, baseada em criar *snapshots* de funções em execução e restaurá-los posteriormente, de forma a reduzir o *cold start*, mostrou-se bastante eficaz para a *runtime* de Java. Neste trabalho, verificamos que para a *runtime* de Python a melhora também é significativa: conseguimos reduzir o tempo de instanciação de uma função em até 1000%.

PALAVRAS-CHAVE

Serverless, FaaS, *cold start*, análise de desempenho.

1. INTRODUÇÃO

Atualmente, plataformas *serverless* são muito atrativas para desenvolvedores e empresas no geral por permitirem que o foco do trabalho seja apenas no desenvolvimento da lógica de negócio do produto, e não na infraestrutura. O provedor do serviço é o responsável por gerenciar o provisionamento de recursos para a aplicação. Outra vantagem é o modelo de pagamento: o cliente paga apenas pelo tempo em que a aplicação está em execução, economizando assim custos como o de ter máquinas ociosas ligadas.

Uma modalidade de plataformas *serverless* é a FaaS (do inglês, funções como serviço). Neste padrão, o cliente implementa funções e estas são executadas na nuvem de forma *stateless*. Os serviços FaaS — como o AWS Lambda e o Firebase Cloud Functions — costumam suportar linguagens gerenciadas, como Java, Python e Go.

Porém, do ponto de vista do provedor, não vale a pena manter em memória todas as funções hospedadas, por conta da natureza da forma de pagamento. Dessa forma, não é interessante que funções ociosas permaneçam em memória, desperdiçando recursos.

Isso leva ao problema do *cold start*: muitas vezes, a função desejada não está disponível em memória, e o processo de instanciá-la pode levar muito tempo (nos nossos experimentos, observamos uma latência de até 600 milissegundos). Assim, o modelo FaaS pode se tornar desvantajoso para clientes que prezam por tempos de resposta rápidos.

Várias pesquisas se propõem a minimizar o *cold start*. Uma das técnicas é a de criação de *snapshots* — salvar o estado da função em um dado momento — a fim de reduzir o tempo de inicialização da *runtime* e instanciação da função.

Em especial, a técnica de *Prebaking* apresentou resultados bastante promissores para a *runtime* Java [1]. Neste trabalho, avaliamos o comportamento de três funções, quando usadas junto ao *Prebaking*: uma *NOOP*, que apenas retorna sucesso; um renderizador de *Markdown* para HTML e um *Image Resizer* que redimensiona imagens, na *runtime* de Python. Pudemos observar que os ganhos também são bastante significativos para a *runtime* de Python. Para todas as funções, o tempo de inicialização foi reduzido de em torno de 500 milissegundos para cerca de 50 milissegundos, o que significa uma melhora de 1000%, aproximadamente.

O restante deste trabalho é organizado da seguinte forma. A seção 2 entra mais a fundo nas plataformas FaaS, no problema do *cold start* e na nossa solução proposta, o *Prebaking*. A seção 3 descreve o *design* dos experimentos conduzidos para avaliar o impacto do *Prebaking* em cada função. A seção 4 apresenta os resultados obtidos. A seção 5 traz as considerações finais e possíveis limitações. Por fim, a seção 6 detalha os trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

Quando uma função hospedada em uma plataforma FaaS é invocada, caso já não esteja em memória, é necessária a instanciação da mesma. Esse processo costuma ser lento, causando o chamado *cold start*. Grandes contribuidores para o *cold start* são: i) o *overhead* das operações da plataforma, como a alocação de recursos para a função e provisionamento de endereços da rede; ii) o tempo de inicialização da *runtime* da função, que, dependendo da aplicação pode ser bem elevado [2]. Por exemplo, uma aplicação de aprendizado de máquina pode ter um modelo grande e que demore para ser alocado em memória.

Já que não temos controle sobre as operações da plataforma, resolvemos atacar o problema por meio da redução do tempo de instanciação da função. A técnica escolhida, por ser capaz de ser

aplicada para qualquer aplicação transparentemente, foi a de *checkpoint*/*restore*.

A técnica de *checkpoint*/*restore* consiste em dois passos: enquanto a aplicação está em em execução, faz-se o *checkpoint*, que consiste em salvar em disco o estado da aplicação. Depois, pode-se fazer o *restore*, que carrega o estado salvo em disco e retoma a execução do ponto em que foi salva.

Dessa forma, realizando o *checkpoint* da aplicação depois que a *runtime* esteja inicializada, as restaurações subsequentes já trarão a aplicação em um ponto em que ela esteja pronta para receber requisições, minimizando assim o *cold start*.

Uma implementação do *checkpoint*/*restore*, e a que é utilizada aqui, é o CRIU[1] (*Checkpoint/Restore In Userspace*). Essa implementação é bastante interessante porque funciona inteiramente no *userspace*, o que elimina a necessidade de acesso privilegiado à máquina.

2.1 Prebaking

Nesta subseção, será descrito o *design* e a implementação de um protótipo da técnica do *Prebaking*, que tem como objetivo principal reduzir os *cold starts* em plataformas *serverless*. Ademais, a técnica visa ter fácil integração com plataformas *serverless* já existentes; não impactar negativamente a performance da função após a inicialização; e não aumentar o custo de operação da plataforma *serverless*.

2.1.1 Design

A técnica do *Prebaking* reduz o tempo de inicialização da função por meio da restauração de *snapshots* de *runtimes* previamente inicializadas. Antes que uma função esteja pronta para atender a requisições, ela executa uma complexa (e muitas vezes lenta) sequência de passos. Esses passos incluem: criar um novo processo para hospedar a *runtime*, a inicialização de suas estruturas de dados e serviços auxiliares, e o carregamento do código da função. Presume-se que é mais rápido restaurar um *snapshot* do que executar novamente todos esses passos.

Métodos baseados em *checkpoint*/*restore* são amplamente utilizados em computação de alta performance para tolerar falhas em aplicações que rodam por muito tempo; quando acontece uma falha, a aplicação pode ser retomada a partir de *snapshots* gerados automaticamente, em vez de reiniciar do zero. Apesar de útil, a restauração de *checkpoints* pode impactar a performance de aplicações. Por um lado, o quão mais frequente a geração de *snapshots*, mais rápida a recuperação do estado logo antes da falha (porque é necessária menor computação). Por outro lado, já que a geração de *snapshots* compete por recursos computacionais, *snapshots* frequentes podem tornar a aplicação mais lenta. No melhor caso, quando não há falha, a geração de *snapshots* acaba sendo apenas um gasto a mais de recursos.

Diferente do caso da computação de alta performance, a técnica do *Prebaking* cria *snapshots* somente quando o usuário atualiza a versão da sua função. Dessa forma, a execução da função não é atrasada, já que essa atualização ocorre antes que a função seja chamada.

A plataforma restauraria o *snapshot* sempre que uma nova instância da função fosse criada. O mesmo *snapshot* pode ser usado para restaurar várias réplicas porque todas têm o mesmo estado no início da execução. Além disso, o *Prebaking* permite a criação de *snapshots* a qualquer momento da inicialização da função. Esta característica abre espaço para otimizar o processo de geração de *snapshots* de forma a minimizar o atraso de restauração. Por exemplo, uma abordagem que é válida para todas as *runtimes* é gerar os *snapshots* após o fim do processo de inicialização. Em vez de lidar com código específico para cada *runtime*, só é necessário esperar que a criação da função termine para gerar o *snapshot*.

Outra otimização possível é escolher com mais cuidado o momento da inicialização para gerar o *snapshot*. Isso requer um conhecimento mais aprofundado da *runtime*, no entanto. Apesar dos benefícios em potencial, lidar com as especificidades de cada *runtime* pode comprometer o objetivo de ter fácil integração com plataformas *serverless* já existentes.

2.1.2 Implementação

O primeiro passo para criar um *checkpoint* é ler a memória e o estado do processo desejado. A forma mais simples de fazer isso é modificar o programa para que ele próprio crie o *checkpoint*. Essa solução não condiz com os nossos objetivos, já que almejamos pela fácil integração com plataformas *serverless* já existentes e a solução demandaria a alteração de todo código submetido para a plataforma.

Um *checkpoint* completamente transparente, isto é, sem que o programa desejado tenha ciência da criação do *checkpoint*, é possível a nível do kernel. Neste nível, o procedimento da criação do *checkpoint* poderia acessar o espaço de endereçamento de qualquer processo. Infelizmente, apesar de possível, a abordagem a nível de kernel nunca foi amplamente adotada. Outra opção é usar soluções que sacrifiquem parte da transparência para manter-se a nível de usuário, que é o caso da *libckpt* [4]. Isso ainda não é ideal porque requer que a aplicação seja recompilada para incluir o código que cria o *checkpoint*.

Recentemente, o CRIU alcançou uma criação de *checkpoint* totalmente transparente a nível de usuário. Em vez de modificar a aplicação durante a compilação, o CRIU injeta o procedimento de criação de *checkpoint* enquanto a aplicação está em execução. Assim que o *checkpoint* termina de ser criado, o CRIU se remove do código, e a aplicação retoma sua execução normalmente.

Primeiro, o CRIU precisa congelar todas as *threads* do processo alvo, para que o estado não mude durante a criação do *checkpoint*. Após parar todas as *threads*, o CRIU precisa descobrir o que deve ser salvo de cada *thread*. Por exemplo, ele lê o arquivo ***/proc/$pid/pagemap*** para encontrar as áreas da memória que foram mapeadas. Depois, o CRIU injeta o procedimento (código parasita) responsável por executar a coleta dos dados utilizando a *system call ptrace*. Quando o código parasita começa a executar, ele se comunica com o processo do CRIU para saber o que coletar, lê o conteúdo do espaço de endereçamento do processo alvo, e envia os dados para o processo do CRIU. Por fim, o CRIU usa a *system call ptrace* para remover o código parasita e se desvincular do processo alvo, que retoma sua execução.

O processo de restauração é mais simples que o de criação. Durante a restauração, o processo do CRIU se transforma no processo do qual foi feito o *checkpoint*. O primeiro passo é ler os dados e restaurar o estado do processo. Depois, ele recria todos os *namespaces* e arquivos abertos. Enfim, a memória é remapeada.

[1] https://criu.org/

O CRIU consegue executar tanto o mecanismo de criação de *checkpoint* quanto o de restauração sem privilégios. Isso é possível devido à funcionalidade de *CAP_CHECKPOINT_RESTORE* [3]. Essa funcionalidade permite a execução de procedimentos como selecionar um *pid* específico ao clonar um novo processo e o acesso a arquivos de mapeamento de memória a nível de usuário.

3. METODOLOGIA

Realizamos um experimento a fim de determinar o impacto que o *Prebaking* tem no tempo de inicialização de uma função. Para tal, foram implementadas três funções: *NOOP*, renderizador de *Markdown* e *Image Resizer*. As três possuem tamanhos diferentes, para que os experimentos tenham uma maior abrangência.

A *NOOP* apenas retorna sucesso para todas as requisições. É uma função trivial, que oferece uma interferência mínima no uso de recursos. Assim, podemos verificar o impacto mínimo do método.

A *Markdown* converte uma *string* em *Markdown*[2] para uma página HTML. A *string* é enviada no corpo da requisição e é retornada a página HTML.

A *Image Resizer* adiciona outro fator de complexidade: durante a inicialização, é carregada uma imagem de 1MB[3] e, para cada requisição, a imagem é reduzida para 10% do seu tamanho original. Dessa forma, o tempo de *cold start* para esta função é maior que para as outras.

Comparamos os tempos de inicialização de 200 execuções de cada uma das funções utilizando o *Prebaking* e utilizando o método convencional de inicialização, baseado nas *system calls* de *fork* e *exec*. Como o foco é o tempo de inicialização, o experimento consiste apenas em um gerador de carga e na *runtime* da função. As plataformas FaaS possuem mais componentes, mas esses foram omitidos a fim de reduzir o ruído e permitir uma maior acurácia na medição dos dados.

Todos os experimentos foram realizados em uma máquina virtual com um processador quad-core Intel(R) Core(TM) i5-3470S 2.90GHz, com 8 GB de memória RAM que executava o Ubuntu 18.04.05 LTS e kernel do Linux 4.15.0-142-generic-x86_64. A versão do Python utilizada foi a 3.6. A *runtime* de Java utilizada foi a OpenJDK 1.8.0_292. O gerador de carga e a *runtime* da função foram reiniciados antes de cada execução.

4. RESULTADOS

Primeiro, iremos analisar se o objetivo principal do *Prebaking* foi cumprido: o *cold start* de funções FaaS realmente foi reduzido?

Na Figura 1, comparamos 200 execuções de cada função utilizando o método do *Prebaking* e o método convencional (*Vanilla*). Podemos ver que houve uma melhora significativa no tempo de inicialização das três funções. A função *NOOP* obteve uma melhora média de 1222%; o renderizador de *Markdown* obteve uma melhora média de 1310%; e o *Image Resizer* obteve uma melhora média de 790%.

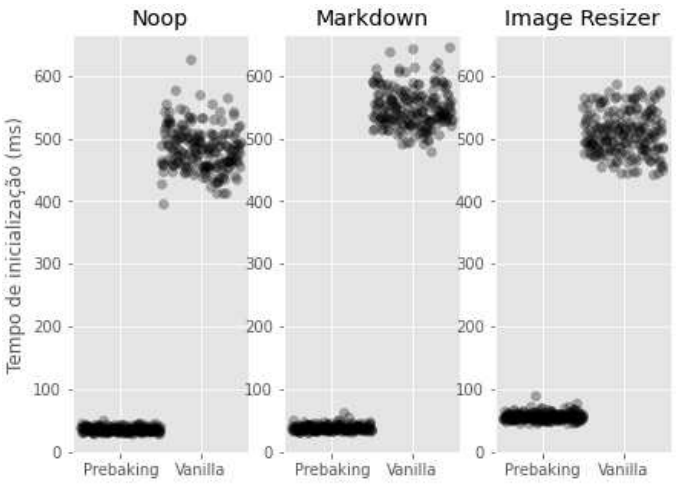

Figura 1 - Tempos de inicialização de cada uma das três funções, com a técnica do *Prebaking* e a *Vanilla*, na *runtime* de Python

Também precisamos verificar um objetivo secundário do *Prebaking*: houve ou não impacto na performance da função?

Na Figura 2, comparamos a função de distribuição acumulada empírica (ECDF) do tempo de resposta de 200 execuções de cada uma das funções após serem inicializadas pela técnica do *Prebaking* e a técnica *Vanilla*. No caso do *NOOP* e do renderizador de *Markdown*, como as curvas coincidem, podemos constatar que não há perda de performance. No caso do *Image Resizer*, o gráfico mostra que a técnica do *Prebaking* obteve melhores tempos de resposta.

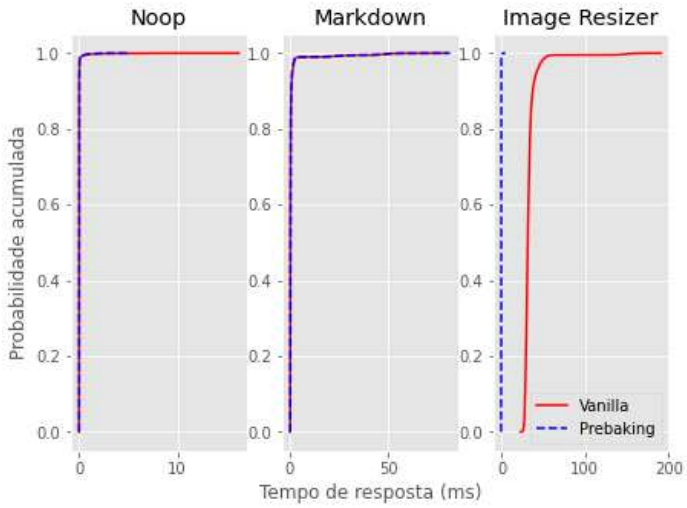

Figura 2 - ECDFs do tempo de resposta de cada uma das três funções, com a técnica do *Prebaking* e a *Vanilla*, na *runtime* de Python

É interessante também notar que os resultados obtidos da execução do experimento para a *runtime* de Java e Python foram bem diferentes. A Figura 3 ilustra os tempos de inicialização para cada uma das três funções, usando as técnicas do *Prebaking* e *Vanilla*, para a *runtime* de Java.

[2] https://github.com/PrincetonUniversity/openpiton

[3] https://i.imgur.com/BhlDUOR.jpg

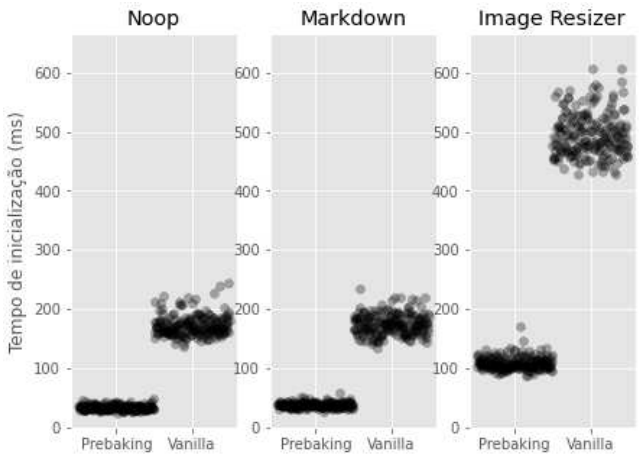

Figura 3 - Tempos de inicialização de cada uma das três funções, com a técnica do *Prebaking* e a *Vanilla*, na *runtime* de Java

No caso de Java, as funções mais simples — *NOOP* e o renderizador de *Markdown* — do método *Vanilla* possuíram menores tempos de inicialização. O *Image Resizer*, provavelmente por ter mais dependências, teve uma inicialização mais lenta. Já no caso de Python, todas as funções tiveram um tempo de inicialização maior, por volta dos 500 ms.

5. CONCLUSÃO

Plataformas *serverless*, em especial as da modalidade FaaS (funções como serviço) apresentam, notoriamente, o problema do *cold start*. Tipicamente, o tempo de resposta de uma função quando chamada pela primeira vez após algum tempo sem ser requisitada é elevado. Alguns fatores diferentes contribuem para esse problema, como o *overhead* das operações da plataforma [2]. A nossa abordagem para minimizar o tempo do *cold start* foi reduzir o tempo de inicialização da *runtime* da função, por meio da técnica do *Prebaking*.

O *Prebaking* se mostrou especialmente eficaz para a *runtime* de Python, obtendo reduções no tempo de inicialização das funções de cerca de 1000%.

Contudo, vale lembrar que os experimentos foram realizados em um ambiente desprovido de grande parte da infraestrutura de uma plataforma *serverless*. A fim de reduzir ruído experimental e focar no tempo de inicialização das funções, o modelo do experimento foi simplificado para possuir apenas um gerador de carga e a *runtime* da função sob teste.

6. TRABALHOS FUTUROS

Motivado pelos bons resultados que a técnica do *Prebaking* demonstrou para Java e Python, é interessante também avaliar o impacto da técnica em runtimes de outras linguagens. As grandes plataformas *serverless*, como a AWS Lambda e o Firebase Cloud Functions, também dão suporte a outras linguagens gerenciadas que não foram aqui contempladas, como Javascript, Go e Ruby.

Visto que o ambiente de experimentação foi bastante controlado, omitindo vários componentes de uma plataforma *serverless* comum, surge a necessidade de testes com uma simulação mais completa, ou até mesmo em um ambiente de produção, com funções reais. É possível que o ganho do *Prebaking* não seja tão significativo quando inserido em um contexto mais próximo da realidade.

Além disso, é também importante a integração da técnica do *Prebaking* com mais plataformas *serverless*. Previamente, foi implementada apenas a integração com a plataforma OpenFaaS. Apesar da OpenFaaS ser uma das plataformas *open source* mais populares, é interessante estender o suporte do *Prebaking* para a maior quantidade de plataformas possível.

7. AGRADECIMENTOS

Gostaria de agradecer primeiramente à Universidade Federal de Campina Grande, que me proporcionou o ensino de altíssima qualidade na área de Ciência da Computação. Agradeço também ao meu orientador, Thiago Emmanuel, por me auxiliar neste trabalho e também por me oferecer várias oportunidades durante a graduação.

Agradeço à minha mãe, Carmen, que sempre me apoiou incondicionalmente durante toda a minha vida. À minha namorada, Mariana, por estar sempre ao meu lado, iluminando meus momentos mais felizes e amenizando meus momentos mais difíceis. Por fim, a todos os meus amigos e colegas que compartilharam comigo essa jornada pela graduação, pois sem eles nada disso seria possível.

8. REFERÊNCIAS

[1] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking Functions to Warm the Serverless Cold Start. In Proceedings of the 21st International Middleware Conference (Middleware '20). Association for Computing Machinery, New York, NY, USA, 1–13. DOI: https://doi.org/10.1145/3423211.3425682

[2] Ping-Min Lin and Alex Glikson. 2019. Mitigating Cold Starts in Server-less Platforms: A Pool-Based Approach. CoRR abs/1903.12221 (2019). arXiv:1903.12221 http://arxiv.org/abs/1903.12221

[3] Linux Kernel. 2020. Linux merged patch for unprivileged checkpoint/restore. Retrieved March 15, 2022 from https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.

[4] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. 1995. Libckpt: Transparent Checkpointing under UNIX. In USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems, New Orleans, Louisiana, USA, January 16-20, 1995, Conference Proceedings. USENIX Association, 213–224. https://www.usenix.org/conference/usenix-1995-technical-conference/libckpt-transparent-checkpointing-under-unix