Mendelova univerzita v Brně
Provozně ekonomická fakulta

Správa politik MoM pro efektivní auto ballooning

Diplomová práce

Vedoucí práce:
Ing. Tomáš Koubek

Bc. Martin Pavlásek

Brno 2015

Chtěl bych poděkovat Ing. Tomáši Koubkovi za vedení a podporu v průběhu vzniku této práce. Dále pak Martinu Sivákovi, Adamu Litke a Jaroslavu Hennerovi za odborné konzultace, přátelům a mým rodičům za morální podporu.

Zde vlož zadání

Čestné prohlášení

Prohlašuji, že jsem tuto práci: **Správa politik MoM pro efektivní auto ballooning**
vypracoval samostatně a veškeré použité prameny a informace jsou uvedeny v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů, a v souladu s platnou *Směrnicí o zveřejňování vysokoškolských závěrečných prací.*
Jsem si vědom, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 Autorského zákona.
Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity o tom, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

V Brně dne 5. ledna 2015 .....

Abstract

Pavlásek, M. Policies management of MoM for effective auto ballooning. Diploma thesis. Brno 2015

This diploma thesis offers an overview of hypervisors and methods how they manage memory. Next part describes Memory overcommit Manager, design policies for more efficiet utilization of memory with memory ballooning. There are also description and implementation of newly created MoM Simulator. Purpose of this tool is to easily verificate the policies without running on a real hypervisor. In the latest part there are a description and a design of another new tool for managing constants. It allows applying policies on each virtual machine intedependently.

Keywords

KVM, virtualization, Memory overcommit Manager, MoM, libvirt, memory management, linux

Abstrakt

Pavlásek, M. Správa politik MoM pro efektivní auto ballooning. Diplomová práce. Brno 2015

Tato diplomová práce obsahuje přehled hypervisorů a jejich metod používaných pro správu operační paměti. V další části se zabývá nástrojem Memory overcommit Manager a návrhem jeho politik pro efektivnější využití operační paměti pomocí techniky memory ballooning. Dále obsahuje návrh a popis realizace nově vytvořené aplikace MoM Simulator. Ta slouží pro snadné ověření funkce politik bez nutnosti provozu na skutečném hypervisoru. V poslední části se nachází popis dále vytvořeného nástroje pro správu konstant umožňující uplatnění politik na jednotlivé virtuální stroje samostatně.

Klíčová slova

KVM, virtualizace, Memory overcommit Manger, MoM, libvirt, správa paměti, linux

Obsah

1 Úvod a cíl práce

1.1 Úvod

Termín *virtualizace* se v dnešní době již docela dobře zabydlel a pomalu se tak z něj stává módní slovo, tzv. buzzword. Možná je to i proto, že toto slovo nabývá mnoha podob a významů: počínaje *virtuální realitou*, *virtuální počítače* přes *virtuální měny* (např. bitcoiny) až po *Virtuální duet*[1]. Klasickým příkladem je ale běžný účet, který nabízí takřka každá banka. Objem prostředků, které na něm máte uloženy, je ve výsledku pouze číslo, které reprezentuje peníze. Penězi ale ve skutečnosti nejsou a stávají se jimi až když člověk stojí před bankomatem s platební kartou v ruce a bankovkami v druhé (Kusnetzky, 2009) (Watts, 2014).

V oblasti IT se tohoto termínu využívá nejčastěji ve smyslu *virtuálního počítače/stroje* či *virtuální privátní sítě (VPN)*. Správa a práce s virtuálními počítači totiž přináší své výhody. Vytvoření nového virtuálního stroje je výrazně rychlejší, než zprovoznění fyzického stroje. Odpadá fyzická návštěva serverovny, zapojování kabelů a potencionální starosti s nimi (uvolněná koncovka, krátký napájecí kabel). Vše se dá vyřešit pomocí konfiguračních nástrojů (v příkazové řádce `virsh`), často s grafickým uživatelským prostředím (například virt-manager, Oracle VM VirtualBox Manager, vSphere Web Access, Xen Orchestra, atd.). Jakmile už jej není zapotřebí, jednoduše se dá beze zbytku zahodit. Právě tato pružnost je velice lákavým atributem. Nese to však s sebou také další aspekty, jako například požadavek vysoké dostupnosti. Každý virtuální stroj je ve své podstatě pouze dalším běžícím procesem, který je svázán s operačním systémem, na kterém běží a je tak na něm závislý. Hostitelský systém však může být vytížen ostatními procesy až do takové míry, že může nastat i omezení provozu běžících virtuálních strojů (Berrangé, 2009) (VMware, 2014) (Xen Project, 2013).

Právě fakt, že jsou virtuální stroje spravovány hostitelským operačním systémem, s sebou přináší zajímavou možnost provozovat více virtuálních strojů, než může fyzická paměť hostitele pojmout. Tato technika se označuje jako *memory overcommitment*. Existuje několik cest, jak ji spravovat. Jednou z nich je využití nástroje *Memory overcommit manager* (dále jen MoM), který podle využití paměti hosta a virtuálních strojů umožňuje automaticky řídit za běhu velikost jim dostupné paměti. K tomuto účelu slouží pravidla, politiky, které podle chování celého systému tuto velikost dynamicky řídí.

Jejich návrh a následné úpravy ale nejsou snadno a rychle ověřitelné. Vyžaduje to nainstalovaný a konfigurovaný fyzický stroj, běžící virtuální stroje a především provoz na nich. To všechno však vyžaduje nezanedbatelný objem času a úsilí. Vhodně navrženými politikami a jejich nastavením tak lze efektivně využít dostupných prostředků.

[1] Virtuální duet je název písně z alba Nanoalbum hudební skupiny Tatabojs (Tatabojs, 2004, 9. stopa)

1.2 Cíl práce

Cílem práce je navrhnout pravidla pro MoM (Memory overcommit Manager), která zajistí efektivnější správu operační paměti virtuálním strojům, než tomu tak je u stávající implementace MoM v rámci projektu oVirt. oVirt je platformou pro správu virtuálních strojů založenou na KVM. Pro efektivní ověřování pravidel vznikne nástroj umožňující sledovat chování politik při definovaném vstupu. Dále bude vytvořen nástroj pro správu konstant pravidel, který umožňuje nastavit parametry politik každému virtuálnímu stroji samostatně. Pro naplnění cíle je nezbytné splnit následující dílčí body:

- Nastudovat architekturu MoM.
- Analyzovat/definovat problémové případy užití (tzv. scénáře).
- Upravit či navrhnout nové politiky, pro které výše definované scénáře pracuje MoM efektivněji.
- Vytvořit simulátor MoMu umožňující vyhodnotit funkci politik se zadaným scénářem.
- Podle potřeby implementovat chybějící části MoMu.
- Vytvořit nástroj pro správu politik umožňující individuální evaluaci politik pro samostatné virtuální stroje.
- Přizpůsobit politiky pro použití nástroje pro správu politik.

2 Virtualizace a správa paměti

Virtualizace je v dnešní době poměrně hodně rozvinutou oblastí. Souvisí s ní nové termíny i metody. Tato kapitola se bude zabývat především různými metodami správy paměti virtuálních strojů hypervisory.

2.1 Přehled terminologie

V této oblasti se běžně využívá řada termínů, některé z nich se vyskytují pouze v souvislosti s virtualizací. Následuje jejich přehled s vysvětlivkami:

Host, hostitel – fyzický počítač, na němž se spouští virtuální stoje (Hagen, 2008),

guest, VM (Virtual Machine), instance VM – virtuální počítač, dále v textu bude použito především termínu *guest*, protože se jedná o více rozdílné slovo, než tomu je například *hostitel a hosté* (Hagen, 2008),

hypervisor – označení pro software, někdy také *super operating system*, který spravuje virtuální stoje na hostiteli (Takemura, 2010),

emulace – běh spouštěné aplikace je prováděn v prostředí simulující vybraný procesor a periferie. Nejznámějším zástupcem je QEMU (Hagen, 2008),

paravirtualizace – pro běh virtuálních strojů se využívá hypervisora, ale vyžaduje pozměněný operační systém guesta (např. přístup k paměti se provádí skrze hypervisora, ne na fyzickém hardware). Běh aplikací na guestovi je rychlejší, než tomu tak je u emulace. Nejznámější je v této oblasti *Xen* (Hagen, 2008) (VMware, 2007),

plná virtualizace – podobně jako *paravirtualizace* využívá *hypervisora*. Rozdíl je ale v tom, že hypervisor v případě potřeby emuluje hardware, na kterém běží. Operační systém guesta může být nemodifikovaný (viz *paravirtualizace*) a nemá informaci o tom, že pracuje ve virtualizovaném prostředí (poskytuje například Microsoft Virtual Server nebo VMware ESXi Server) (Hagen, 2008) (Abels, 2005),

politiky, pravidla – sada podmínek a výrazů, které rozhodují např. o nové velikosti přidělené operační paměti virtuálního stroje, viz. MoM (kapitola 4),

balónování, memory ballooning – metoda dočasného omezení velikosti dostupné operační paměti guesta,

copy on write (CoW) – technika uchovávání dat, kdy se při operaci zápisu vytvoří kopie z původního obsahu a na ní se teprve realizuje samotný zápis. Tento termín se často užívá v souvislosi s přístupem ke sdíleným prostředkům (Symantec, 2014),

minor page fault – proces se pokouší přistoupit ke stránce v paměti, která ještě nebyla zinicializována a operační systém ji v tomto případě potřebuje nejdříve naplnit (RedHat, 2014)

memory overcommitment – umožňuje mít spuštěno více virtuálních strojů s celkovou velikostí paměti přesahující dostupnou paměť na hostovi (Kolovson, 2013),

swap – dalším běžně užívaným synonymem je *odkládací prostor*. Jedná se nejčastěji o vyhrazený oddíl na pevném disku (v operačních systémech typu linux může být swap i běžným souborem), ke kterému operační systém v případě potřeby přistupuje jako k operační paměti. Výhodou je snadné rozšíření jeho velikosti v případě potřeby, nicméně převažují negativa. Jeho největší nevýhodou je totiž přístupová doba, která je u plotnových pevných disků *výrazně delší*, než tomu je u operační paměti (VMware, 2009).

2.2 Správa paměti v linuxu

Stránka je základní jednotkou při manipulaci s pamětí (typicky při alokaci, odkládání na disk, atd.), její běžná velikost je řádově několik jednotek kB. V jádře (kernelu) se ale paměť alokuje jiným způsobem a toto alokování po stránkách se tam nepoužívá. V terminologii se vyskytují dva podobné výrazy:

Swapování – odložení kompletně celého procesu do prostoru swapu.

Stránkování – pracuje pouze se stránkami v paměti, které jsou velké pouze několik kB.

Linux však používá *stránkování*, protože je obvykle efektivnější, nicméně běžně se tyto termíny zaměňují a *swapováním* je ve skutečnosti myšleno *stránkování* (Jelínek, 2008).

OOM (Out Of Memory) killer je procedurou unitř jádra Linuxu (kernelu) a spouští se, pokud systému dojde fyzická paměť. Jejím úkolem je násilně ukončit tolik procesů, aby tento nedostatek vyřešil. Postupuje tak, že projde všechny procesy a vybere takový, který má nejhorší skóre označené jako *badness*. Tato hodnota se vypočítává podle více atributů procesu. Mezi nimi jsou:

- objem alokované paměti,
- počet potomků procesu (systémové volání `fork`),
- množství spotřebovaného procesorového a reálního času běhu, priority (nice).

Další možností, jak systém reaguje na nedostatek fyzické paměti, je uvolnění stránek v paměti z různých míst. Tyto stránky mohou být:

- odložitelné – například obsah systému souborů typu `tmpfs`, neaktivní a nezamčené anonymní stránky procesů,
- synchronizovatelné – cache a buffery, stránky souborů mapovaných do paměti,
- přímo uvolnitelné – např. cache adresářových položek.

Nelze uvolnit aktivní stránky, stránky zamčené v paměti, stránky přímo používané jádrem nebo procesy v režimu jádra (Jelínek, 2008).

Buffer cache – data z pevných disků bývají často čteny procesy několikrát během krátké doby (např. použití příkazu `ls`). Následující čtení ale už není nutné provádět znova, ale z cache pro to určenou. Tato metoda se označuje jako *disc buffering* a paměť, která se pro tento účel využívá *buffer cache*. Velikost této cache mění kernel podle potřeby – objemu využité paměti procesy. Zapisují se do ní bloky paměti (běžná velikost 1 kB), nikoli celé soubory. Požadavek na změnu těchto dat se provádí v buffer cache. K zápisu na skutečné cílové zařízení dochází až voláním příkazu `sync`, resp. je voláno programem `update`. `sync` způsobí okamžité vynucené uvolnění cache a tím zápisu dat na skutečné místo (Wirzenius, 2004).

3 Metody správy paměti hypervisory

V současné době existuje celá řada hypervisorů a s nimi také rozdílné přístupy jak spravovat paměť virtuálních strojů. Následující přehled vysvětluje metody správy paměti, které jsou používány v jednotlivých hypervizorech.

Ti se dají rozdělit do dvou velkých kategorií podle toho, kde (vzhledem k hardware) virtuální stroje běží. Rozdíl ukazuje Obrázek 1. Avšak toto dělení je v dnešní době spíše orientační. Není výjimkou, že některý z hypervisorů nespadá čistě do právě jedné z kategorií.

Příkladem je *KVM*, které sice vyžaduje pro svůj běh stávajícího hostitelského operačního systému (typ 2), nicméně téměř veškerý svůj čas tráví v tzv. *direct execution mode* (typické pro typ 1). Následující příklady jsou setříděny podle toho, kde jsou typicky uváděny (Pariseau, 2011).

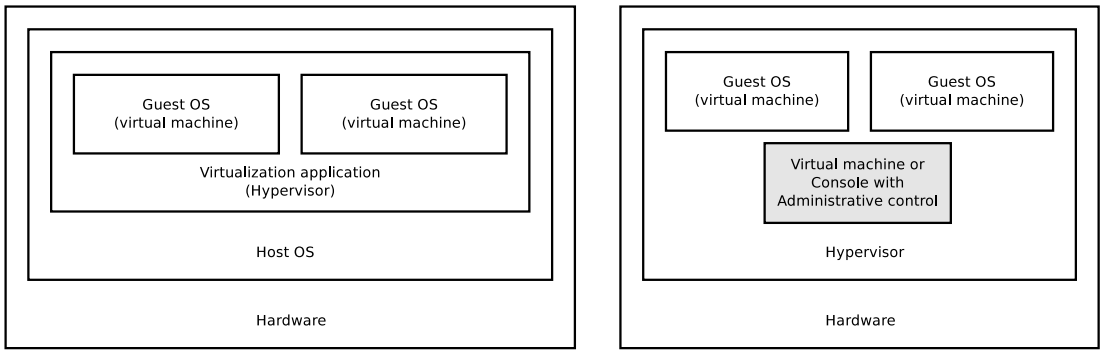
Typ 1 – označované také jako *bare-metal virtualization*. Virtuální stroje běží přímo na hardwaru bez dalších abstraktních vrstev. Je tak možné využít de facto 100 % prostředků výkonu serveru. Zástupci této kategorie jsou:

- Xen,
- KVM,
- VMware ESXi,
- Microsoft Virtual Server/Microsoft Hyper-V Server.

Typ 2 – virtuální stroje běží pod hypervisorem, který je provozován nad stávajícím operačním systémem, který běží přímo na hardwaru. Ten samozřejmě spotřebovává část systémových prostředků hosta. Mezi nejznámější hypervisory této kategorie patří:

- Oracle VM VirtualBox,
- Parallels Workstation,
- VMware Workstation,
- OpenVZ,
- Microsoft Virtual PC/Windows Virtual PC.

Hypervisoři podporují různé metody pro správu paměti virtuálních strojů – tento přehled je uveden v Tabulce 1.



Obrázek 1: Rozdíl mezi hypervisory typu 1 a 2 (Hagen, 2008)

3.1 VMware ESXi

Jedná se o hyperisora typu 1 (tzv. *bare-metal virtualization*), který běží přímo na hardware bez nutnosti běhu dalšího operačního systému pod sebou. V porovnání s ostatními hypervisory stejné skupiny spotřebuje pro svůj běh výrazně méně paměti, pouze cca 150 MB. Zajímavou vlastností je *vMotion*, která umožňuje živou migraci VM – přesun celého virtuálního stroje (paměť, CPU, disk) mezi různými hosty bez potřeby zastavení virtuálního stroje (VMware, 2013) (VMware, 2009).

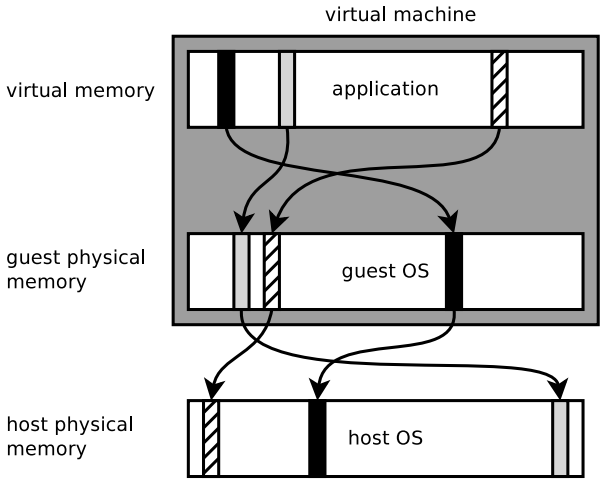
3.1.1 Transparent page sharing (TPS)

Metoda sdílených stránek v paměti operuje mimo paměť, kterou vidí guest. Hypervisor v intervalech hledá stejné stránky v paměti a u vhodných kandidátů provede jejich sloučení (sdílení). Algoritmus pro vyhledání stejné stránky funguje následovně:

- Pro stránku, ke které guest přistupuje, se vypočítá hash a ten se porovná v globální tabulce stránek jako klíč. Pokud se klíče shodují, až nyní se provede kompletní porovnání obsahů stránek, aby byla vyloučena kolize v hashovací funkci (jedná se totiž o náročnou operaci). Pokud je vše v pořádku, změní se typ záznamu na sdílenou v tabulce mapující fyzickou paměť guesta na fyzickou paměť hosta. Toto mapování se provádí mimo úroveň guesta a ani guest OS o tom není nijak informován.
- Zápis do sdílených stránek se provádí technikou Copy on write (CoW). V tomto případě jde o *minor page fault* a tím se vytvoří soukromá kopie stránky pro konkrétního guesta. Zápis do sdílené stránky zvyšuje overhead v porovnání se zápisem do nesdílené, který je způsoben prací navíc v page fault handler.
- Hypervisor prochází fyzickou paměť guesta v náhodných intervalech. V konfiguraci se dá specifikovat frekvence kompletního skenování paměti guesta. ESXi je schopné tyto intervaly nastavit i podle aktuálního použití. Pokud není velká

šance pro úspěšné nalezení sdílených stránek, sníží se i intervaly skenování paměti, a naopak.

(VMware, 2009)



Obrázek 2: Způsob mapování virtuální paměti guesta na fyzickou paměť hosta (Raffic, 2009)

3.1.2 Ballooning

Ballooning je metoda, kterou může host požádat guesta o uvolnění stránek fyzické paměti. To se děje skrze pseudozařízení na straně guesta, které je ale ovládáno pouze hostem. Ten požádá o nastavení změny balónu a tím dojde k naalokování stránek k tomuto zařízení. Takto „využité stránky“ nemohou být v žádném případě guestem odloženy na swap. Jakmile dojde k naalokování, zařízení to oznámí hostovi seznamem stránek a host tyto stránky uvolní ve své fyzické paměti.

Tento postup nevede k žádné ztrátě dat, protože tyto stránky nejsou skutečně využity guestem, takže ani tak není důvod je uchovávat. Tato operace se nazývá „inflating balloon“. Paměť se opačným způsobem dá guestovi zase vrátit zpět (jedná se o tzv. „deflating balloon“) – zmenšení velikosti balónu způsobí naalokování stránky ve fyzické paměti hosta.

Tato technika se typicky provádí v případě, že hostovi dochází fyzická paměť a je tak pod tlakem. Nafukování snižuje paměťovou zátěž hosta a zvyšuje využití paměti guesta. De facto se tak uvolňuje tlak hosta na úkor virtuálních strojů. V případě, že guest má mnoho volné paměti, není tímto omezením nijak dotčen. Pokud však je už sám pod tlakem, začne odkládat stránky na swap, aby uspokojil požadavek balónu na alokaci paměti. Takto zprostředkovaně hypervisor docílí odložení vhodných stránek guesta na swap (VMware, 2009).

3.1.3 Hypervisor swapping

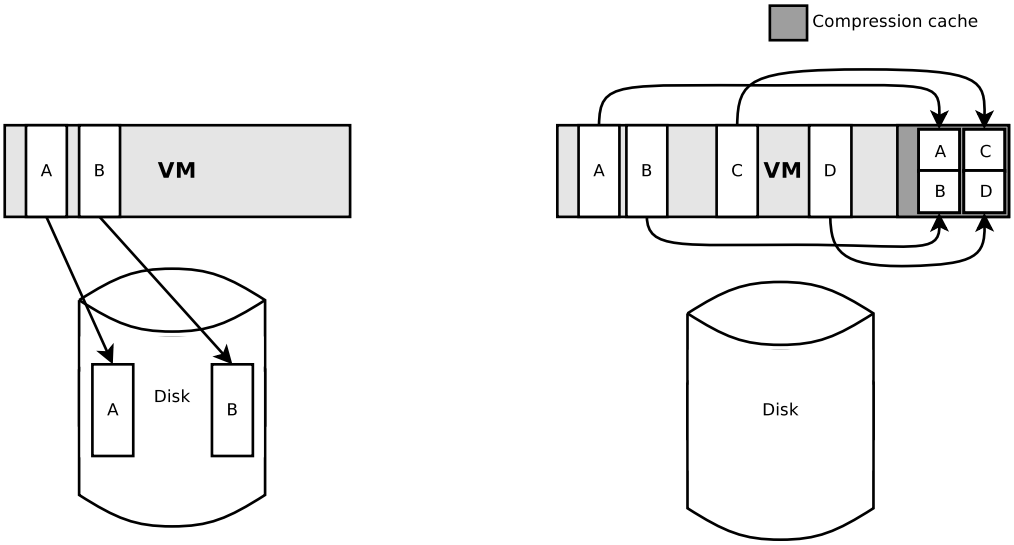
V krajním případě, kdy ani ballooning neposkytne hostovi dostatek paměti, vytváří se pro nově spuštěné virtuální stroje soukromý swap file. Hypervisor tak může rovnou odložit fyzickou paměť guesta do tohoto souboru a tím uvolnit fyzickou paměť hosta. Nevýhodou je však velká šance na degradaci výkonu guesta. Linux nikdy neodloží na swap takové stránky v paměti, které přímo ovlivňují výkon Kernelu. Problém je ale v tom, že hypervizor neví, co která stránka obsahuje (z pohledu guesta) (VMware, 2009).

Problém dvojitého stránkování – jedná se o nežádoucí chování guesta a hosta, kdy dochází k přesunu fyzické stránky guesta na swap jak guesta, tak i ze strany hosta. Klíčovým faktorem tohoto chování je fakt, že hypervisor nemá informaci o tom, které stránky guest přenáší na vlastní swap (VMware, 2009).

3.1.4 Page compression

Tato technika provádí kompresi stránek v paměti a uplatňuje se výhradně při *hypervisor swapping*. Stránky, které by jinak hypervisor byl nucen odložit na disk (na swap), uchovává v tzv. *compression cache*, která se nachází na fyzické paměti hosta (znázorněno na Obrázku 3).

Hlavním přínosem tohoto postupu je totiž výrazně rychlejší následný přístup k obsahu takové stránky než přímého čtení obsahu z pevného disku. Ne každá stránka je však takto komprimována. Postupuje se takto pouze v případě, že by se kompresí dalo ušetřit více než 50 % objemu dat. V opačném případě se tato stránka rovnou odloží na swap.



Obrázek 3: Schéma metody *page compression* použité ve VMware ESXi

V momentě, kdy některý z guestů chce přistoupit k obsahu stránky, která se na straně hosta nachází v *compression cache*, hypervisor provede její dekompresi, poskytne tak guestovi původní obsah a následně odstraní z této cache. Pro *compression cache* není vyhrazena žádná zvláštní oblast v paměti hosta a ani její velikost není konstantní (je nulová pokud k *hypervisor swapping* vůbec nedochází). Zato je definována její maximální velikost. Tento objem narůstá s potřebou odložit stránky paměti guesta na swap hosta. Jakmile se tato cache zcela zaplní, hypervisor odloží na swap takové stránky, které byly nejdéle nepoužité. Takové stránky se vždy nejdříve dekomprimují a následně odloží na swap.

Nastavení maximálního objemu této cache má významný vliv na výkon takto zasaženého virtuálního stroje. Příliš nízká hodnota vede k nízké efektivitě a způsobuje nepřiměřené využití swapu hosta i v případě, kdy má host ještě dostatek fyzické paměti. Opačný extrém má také negativní dopad na využití fyzické paměti hosta a v konečném důsledku také na virtuální stroje. Velký objem *compression cache* může vést k plýtvání pamětí hosta udržováním dlouho nepoužívaných stránek, a to pak způsobuje tlak na hosta. Doporučovaná horní hranice velikosti této cache je 10 % (VMware, 2009).

3.1.5 Shrnutí postupů výše

Rychlost odezvy na požadavek o znovuzískání stránek paměti (a také jejich počet) se liší podle použitého postupu. *Transparent Page Sharing* je závislé na intervalech skenování (`page scan rate`) a skladby stránek v paměti, u *balloningu* je nutná kooperace s guestem, zda vůbec a jak rychle vyhoví požadavku na uvolnění paměti. *Hypervisor swapping* je však metoda, kdy lze dosáhnout uvolnění garantovaného objemu paměti za určitý čas, avšak za cenu významného snížení výkonu a odezvy guesta (VMware, 2009).

3.2 Linux KVM

Zkratka *KVM* pochází z *Kernel based Virtual Machine* a jedná se modul do kernelu, který zajišťuje virtualizaci *typu 1* (jiné prameny uvádějí *typ 2*) (Pariseau, 2011). KVM do značné míry nahradil Xen jako dříve výchozí řešení virtualizace na většině linuxových systémech. Od verze Kernelu 2.6.20 se stal jeho standardní součástí, což mělo významný vliv na jeho oblíbenosti (Hagem, 2014).

3.2.1 Kernel Samepage Merging (KSM)

Tato technika umožňuje procesům sdílet stejné stránky v paměti. Zkratka *KSM* bývá někdy překládána jako *Kernel Shared Memory*. Nejdříve musí samotná aplikace zaregistrovat oblast paměti procesu KSM. Samotné sdílení pak probíhá následovně:

KSM prozkoumá povolenou oblast za účelem nalezení stejných stránek v paměti. Pokud uspěje a nalezne dvě a více takových stránek, označí je jako sdílené a nahradí je jedinou kopií, která je chráněna proti zápisu (změna by se tak dotkla

všech původních stránek, což není žádoucí). Na takto sdílené stránky je ale stále možné zapisovat, protože KSM v tomto případě vytvoří kopii stránky za použití metody *copy-on-write* a původní proces pak pracuje s touto kopií stránky.

Prohledávání (skenování) stránek je ale výpočetně náročná operace a proto je velice důležité nastavit parametry skenování rozumně. V případě, že bude KSM pracovat příliš agresivně, může takto zatěžovat i celé jedno vlákno na procesoru. Jedná se především o následující případy:

- skenování příliš mnoha stránek v rámci jednoho intervalu,
- intervaly pro skenování jsou příliš krátké (provádí se tak velmi často).

Nevýhodou KSM je ale procesorová náročnost. Největší efektivnosti dostahuje při provozu homogenních VM se stejným OS (IBM Corporation, 2012).

3.2.2 Memory balloning

Podobně jako VMware ESXi, i KVM podporuje *ballooning paměti*. Rozdíl je však v tom, že KVM poskytuje rozhraní, kterým lze měnit velikost balónu, ale samo autonomně jeho velikost neřídí. VMware toto řízení má již vestavěné v ESXi a lze jej pouze nakonfigurovat pomocí změny parametrů a limitů.

Balooning je tak spoluprací mezi hostem a guestem při správě paměti a dovoluje tak hostovi zužitkovat paměť původně vyhrazenou pro guesta. Změna velikosti balónu (konkrétně nafouknutí) se realizuje následovně:

- Hypervizor odešle požadavek guestovi na vrácení určitého objemu paměti. To provede skrze `virtio_balloon` zařízení v guestovi.
- `virtio_balloon` se pokusí uspokojit tento požadavek a to tak, že z pohledu guesta obsadí potřebný počet stránek v paměti.
- Tento seznam obsazených stránek pak vrátí hypervisoru jako odpověď na požadavek nafouknutí.

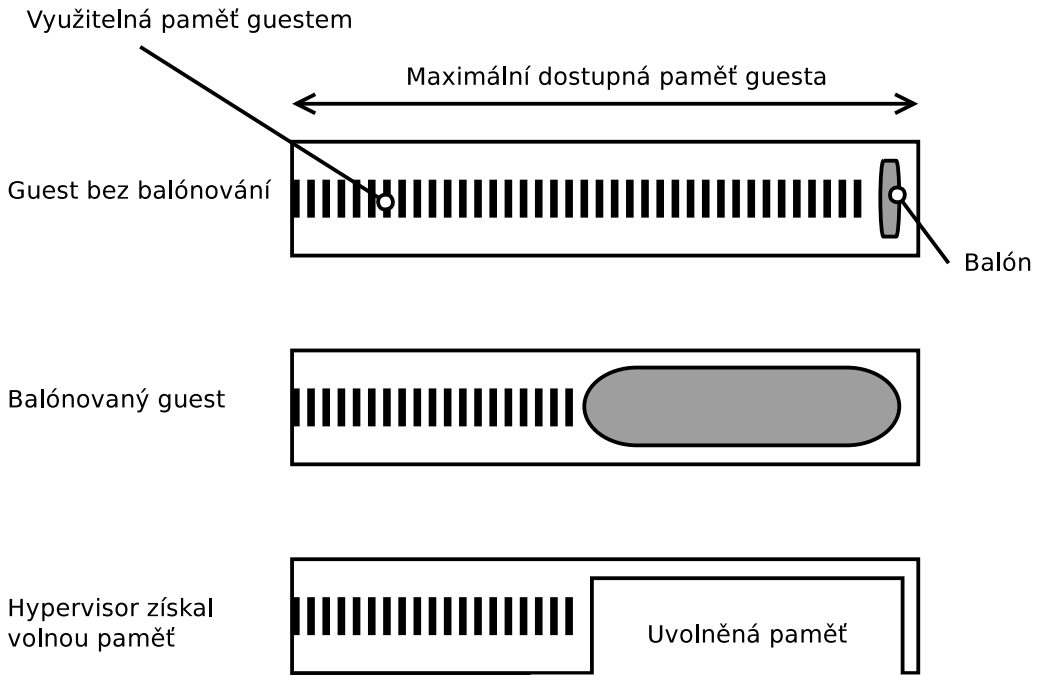
Může se stát, že `virtio_balloon` nebude schopen zcela vyhovět požadavku hosta, protože některá aplikace na guestovi si může „přišpendlit“ paměť. Vždy se ale snaží co nejvíce vyhovět, byť jen částečně. Vyfukování balónu probíhá velmi podobně pouze s tím rozdílem, že paměť v balónu se znovu zpřístupní guestovi k vlastní potřebě. Funkce balónu je znázorněna na Obrázku 4.

V porovnání s technikou *KSM* se operuje s balónem pouze pomocí definovaných příkazů (nafouknutí, vyfouknutí) a následného ověření, jak moc tato akce byla úspěšná. Jednoduše tak lze přímo ovlivnit míru uvolněné paměti. Hypervisor není povinen tuto uvolněnou paměť využít pro dalšího guesta. Tato technika má však také několik nevýhod:

- V guestovi je nutné načíst `virtio_balloon`. Oproti tomu KSM pracuje zcela bez vědomí guesta,

- balónování paměti guesta může mít negativní dopad na výkonnost virtuálního stroje (guest může začít swapovat),
- může způsobit selhání aplikací běžících na guestovi,
- zvýšení I/O blokových operací na guestu, protože operační systém na něm bude mít omezené možnosti využívat cache.

(IBM Corporation, 2012)



Obrázek 4: Princip *memory ballooning* (Jones, 2010) .

3.2.3 Swapping

Jedná se o další prostředek overcommitu paměti na guestech, je ale nejméně vhodnou možností. Swapování obecně je totiž znatelně méně výkonným prostředkem v porovnání s balónováním nebo sdílení stránek v paměti (KSM) (IBM Corporation, 2012).

3.3 Libvirt

Nejde o hypervisora v pravém slova smyslu, protože *libvirt* poskytuje jednotné API umožňující práci s virtuálními stroji (a nejen jimi) – zakrývá tak různorodost použitého skutečného hypervisora, tzv. *backendu* (viz. níže). Jedná se v však o natolik oblíbený *toolkit* (více než 30 plnohodnotných aplikací), že jej nelze opomenout. Aktuálně podporuje tyto hypervisory:

- KVM/QEMU Linux hypervisor,
- Xen hypervisor na Linux a Solaris hostovi,
- LXC Linux systém kontejnerů,
- OpenVZ Linux systém kontejnerů,
- User Mode Linux paravirtualizovaný kernel,
- VirtualBox,
- VMware ESXi a GSX,
- VMware Workstation a Player,
- Microsoft Hyper-V,
- IBM PowerVM,
- Parallels,
- Bhyve.

Libvirt se tak stal základním stavebním kamenem mnoha projektů. Jsou mezi nimi command-line utility (CLI[2]), webové aplikace, ale dokonce i nástroj, který využívá XMPP (Jabber) protokol. Níže se nalézá přehled vybraných projektů:

oVirt – pomocí webového rozhraní a REST API umožňuje spravovat velké množství virtuálních strojů napříč celými datacentry. Jednou z jeho součástí je i MoM.

Archipel – XMPP/Jabber protokol, podobně jako okamžitá (instant) textová komunikace dvou osob i tento nástroj reaguje na události (libvirtu tak i člověka) okamžitě po jejich proběhnutí. Proto odchozí zpráva od člověka typu „vypni se“ je normálním komunikačním prostředkem.

virsh – utilita do příkazové řádky (CLI) poskytující interaktivní shell a velké množství příkazů pro práci s virtualizovanými prostředky.

[2]Command Line Interface

guestfish – také CLI utilita s interaktivním shellem pro úpravu souborů (čtení, zápis, …) na pevném disku připojenému k virtuálnímu stroji.

virt-manager – aplikace s grafickým rozhraním pro správu malého počtu hostů (10-20) a na nich běžících virtuálních strojů.

OpenStack – označován jako *cloud operating system*, umožňuje spravovat virtuální stroje, bloková zařízení, virtualizovanou síť atd. pomocí webového rozhraní nebo REST[3] API (Openstack, 2014).

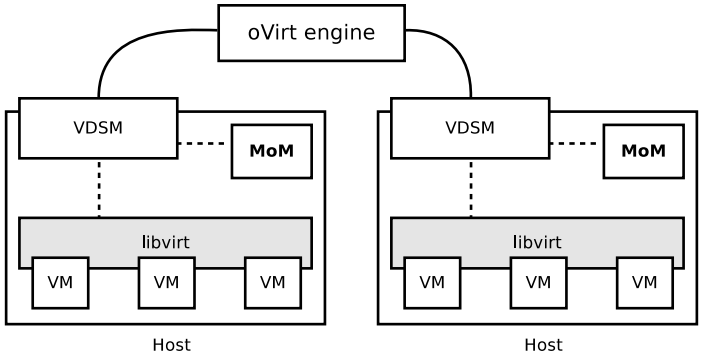
foreman – webová aplikace s cílem spravovat vše, co je potřeba při přidání nového virtuálního stroje k síti (DNS, HDCP, TFTP, …) a virtuální stroje samotné automatizovaně pomocí tzv. puppets.

(libvirt, 2014)

3.3.1 oVirt

oVirt je komunitou podporovaná verze produktu RedHat Enterprise Virtual Manager/Management. Je jednou z mnoha aplikací vybudovanou nad *libvirtem* (kapitola 3.3). Z toho plynou i další podporované vlastnosti i omezení. Pomocí *libvirt API* tak jsou definovány metody pro balónování a KSM (sdílení stránek). Obě metody již jsou zakomponovány do MoMu a je jednou z dílčích komponent oVirtu (Obrázek 5)

Centrálním orchestračním prvkem je *oVirt engine*. Ten spravuje veškeré zdroje (storage, hosté, atd.). S *engine* je možné komunikovat i pomocí REST API. Na každém hostovi běží *VDSM*[4], který přímo komunikuje s libvirtem, spravuje LVM atd. (RedHat, 2012) (Fediuck, 2012).



Obrázek 5: Integrace MoMu v oVirtu.

[3]REpresentational State Transfer(Richardson, 2007)
[4]VDSM – Virtual Desktop and Server Manager

3.4 Xen

Xen patří do skupiny hypervisorů realizující *paravirtualizaci*. Mezi běžnými virtuálními stroji je provozován ještě speciální instance – *domain 0*. Úlohou je spravovat dění na hypervisoru (např. spouštět/zastavovat ostatní VM). Jsou podporované oba druhy operačních systémů na guestech – modifikovaní i nemodifikovaní. Modifikovaný OS má informaci o tom, že je provozován na virtualizovaném prostředí a využívá upravené ovladače k dosažení vysokého výkonu. Je nejvýkonnější open-source virtualizačních technologií na systémech Linux. Je podporováno několik metod používaných pro správu paměti guestů, které lze nalézt v Tabulce 1 (Abels, 2005) (Hagem, 2014).

3.4.1 Transcendent Memory (tmem)

Jedná se o time-sharing metodu založenou na balónování a odstraňuje některé její slabiny. Jedná se o kolekci dostupné fyzické paměti hosta a API rozhraní, které k ní zprostředkovává přístup. Těchto kolekcí (tzv. *pools*) může existovat i několik a jsou poskytovány hypervisorem (*tmem host*). Guesti (*tmem clients*) mohou k této paměti přistupovat výhradně skrze dobře definované API a jsou na ně uplatňovány sady pravidel a omezení. Při vhodném použití *tmem API* mohou guesti využívat *tmem pool* jako rozšíření vlastní paměti, což má za následek snížení IO[5] zátěže a tím zlepšení výkonu.

tmem pool organizuje velké množství stránek v paměti a proto je pro přístup ke stránkám řešen pomocí seznamu hashovaných objektů. Objekty jsou organizovány pomocí datové struktury *radix tree* a nachází se v něm jako kořeny těchto stromů. Každý koncový uzel (leaf) stromu odkazuje přímo na *page descriptor*[6]. Ty jsou uloženy jako dva obousměrné seznamy LFU[7]. Jeden z nich se udržuje pro každý virtuální stroj samostatně a druhý globálně napříč všemi VM.

V případě nedostatku paměti (a závislosti na uplatňovaných pravidlech, politikách) se získávají *page descriptor*y od konce těchto seznamů (Magenheimer, 2009).

Tabulka 1: Podporované metody správy paměti hypervizorů (Banerjee, 2013)

Metoda \ Hypervizor	ESXi	Hyper-V	KVM	Xen
Memory sharing (TPS, KSM)	✓		✓	
Ballooning	✓	✓	✓	✓
Page compression	✓			
Hypervisor swap	✓	✓	✓	
Memory hot-add		✓		
Transcendent memory				✓

[5]IO (Input Output) operace, nejčastěji se užívá ve smyslu s přístupem na pevný disk
[6]odkaz, reference na stránku v paměti
[7]LFU (Least Frequently Used) – nejméně často nepoužívané

4 Memory overcommit Manager

MoM je nástroj napsaný v jazyce `python`, který (obecně) umožňuje řídit systémové prostředky (zdroje) virtuálních strojů. Jedná se tak o aplikaci zajišťující QoS[8], jímž autorem je Adam Litke. Momentálně je nejvíce propracována správa operační paměti guestů. Je podporován *memory ballooning* a *KSM*. Smyslem je rozložit tlak (v případě potřeby) nedostatku paměti z hosta na guesty. Host takto může přerozdělit dostupnou paměť efektivněji.

4.1 Stavba, struktura

MoM je složen z řady jednoúčelových malých modulů, které spolu kooperují a předávají si navzájem výsledky své práce. Vstupní data celého systému jsou získávána pomocí zásuvných modulů (pluginů). MoM tak není pevně svázán pouze pro řízení objemu dostupné operační pamětí virtuálních strojů. Lze tak snadno rozšířit rozsah působnosti nástroje i na další systémové zdroje, kterými mohou být například:

- procesorový čas – je možné řešit pomocí cgroups[9] (Torvalds, 2010),
- přenosová rychlost přístupu k blokovým zařízením (IO bandwidth),
- shaping síťového provozu.

Pro rychlý přehled o stavbě dobře poslouží následující Obrázek 6, kde jsou vyobrazeny pouze nejdůležitější části. Mnohem více detailů se nachází v Obrázku 7.

4.1.1 HypervisorInterface

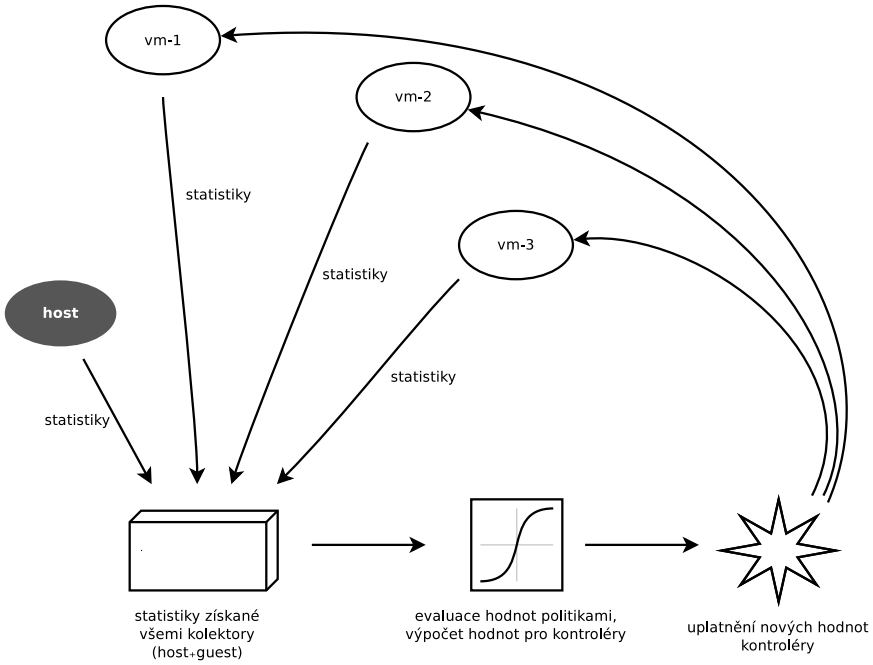
Jedná se o abstraktní objektovou třídu, jímž účelem je definovat rozhraní (definice obsažených metod), které MoM uvnitř používá. Tím, že se s hypervizory reálně komunikuje pomocí jejích API, jde tak vlastně o spojovací prvek mezi různými rozhraními (aktuálně jimi jsou `libvirt` a `vdsm`) spolu s vnitřní strukturou MoMu samotného.

4.1.2 Monitor

Monitor je objekt, jehož úkolem je shromáždit všechny dostupné statistiky z virtuálního stroje, který obsluhuje. S každým virtuálním strojem je svázán právě jeden Monitor. V MoMu se vyskytují dva druhy monitorů - `GuestMonitor` a `HostMonitor`.

[8] Quality of Service (RFC 2212, 2009)

[9] control groups - rozšíření do Kernelu linuxu umožňující regulaci, měření spotřebovaného objemu a oddělení systémových prostředků kterými jsou například procesor, paměť nebo také IO přístup k diskům (Menage, 2014)



Obrázek 6: Zjednodušené schéma práce MoMu

HostMonitor – již podle názvu provádí sběr statistik na hostiteli. Ty poskytují pluginy pro něj určené (samostatná sekce v konfiguračním souboru).

GuestMonitor – existuje právě jeden ke každému běžícímu virtuálnímu stroji. Jeho spuštění i zánik má na starost `GuestManager`, který udržuje množinu běžících instancí objektů `GuestMonitor`.

4.1.3 Collector

Kolektor obecně je objekt poskytující data o cílovém systému (platí pro hosta i guesty) v definované výstupní struktuře. Je dobrým zvykem pojmenovávat kolektory podle účelu a místa, například: `HostMemory`, `GuestMemory`, `GuestBalloon`. Data, která vrací, jsou asociativním polem a ne vždy musí obsahovat všechny klíče. Každý atribut, klíč, se v MoMu označuje jako *field*. Ty jsou dvojího druhu:

field – povinná položka, její neexistence je v monitoru ohlášena jako chyba. Je to kvůli pravidlům, politikám, které na ni závisí.

optionalField – již podle názvu se jedná o volitelnou položku. Pokud tento klíč chybí, jedná se také o korektní stav. Může se tak jednat o zachycení hodnoty do exportovaných dat, která však v politikách mohou i nemusí být vůbec použita.

4.1.4 Plot

Data získaná z kolektorů je možné pomocí tohoto modulu nechat exportovat do textových souborů a následně vizualizovat. Výstupní formát je variantou CSV[10], s tabulátorem použitý oddělovač a znakem # pro komentáře.

4.1.5 PolicyEngine

Tento objekt má na starosti parsování politik a nakonec především interpretaci spolu s daty dodanými jednotlivými kolektory. Obsahuje metody pro dynamickou práci se zdroji pravidel. Je proto možné s předpisem pravidel manipulovat za běhu (např. nahradit jednu část za jinou, nebo ji zcela odstranit). Jeho výstupem je množina nových hodnot pro jednotlivé guesty (a také hosta) a předá je všem kontrolérům k aplikování.

4.1.6 Controller

Kontrolér je výkonný akční prvek, který podle zadaných parametrů provede definovanou operaci. Typickým příkladem je: *změň velikost dostupné paměti u vm-4 na 4562 MB*. Vstupními daty jsou zpracovaná data z `PolicyEngine` a akci samotnou provádí skrze dodané rozhraní typu `hypervisorIterface`.

4.2 Popis funkce

Většina komponent MoMu běží v samostatných vláknech záhy po spuštění hlavního skriptu `momd`, přičemž vlákna `GuestMonitor` se vytváří a zanikají podle aktuálně běžících virtuálních strojů. Všechna vlákna (s výjimkou těch, která spravuje `GuestMonitor`) mají konfigurovatelné intervaly doby, kdy jsou aktivní (konfigurační direktivy končící na `-interval`).

Ve stručnosti lze funkci popsat takto: V časových intervalech se posbírají naměřené hodnoty guestů (objem aktuálně dostupné a využité paměti), podle politik (pravidel) se vypočítají nové hodnoty a ty se následně nazpět aplikují na guesty (např. změna maximální dostupné paměti na x MB).

Ve skutečnosti se toho ale děje výrazně více (viz. Obrázek 7):
V hlavní smyčce aplikace se v intervalech `main-loop-interval` ověřuje, zda jsou naživu všechna dílčí vlákna – `HostMonitor`, `GuestManager` a `PolicyEngine`. Smyslem je zajistit korektní ukončení celé aplikace, pokud se někde vyskytne závažný problém.

V intervalech definovaných v konstantě `guest-manager-interval` se provádí detekce aktuálně běžících virtuálních strojů. Nově zjištěným je v tomto okamžiku vytvořena vlastní instance `GuestMonitor`. Ta běží v novém samostatném vlákně

[10]Comma Separated Values – textový formát, který definuje datové řady na samostatných řádcích a datové sloupce jsou odděleny čárkou.

pro jejich obsluhu a zanikým guestům je jejich vlákno ukončeno (vypnutý guest nespotřebovává žádné systémové prostředky, takže není důvod se o něj dále starat).

Po uplynutí času definovaným jako `guest-monitor-manager` se probudí vlákno `GuestManager` a provede sběr dat ze všech aktuálně běžících virtuálních strojů skrze své monitory. Monitor projde všechny kolektory, dotáže se jich na své výstupy a ty následně uloží do souhrnného asociativního pole (dict). V tomto okamžiku se také volá modul `Plot` (pokud je povolen), který zajistí export těchto hodnot pro další zpracování (například vizualizace). Také `HostMonitor` seskupí data poskytnutá kolektory, stejně tak zajistí jejich případný export.

V tuto chvíli již má MoM všechna naměřená data k dispozici na jednom místě – mimo monitory atd. Data jsou dále zabalená do objektu `Entity`, která poskytuje nad daty pomocné funkce (výpočet průměrné hodnoty, nastavení požadované hodnoty kontroléru, …), které jsou dostupné při jejich vyhodnocování politikami.

Po uplynutí doby `policy-engine-interval` jsou entity přivedeny na vstup `PolicyEngine`, kde probíhá jejich skutečná evaluace. Výstupem jsou množiny nových hodnot pro kontroléry. Ty už reálně aplikují požadované změny – typicky se tak jedná nakonec o API volání hypervizora na konkrétní akci (změna velikosti dostupné paměti virtuálního stroje).

4.3 Politiky a jejich zpracování

Ústředním prvkem celého MoMu jsou politiky. Veškeré ostatní části jen zabezpečují ostatní práci (sběr dat, manipulace s balónem, kontrola běžících guestů, …). Právě těmi jsou řízeny systémové zdroje. Jejich vhodným návrhem tak lze zefektivnit využití těchto zdrojů nebo jimi plýtvat (při nevhodné konfiguraci). Jediným objektem, který s nimi pracuje, je `PolicyEngine`, který lze nalézt na Obrázku 7.

4.3.1 Definice politik

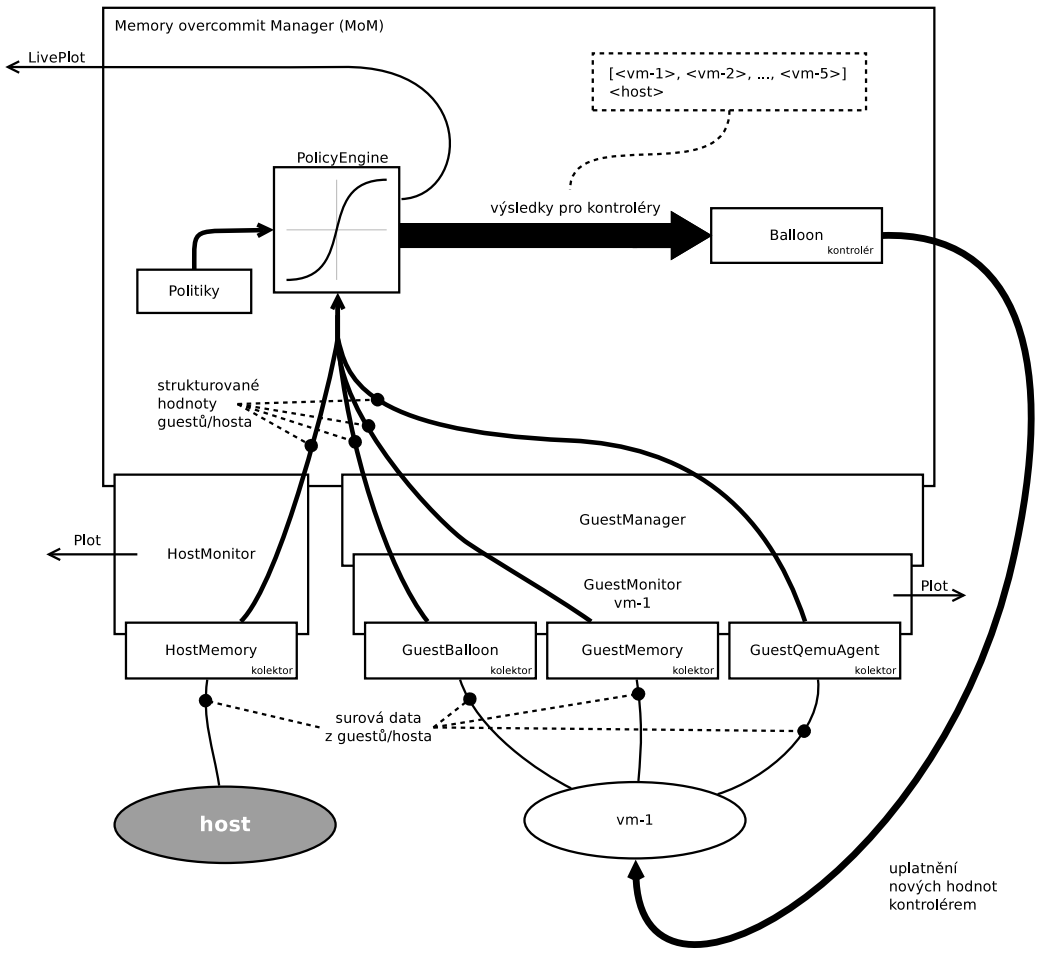
Pravidla řídící MoM jsou psaná v dialektu jazyka Lisp. Tento jazyk je charakteristický uzavíráním veškerých výrazů do kulatých závorek. Výrazy se zapisují v prefixové notaci, bloky více výrazů se uzavírají mezi složené závorky. Takto například vypadá větvení:

```
(if (< guest_memory_current guest_memory_limit_min)
  {
    # Guest je příliš utlačován, aktuálně má méně paměti,
    # než je nastavené minimum, bude potřeba mu přidat paměť.
    (defvar new_mem (+ guest_memory_current const_mem_increment))

    # Nula zde znamená ektivalent 'NOP' v assembleru či 'pass' v pythonu.
    0
  }
  {
```

```
    # Objem aktuálně přidělené paměti guesta se pohybuje
    # v nastavených mezích.

    (debug 'Nastavuji pamet na ' guest_memory_limit_min)
    (guest.Controller guest_memory_limit_min)
  }
)
```



Obrázek 7: Podrobné schéma struktury komponent MoMu.

5 Metodika

Cílem práce je navrhnout změny v politikách MoMu, které zajistí efektivnější správu paměti. Politiky, které řídí práci MoMu, jsou napsány v dialektu jazyka `Lisp`. Přehled o konkrétních dostupných funkcí a operátorů popisuje následující podkapitola 5.1.

Navržené změny v politikách ale není možné rychle vyzkoušet a tím snadno ověřit jejich funkci. Jejich vývoj tak je zdlouhavým procesem. Pro účely ověření funkce politik bude vytvořen MoM Simulator. Jedná se o rozšíření MoMu, které bude (stejně jako MoM) naprogramováno v jazyce `python`. Zcela tak odpadne nutnost změny hardwarové i softwarové konfigurace a ušetřený čas tak lze věnovat práci se samotnými politikami. Spojení s MoMem bude realizováno formou nové implementace rozhraní `hypervisorInterface`. Bude tak sloužit jako speciální driver pro MoM, který bude předkládat externě dodané hodnoty využité paměti (a další údaje) jako výstup API volání libvirtu.

Samotné ověření politik bude provedeno skrze několik různých scénářů. Nejdříve budou získány (referenční) výsledky scénářů při použití stávajících politik. Výsledkem budou průběhy objemů využité/volné/dostupné paměti, ideálně reprezentovány ve formě grafů. Následně budou stejné scénáře použity na obě změny v pravidlech – samostatně. Tímto způsobem tak bude možné dobře sledovat projevy dílčích změn.

Kritériem je především průběh využití paměti hosta – je žádoucí, aby nevyužíval *swap*, protože tato činnost má výrazný negativní vliv na výkon celého operačního systému hosta a tím i guestů. Pokud to je ale nezbytné, požadujeme, aby se tak dělo co nejkratší možnou dobu. Další sledovanou charakteristikou je poměr volné a využité paměti hosta, při kterém už dochází k balónování. Sledujeme, zda není prováděno balónování příliš brzy, nebo příliš pozdě.

V poslední části bude vytvořen nástroj, který bude v konečném důsledku umožňovat aplikovat části politik na jednotlivá VM individuálně. Nástroj bude dedikovaný vzhledem k MoMu a bude tak pracovat samostatně. Také bude implementován v jazyce `python`.

5.1 Dostupné funkce a operátory v politikách

Implementace jazyka Lisp v MoMu obsahuje sadu základních funkcí, které je možné použít pro výpočty výrazů použitelné při řízení kontrolérů. Všechny proměnné jsou definovány na globální úrovni.

def Příkaz pro definici vlastní funkce s parametry. Později je možné funkci i předefinovat. Jednoduše se přepíše, avšak na tuto skutečnost není nikde upozorněno. Syntaxe:

```
(
    def my_function (param1[, param2, ...])
    {
        # Body of my_function
    }
)
```

defvar Příkaz umožňující vytvoření proměnné a přiřadit do ní hodnotu. Změna hodnoty se provádí příkazem `set`. Syntaxe:

```
(
    defvar my_variable 0.95
)
```

set Příkaz pro přiřazení nové hodnoty do již dříve existující proměnné. Syntaxe:

```
(
    set my_variable 3.14
)
```

abs Matematická funkce, která vrací absolutní hodnotu svého parametru. Syntaxe:

```
(
    abs (-123)
)
```

debug Pomocná funkce, která vypíše do logu hodnotu proměnné udanou jako parametr (vypisuje ji pouze při `log-level` nastavené na `debug`). Pro běžný provoz nemá příliš smysl, avšak při návrhu pravidel a jejich ladění je velice užitečná. Syntaxe:

```
(
    debug (my_value)
)
```

with Ekvivalent příkazu *foreach* známý z jiných jazyků, umožňující iterovat nad polem a zpracovat každou položku samostatně. Syntaxe:

```
(
    with array item (
        # some work with item
    )
)
```

min Funkce vrací hodnotu toho parametru, který má nejmenší hodnotu. Funkce přijímá dva a více parametrů. Syntaxe:

```
(
    min (value1, value2 [, ...])
)
```

max Funkce vrací hodnotu toho parametru, který má největší hodnotu. Funkce přijímá dva a více parametrů. Syntaxe:

```
(
    max (value_1, value_2 [, ...])
)
```

and Logický operátor součinu. Syntaxe:

```
(
    and expr_1 expr_2
)
```

or Logický operátor součtu. Syntaxe:

```
(
    or expr_1 expr_2
)
```

StatAvg Jedná se o funkci, kterou poskytuje objekt reprezentující guesta (vč. hosta), použije posledních několik hodnot (nejvýše `sample-history-length`) z uvedené položky (field) kolektoru a vypočítá hodnotu arimetmického průměru, kterou vrátí. Syntaxe:

```
(
    guest.StatAvg 'field'
)
```

StatStdDeviation Funkce, kterou poskytuje objekt reprezentující guesta (vč. hosta) a použije posledních několik hodnot (nejvýše `sample-history-length`) z uvedené položky (field) kolektoru pro vypočítání standardní odchylky hodnot. Syntaxe:

```
(
    guest.StatStdDeviation 'field'
)
```

Control Jedná se o funkci, kterou poskytuje objekt reprezentující guesta (vč. hosta) a nastaví novou hodnotu pro kontrolér `controller_name`. Vícenásobným voláním dochází k přepisu, takže se uplatní pouze poslední použitá hodnota. Syntaxe:

```
(
    guest.Control 'controller_name' value
)
```

GetControl Jedná se o funkci, kterou poskytuje objekt reprezentující guesta (vč. hosta) a vrací aktuální hodnotu kontroléru. Syntaxe:

```
(
    guest.GetControl 'controller_name'
)
```

UpdateStatVal Jedná se o funkci, kterou poskytuje objekt reprezentující guesta (vč. hosta) a změní poslední (tzn. aktuální) hodnoty položky (field). Tato metoda se dá použít pro statistické funkce množin čísel, která nepochází z některého kolektoru, ale jsou definovány až v průběhu evaluace politik. Interně vzato, pokud pole se statistikami pro `field` dosud neexistuje, dojde k jeho automatickému vytvoření. Syntaxe:

```
(
    guest.UpdateStatVal 'field' new_value
)
```

SetVar Funkce uloží hodnotu `value` do proměnné `variable_name` (první přiřazení i změna hodnoty se provádí stejnou metodou). Zásadní rozdíl oproti použití metod `(defvar ...)` a `(set ...)` je v délce zachování hodnoty. Proměnná definovaná pomocí `SetVar` bude přístupná i v dalším zpracování politik skrze `PolicyEngine`, což pro proměnné vytvořené pomocí `defvar` neplatí. Jejich existence skončí s posledním výrazem politik. Syntaxe:

```
(
    SetVar 'variable_name' value
)
```

GetVar Funkce vrací hodnotu proměnné `variable_name`, která byla definována pomocí `SetVar`. Syntaxe:

```
(
    GetVar 'variable_name'
)
```

– Znaménko mínus lze použít kromě výpočtu rozdílu také jako unární operátor. Syntaxe:

```
(
    - expr_1
)
```

Další operátory Lze použít všechny relační operátory `>`, `>=`, `<`, `<=`, `==`, `=!` a matematické `+`, `-`, `*`, `/` Syntaxe:

```
(
    operator expr_1 expr_2
)
```

6 MoM Simulator

MoM komunikuje se svým okolím skrze zvolený ovladač (implementaci rozhraní objektu `hypervisorInterface`). Aktuálně dostupné jsou `libvirt` a `vdsm`, které zajišťují konverzi mezi různými API. Bylo vytvořeno další rozhraní s názvem `fakeInterface` poskytující hodnoty z externího souboru namísto API volání skutečného hypervisora. Takto je možné sledovat chování MoMu i v situacích, kdy nemáme k dispozici potřebnou hardwarovou konfiguraci, nebo si rychle ověřit změnu v politikách. Vznikl také samostatný nástroj pro snazší generování vstupních dat a další pro vizualizaci průběžných i konečných výsledků simulace. Aktuálně nejsou podporovány další používané techniky v MoMu, mezi které patří KSM (Kernel Same page Merging).

Vlastnosti

- podpora memory ballooning,
- podpora startu virtuálního stroje s již dříve omezenou dostupnou pamětí (ekvivalentní atributu `currentMemory` v XML definici domény v libvirtu),
- podpora virtuálního stroje, který nelze balónovat (guest nepodporuje ovladač pro memory ballooning),
- automatické ukončení MoMu v případě chybějících vzorků dat.

6.1 Popis funkce

Simulátor není samostatnou aplikací v pravém slova smyslu, jedná se víc o doplněk do MoMu, který již existuje a rozšiřuje tak jeho možnosti. Proto je jeho funkce závislá na pořadí volání metod, které jsou deklarovány v rodičovské třídě `HypervisortInterface`.

Průběh volání lze popsat následovně: Celou kaskádu získávání dat z kolektorů začíná `GuestManager` voláním metody `getVmList`. Tato metoda má za úkol vrátit pole s identifikátory aktuálně běžících virtuálních strojů (jedná se o běžné řetězce znaků). Na tomto místě se z tohoto důvodu provádí inkrementace ukazatele (indexu, počítadla) pro identifikaci, které vzorky v řadě jsou považovány za aktuální. Zdroj dat obsahuje (měl by obsahovat) v každém stavu hodnotu využité paměti všech guestů v `kB`. V případě, že je tato hodnota rovna `-1`, simulátor interpretuje tuto hodnotu za vypnutý stav a ve výsledném seznamu virtuálních strojů se proto takoví guesti nevyskytují.

Následuje řada volání pro získání nového vzorku statistik a dalších informací pro každého guesta samostatně skrze kolektory. `getVmInfo` se používá pro předání dalších obecných informací o virtuálním stroji (typicky jeho název, protože metoda přijímá identifikaci VM pomocí UUID[11]).

[11] Universally Unique IDentifier (RFC 4122, 2005)

`getVmMemoryStats` poskytuje informaci o nevyužité paměti. Jedná se o rozdíl `balloon_cur` (tuto hodnotu poskytuje `GuestBalloon` kolektor) a aktuálního vzorku využité paměti. `fakeHypervisor` rozšiřuje výstup této metody o položku `_mem_used`, která není určena pro použití v politikách (proto to podtržítko jako nepsaná konvence), ale poskytuje lepší přehled o zatížení systému ve výsledném grafu.

Pomocí metody `getVmBalloonInfo` je možné získat maximální a aktuálně dostupnou velikost paměti. Pokud guest nepodporuje ballooning, jsou si obě hodnoty rovny. `getHostMemoryStats` vrací stav hosta popsaného třemi hodnotami:

- `mem_available`,
- `mem_free`,
- a doplňkovou metainformací `_mem_used`, která není určena pro použití v politikách, slouží pouze pro následnou vizualizaci.

Využitá paměť je sumou aktuálně přidělené paměti všech virtuálních strojů `mem_free` a aktuálním vzorkem pro hosta.

FakeHostMemory je plugin vycházející z `HostMemory`. Ten pro sběr hodnot volá příkazy rovnou a nabízí tak jejich strukturovanou podobu. Z důvodu potřeby podstrkávat i tyto hodnoty ze strany `fakeInterface` vznila tato upravená kopie, která pro tento účel využívá metod rozhraní `fakeInterface`.

6.2 Konfigurace

MoM Simulátor rozšiřuje původní konfigurační soubor o několik dalších voleb, které přímo souvisí se simulátorem.

```
[simulator]
source-file: mom/scenario.csv

[liveplot]
fields: balloon_cur, mem_unused, mem_free, _mem_used, mem_available
export-samples: plot.json
```

V sekci `simulator` se nastavuje cesta ke vstupním datům k samotné simulaci. Část `liveplot` ovlivňuje výstup dat pro pozdější vizualizaci. `fields` je čárkou oddělený seznam fields, které jsou získány z kolektorů a budou exportovány – funguje tak jako filtr. Volba `export-samples` umožňuje nastavit cestu k souboru, kde `PolicyEngine` před zpracováním hodnot z kolektorů provede export dat ve formátu JSON.

6.3 Vstupní formát

Vstupním formátem pro simulátor je textový soubor CSV. Jako oddělovač je použit znak `,` (čárka). Pokud je na začátku řádku znak `#`, považuje se celý řádek jako komentář a ignoruje se. Všechna dále uvedená čísla představují objem paměti v `kB`. Každý řádek reprezentuje jednoho guesta. Pouze první z nich je ale chápán jako host.

host – první číslo je maximální dostupná paměť, všechna následující čísla představují aktuálně využitou paměť.

virtuální stroje – jsou popsány více parametry, konkrétně v tomto pořadí:

- maximální dostupná paměť, pokud je před číslem uveden znak `c` (od slova `constant`), simulátor bude ignorovat pokusy o změnu dostupné paměti, což odpovídá chování guesta, který nepodporuje memory ballooning,
- aktuálně přidělená paměť (= velikost balónu), u guestů, kteří nepodporují ballooning se tato hodnota ignoruje a použije se místo ní maximální dostupná paměť,
- libovolná hodnota, která se bude ignorovat (viz. poznámka níže) a
- všechny následující hodnoty představují aktuálně využitou paměť.

Ukázka vstupních dat:

```
# Zkoušíme hosta s 64GB pamětí, ale 42GB je zabráno i jiným způsobem,
# než jsou pouze virtuální stroje.
# Dva virtuální stroje (3GB a 8GB), oba startují až ve 2. kroku simulátoru,
# třetí z nich nelze balónovat (není podoporován memory ballooning).
64000000,         42000000,42000000,42000000
 3000000, 3000000,      -1,      -1, 2000000
 8000000, 8000000,      -1,      -1, 2000000
c3000000, 3000000,      -1,      -1, 2000000
```

Poznámka k ignorovanému vzorku je vhodné u všech guestů použít jako první hodnotu vzorku využité paměti *libovolnou hodnotu*, která se bude ignorovat. Toto doporučení vychází z aktuální implementace MoMu a simulátoru, kdy se nejprve spouští vlákno `HostMonitor` a až teprve následně `GuestManager`. `HostMonitor` záhy po své inicializaci provádí první sběr dat z kolektorů a tak se poprvé pomocí `fakeHypervisor` táže na hodnoty vzorků ze zdrojového souboru pro simulátor žížal. Aktuální vzorky paměti guestů však nikdo nepřečte, protože patřičné vlákno, `GuestManager`, dosud ještě neběží. V následujícím intervalu (v konfiguraci to je hodnota konstanty `guest-monitor-interval`) je již vše zinicializováno a čtení hodnot vzorků probíhá normálním způsobem.

6.4 Generátor scénářů

Manuální příprava scénáře se může velmi brzy stát pracnou činností s vysokou pravděpodobností chyby (orientovat se mezi velkým množstvím čísel není snadný úkol). Proto byl vytvořen tzv. *generátor scénářů*, který umožňuje produkovat hodnoty pro MoM Simulátor programově, přímo v prostředí Pythonu. Použití tohoto nástroje pro simulaci ale není nezbytné. Protože se jedná o běžný textový formát, lze tato data generovat i jiným způsobem (např. pomocí OpenOffice Calc nebo ručně). Tento generátor ale poskytuje některé praktické vlastnosti:

- poskytuje abstraktní vrstvu nad výstupem,
  - zajišťuje dodržení formátu podporovaný simulátorem,
  - produkuje lidmi dobře čitelný výstup (zarovnává čísla na pevnou šířku), takže vzhledově jde o přehlednou tabulku, kde každý sloupec představuje jednu množinu hodnot pro daný stav simulátoru,
  - podpora vložení bloku s komentářem, kde lze slovně popsat účel a průběh scénáře,
  - přímý export do zadaného souboru nebo na standardní výstup, pokud soubor není explicitně zadán,
- vypisuje upozornění, pokud některé akce vedou k nevalidnímu výstupu (například záporné využití paměti mimo speciální konstantu `-1` atd.),
- jedná se o objektovou třídu v Pythonu, takže je možné využívat veškeré možnosti, které tento jazyk nabízí.

6.4.1 Dostupné pomocné metody

Objekty reprezentující hosta i guesty jsou potomky objektu `GuestBase` (proto v následujícím výčtu bude sice užíváno termínu `guest`, ale bude to platit i pro hosta), která poskytuje následující praktické metody:

start(amount) spustí guesta se zadaným množstvím obsazené paměti (je aliasem pro `set`),

stop() nastaví využitou paměť na speciální hodnotu `-1`, která reprezentuje vypnutý stav guesta,

no_change() použije předchozí hodnotu využité paměti jako hodnotu aktuálního vzorku,

set(amount) vytvoří nový vzorek s využitou paměť o `amount` MB,

rand_mean_as_curr() uloží aktuální stav využití paměti jako střední hodnotu pro následující volání náhodného generátoru. Tato hodnota zůstává zapamatována až do doby dalšího zavolání stejné metody,

random_norm(mean, deviation) vygeneruje pseudonáhodné číslo s Gaussovým rozložením o parametrech *mean* (střední hodnota) a *deviation* (rozptyl) a použije tuto hodnotu jako nový vzorek. Pokud je ale parametrem *mean* předáno `None`, použije se hodnota uložená posledním voláním metody `rand_mean_as_curr()`,

add(amount) vytvoří nový vzorek využití paměti zvýšený o `amount` MB oproti předešlému objemu,

reduce(amount) vytvoří nový vzorek využití paměti snížený o objem `amount` MB oproti předešlému objemu,

balloon_disable() tato metoda je dostupná pouze pro skutečné guesty a při exportu dat označí tento stroj příznakem, který vyjadřuje chybějící podporu ballooningu. Funkci tak lze volat kdykoli před exportem.

6.4.2 Ukázka

Následuje příklad použití generátoru pro vygenerování scénáře:

```
# Inicializace simulátoru pomocí hosta - má k dispozici 16GB fyzické paměti
sim = Simulator(16000)

# přidáme 2 guesty s maximální pamětí 2 GB, aktuálně dostupná je však 1.8 GB
sim.add_guest(2000, 1800)
sim.add_guest(2000, 1800)

# Nejdříve nastartujeme pouze hosta s využitými 10 GB. Guesti jsou vypnutí.
sim.host.start(10000)
map(lambda x: x.no_change(), sim.guests)

# Spustíme všechny guesty najednou, každý s obsazeným 1 GB
sim.host.no_change()
map(lambda x: x.start(1000), sim.guests)

# Následující příkazy neprodukují nové stavy pro simulátor, pouze změní
# jeho vnitřní stav.
sim.host.rand_mean_as_curr()
map(lambda x: x.rand_mean_as_curr(), sim.guests)
```

```
# Následuje vygenerování 5-ti vzorků, kdy se na guestech náhodně mění
# zatížení s rozptylem hodnot 15MB kolem dříve zapamatovaných 1GB
for i in range(5):
    sim.host.no_change()
    map(lambda x: x.random_norm(mean=None, deviation=15), sim.guests)

sim.export('scenario.csv', comment='16GB host, 2x 2GB VM, ...')
```

6.5 Vizualizace

Pro účel vizualizace byl vytvořen skript `show_plot.py`, který zobrazuje všechna vstupní data v `PolicyEngine` přehledně ve formě grafu. Tyto hodnoty jsou těsně před zpracováním politikami uloženy strukturovaně v textovém formátu JSON. S výhodou se tak tento export dá využít i pro průběžné sledování práce MoMu. Všechny grafy obsažené v této práci jsou výstupem právě tohoto nástroje.

-f | **--file** <**soubor**> zdrojový soubor s daty,

-i | **--interval** <**cislo**> interval v sekundách pro automatické znovunačtení vstupních dat (nepovinný parametr),

-w | **--width** <**cislo**> počet grafů umístěných vedle sebe v okně (nepovinný parametr, standardně se použije 2),

-o | **--output** <**soubor**> název souboru pro export výsledného obrázku. Výstupní formát se odhadne podle použité přípony souboru. Pokud je nastaven interval automatického načítání vstupu, dochází ve stejných intervalech i k novému generování výsledného obrázku. Obrázek lze ale získat přímo z hlavního okna grafu pomocí tlačítka s ikonou diskety i přesto, že tento parametr nebyl použit (nepovinný parametr),

-q | **--quite** potlačí vytvoření a zobrazení okna s grafem. Provede zpracování vstupu a vykreslí graf do vnitřního bufferu. Tato volba je užitečná pouze v kombinaci `-o` pro dávkový export vstupního JSON do formy obrázku. Parametr `-i` pro opakované načtení vstupních hodnot je ignorován.

6.5.1 Ukázka použitého formátu

Je zcela v pořádku, že v hodnotách guestů chybí některé indexy. Jsou totiž způsobeny vypnutým stavem virtuálního stroje (také validní stav). Aby bylo zamezeno ukládání prázdných hodnot, jsou jednotlivé indexy vzorků ukládány jako klíče asociativního pole, což umožňuje snadno uchovávat i takto nesouvislé řady hodnot. JSON obsahuje klíče reprezentující jednotlivé guesty. Každý z nich obsahuje datové řady.

```
{
  "host": {
    "mem_free": {
      "0": 22000.0,
      "1": 986.0,
      "2": 986.0,
      "3": 2052.0,
      "4": 3033.5,
      "5": 4003.125,
      "6": 4904.37,
      "7": 5749.603,
      "8": 6593.076,
      "9": 7263.947
    },
    "mem_available": {
      "0": 64000.0,
      "1": 64000.0,
      "2": 64000.0,
      "3": 64000.0,
      "4": 64000.0,
      "5": 64000.0,
      "6": 64000.0,
      "7": 64000.0,
      "8": 64000.0,
      "9": 64000.0
    },
    "_mem_used": {
      "0": 42000.0,
      "1": 63014.0,
      "2": 63014.0,
      "3": 61948.0,
      "4": 60966.5,
      "5": 59996.875,
      "6": 59095.630,
      "7": 58250.397,
      "8": 57406.924,
      "9": 56736.053
    }
  },
  "fake-vm-1": {
    "balloon_cur": {
      "4": 3610.0,
      "2": 3429.5,
      "3": 3258.025,
      "7": 4000.0,
      "8": 3800.0
    },
    "mem_unused": {
      "4": 1612.0,
      "2": 1468.5,
      "3": 1249.025,
      "7": 2000.0,
      "8": 1782.0
    },
    "_mem_used": {
      "4": 1998.0,
      "2": 1961.0,
      "3": 2009.0
      "7": 2000.0,
      "8": 2018.0
    }
  },
}
```

7 Testovací scénáře a evaluace aktuálních politik

Pro ověření práce MoMu bylo navrženo několik různých scénářů, které popisují případy možných situací. Součástí každého z nich je popis prostředí, parametrů a samotný průběh. Cílem je ukázat možné slabé stránky stávajících politik.

V následujících částech se bude často objevovat termín *pseudokonstantní zatížení paměti*.

Pseudokonstantní zatížení paměti – tímto termínem je myšlena náhodná (Gaussovo rozložení) změna využití paměti v čase s konstantní střední hodnotou. V textu vždy těsně následuje uvedená použitá velikost rozptylu.

Zpožděné balónování – jedná se o jev zřejmý z grafického výstupu simulátoru. V případě, že je guest stlačen, vypnut a opětovně spuštěn, je ve výsledném grafu vidět jeho počáteční velikost paměti po novém startu nižší, než byla v posledním spuštěném vzorku. Tento výsledek je korektní, protože MoM Simulátor poskytuje hodnoty ještě *před* zpracováním dat politikami (tzn. vstup `PolicyEngine`). Data jsou zpracována v tomto pořadí: získání dat z kolektorů, export pro LivePlotter, zpracování politikami, změna velikosti paměti kontroléry (další interval). Proto dojde k exportu pro graf, následně změna balónu a teprve v dalším intervalu je tato změna patrná z grafu.

7.1 Princip funkce stávajících politik

Nejdříve se prověří stav hosta, zda mu již dochází pamět (je tzv. pod tlakem) a podle toho bude rozhodnuto, zda bude přidělená paměť guestů růst (funkce `grow_guest`, dostanou od hosta paměť) nebo jim bude naopak odebrána (funkce `shrink_guest`) dostupná paměť. Podle toho se zavolá příslušná metoda pro každého guesta zvlášť.

```
(defvar host_free_percent (/ (Host.StatAvg "mem_free")
                             Host.mem_available))
(if (< host_free_percent pressure_threshold)
    (with Guests guest (shrink_guest guest))
    (with Guests guest (grow_guest guest)))
```

grow – účelem této metody je regulovaným způsobem vrátit paměť guestovi až do jeho maxima. V případě, kdy již guest dostal zpět veškerou svou paměť, nekoná se zde nic zajímavého a metoda ve výsledku nic neprovede. Využitá paměť se zde počítá z průměrných hodnot statistik (za posledních `sample-history-length` intervalů). Nejmenší objem volné paměti, kterou necháme pod správou guesta, je 20 % z aktuálně dostupné paměti.

```
(def grow_guest (guest)
{
```

```
  (if (< guest.balloon_cur guest.balloon_max) {
    (defvar guest_used_mem (- (guest.StatAvg "balloon_cur")
                  (guest.StatAvg "mem_unused")))
    (defvar balloon_min (max guest.balloon_min (+ guest_used_mem
                (* min_guest_free_percent guest.balloon_cur))))
```

Vypočítáme (dočasně) objem nově přidělené paměti jako zvýšení stávajícího objemu o 5 % (hodnota `max_balloon_change_percent`).

```
    (defvar balloon_size (* guest.balloon_cur
                (+ 1 max_balloon_change_percent)))
```

Toto číslo však může nabývat hodnoty mimo validní rozsah, a proto je nutné ji ještě zkorigovat – nesmí být záporná a nesmí překročit svůj maximální objem dostupné paměti.

```
    (if (< balloon_size balloon_min)
      (set balloon_size balloon_min) 0)
    (if (> balloon_size guest.balloon_max)
      (set balloon_size guest.balloon_max) 0)
```

Posledním krokem je už pouze opatření vůči nepatrným změnám velikosti balónu. To způsobuje overhead, který takto můžeme omezit.

```
    (if (change_big_enough guest balloon_size)
      (guest.Control "balloon_target" balloon_size) 0)
  } 0)
})
```

shrink – způsobuje tlak na guesty postupným ubíráním dostupné operační paměti, i za cenu swapování gestů. Politiky rozlišují mezi dvěma stupni nedostatku paměti hosta:

- *normální*, pokud klesne volná paměť pod 20 %. To způsobí začátek utlačování guestů,
- *kritický* má hranici 5 % a operuje s aktuální mírou tlaku na hosta jako objemem volné paměti hosta. Hodnota v tomto případě bude vždy nabývat záporných hodnot, což jak bude dále vysvětleno, způsobí přímo úměrné využití swapu guesta.

```
(def shrink_guest (guest)
{
  (if (<= host_free_percent pressure_critical)
    (defvar guest_free_percent (+ -0.05 host_free_percent))
    (defvar guest_free_percent (* min_guest_free_percent
                  (/ host_free_percent pressure_threshold))))
```

Využitá paměť se počítá stejným způsobem, jako je tomu u `grow_guest` (aritmetický průměr posledních několika vzorků dat). Nejmenší hodnota nové velikosti paměti guesta (označována jako `ballon_min`) se vypočítá podle následujícího výrazu:

```
balloon_min = max(guest.balloon_min,
guest_used_mem + (guest_free_percent * guest.balloon_cur))
```

Konkrétně, pokud bychom uvažovali guesta, který má k dispozici 1000 MB (`guest.balloon_cur`), z toho obsazených 600 MB (`guest_used_mem`) a host má volné pouze 2 % paměti (podle výrazu výše je vyjádřeno jako $-0.05+0.02$) (výchozí hodnota `guest.balloon_min` je `0`),

```
balloon_min = max(0, 600 + (-0.03 * 1000)) = 570
```

Tímto způsobem by bylo hostovi navráceno 430 MB (rozdíl mezi 1000 a 570) a guest by odložil na swap 30 MB.

```
(defvar guest_used_mem (- (guest.StatAvg "balloon_cur")
              (guest.StatAvg "mem_unused")))
(defvar balloon_min (max guest.balloon_min (+ guest_used_mem
             (* guest_free_percent guest.balloon_cur))))
```

Takto velká náhlá změna paměti by mohla v guestovi vést k ukončení procesů. Proto se celková velikost paměti guesta zmenší maximálně o 5 % (`max_balloon_change_percent`)

```
(defvar balloon_size (* guest.balloon_cur
            (- 1 max_balloon_change_percent)))
(if (< balloon_size balloon_min)
  (set balloon_size balloon_min)
  0)
```

V případě, že je změna objemu paměti dostatečně velká, nastaví se kontroléru nová hodnota skrze `balloon_target` na již dříve vypočítanou `balloon_size`.

```
(if (and (<= balloon_size guest.balloon_cur)
    (change_big_enough guest balloon_size))
  (guest.Control "balloon_target" balloon_size)
  0)
})
```

Funkce `change_big_enough` předchází overheadu, způsobený prováděním množstvím drobných změn, které nemají významný vliv na chod obou systémů. Vrací hodnotu `1` pokud je chystaná změna velikosti paměti větší, než udává konstanta `min_balloon_change_percent` procent z aktuálně přidělené velikosti paměti. V opačném případě vrací `0`, což vede k nesplnění podmínky `if`.

```
(def change_big_enough (guest new_val)
{
    (if (> (abs (- new_val guest.balloon_cur))
           (* min_balloon_change_percent guest.balloon_cur))
        1 0)
})
```

7.2 Použitá konfigurace MoMu

Chování MoMu je možné ovlivnit sadou hodnot pomocí konfiguračního souboru. Protože jejich hodnoty ovlivňují výsledné chování MoMu, následuje pro úplnost výčet použité konfigurace, která byla použita pro veškeré výsledky v této práci. Jedná se ale pouze o výřez ze souboru, který však přímo ovlivňuje výsledky. Veškeré ostatní volby a komentáře jsou vynechány.

```
[main]
main-loop-interval: 2
host-monitor-interval: 2
guest-monitor-interval: 2
guest-manager-interval: 2
policy-engine-interval: 2
hypervisor-interface: fake
controllers: Balloon
sample-history-length: 5

[simulator]
source-file: mom/scenario.csv

[liveplot]
fields: balloon_cur, mem_unused, mem_free, _mem_used, mem_available
export-samples: plot.json

[host]
collectors: FakeHostMemory

[guest]
collectors: GuestMemory, GuestBalloon
```

7.3 Scénář 1: Vynucení swapování na hostovi

Smyslem tohoto scénáře je vyvinout náhle tak silný tlak na hosta, aby tak byl donucen využít swap. Tato situace je obecně vzato nežádoucí, protože využívání swapu na hostovi má negativní dopad na odezvu celého systému včetně VM, které na něm běží.

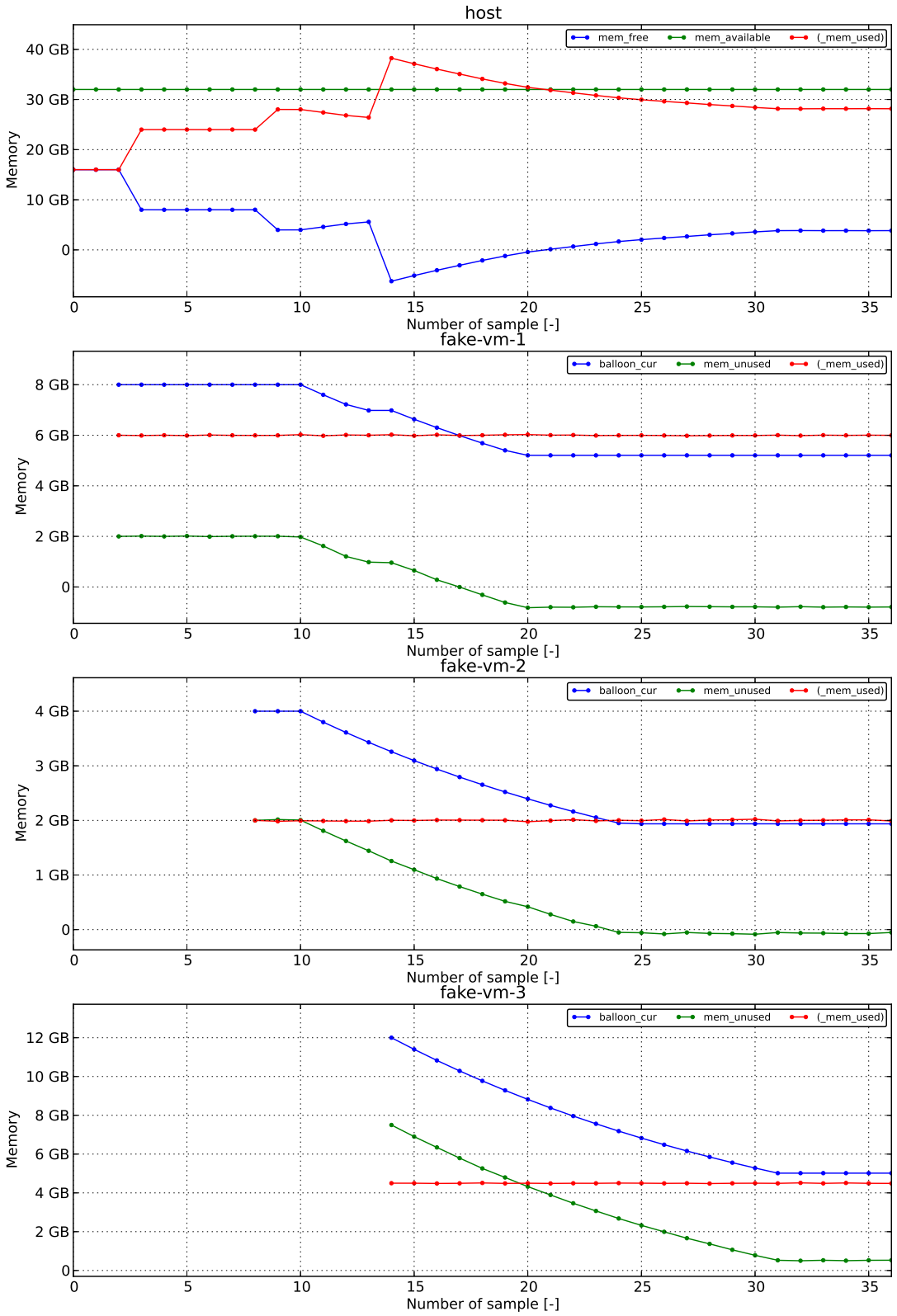
Parametry hosta 32 GB, pseudokonstantní zatížení paměti 16 GB (rozptyl 10 MB).

Parametry virtuálních strojů 3× VM s maximální dostupnou pamětí 8, 4 a 12 GB, pseudokonstantní zatížení paměti 6, 2, 4.5 GB (vždy s rozptylem 10 MB), všechny s podporou zařízení pro memory ballooning, jednotlivé VM startují postupně s rozestupy 5 vzorků.

Popis scénáře Postupně se provede zapnutí prvních dvou VM. Tím dojde k dostatečnému využití paměti hosta, což způsobí balónování. Brzy poté se zapne poslední VM s 12 GB a host tak začne odkládat stránky v paměti na swap. Tím je dosaženo silného tlaku na hosta. Všechna VM jsou nadále spuštěná až do konce simulace, aby bylo možné sledovat dobu stlačování virtuálních strojů.

7.3.1 Výsledky

Z průběhu využití paměti hosta je zřejmé, že balónování sice vrací hostovi postupně paměť v případě tlaku, ale ten trvá více než 15 vzorků. Tato délka by měla být co nejkratší, protože odkládání stránek v paměti na swap je významnou zátěží celého systému hosta (především IO operace a využití procesoru). Je zde dobře patrná funkce MoMu – postupně omezovat guesty ve prospěch hosta v případě, že má nedostatek vlastní volně paměti.



Obrázek 8: Scénář 1: využití paměti s původními politikami

7.4 Scénář 2: Host s velkým množstvím paměti

Pokud máme k dispozici hosta s např. 64 GB fyzické paměti, můžeme si dovolit na něm provozovat větší počet virtuálních strojů. Na takto výkonné konfiguraci ale dochází k chybné detekci hranice, kdy je host *skutečně* pod tlakem a dochází mu paměť. Tato hranice je stanovena v politikách jako pevně definované procento z celkové dostupné fyzické paměti hosta.

Aktuálně to jsou 20 % (hranice, která spouští balónování) a 5 % (indikuje silný nedostatek paměti a agresivněji stlačuje guesty). Při 8 GB je 5 % 400 MB, což se často dá uvažovat za rozumnou rezervu. V případě 64 GB je ale tato hranice rovna 1.28 GB. Při takovém volném objemu paměti si ještě můžeme dovolit spustit další virtuální stroj s 1 GB paměti bez citelné újmy na výkonu celého hostitelského systému.

Parametry hosta 64 GB, pseudokonstantní zatížení paměti 18 GB (rozptyl 10 MB). Konkrétní hodnota zatížení byla zvolena tak, aby při spuštění všech virtuálních strojů zbylo hostovi < 20 % volné paměti.

Parametry virtuálních strojů $3 \times$ VM s maximální dostupnou pamětí 12 GB, pseudokonstatní zatížení 5, 5 a 9.5 GB (vše s rozptylem 10 MB)

Popis scénáře Postupně se spustí všechny virtuální stroje (v intervalech 2 vzorků). Celkový objem obsazené paměti hosta tak překročí treshold nastavený jako procento celkové paměti hosta (hodnota `pressure_threshold`) který způsobí ballooning guestů.

MoM přejde k balónování guestů při splnění následující nerovnice, kdy je využitá paměť hostem (18 GB) a všemi guesty ($3 \cdot 12$ GB) větší hodnota hranice `pressure_threshold` (20 %):

$$20\% \text{ z } 64 = 12.8 \text{ [GB]}$$

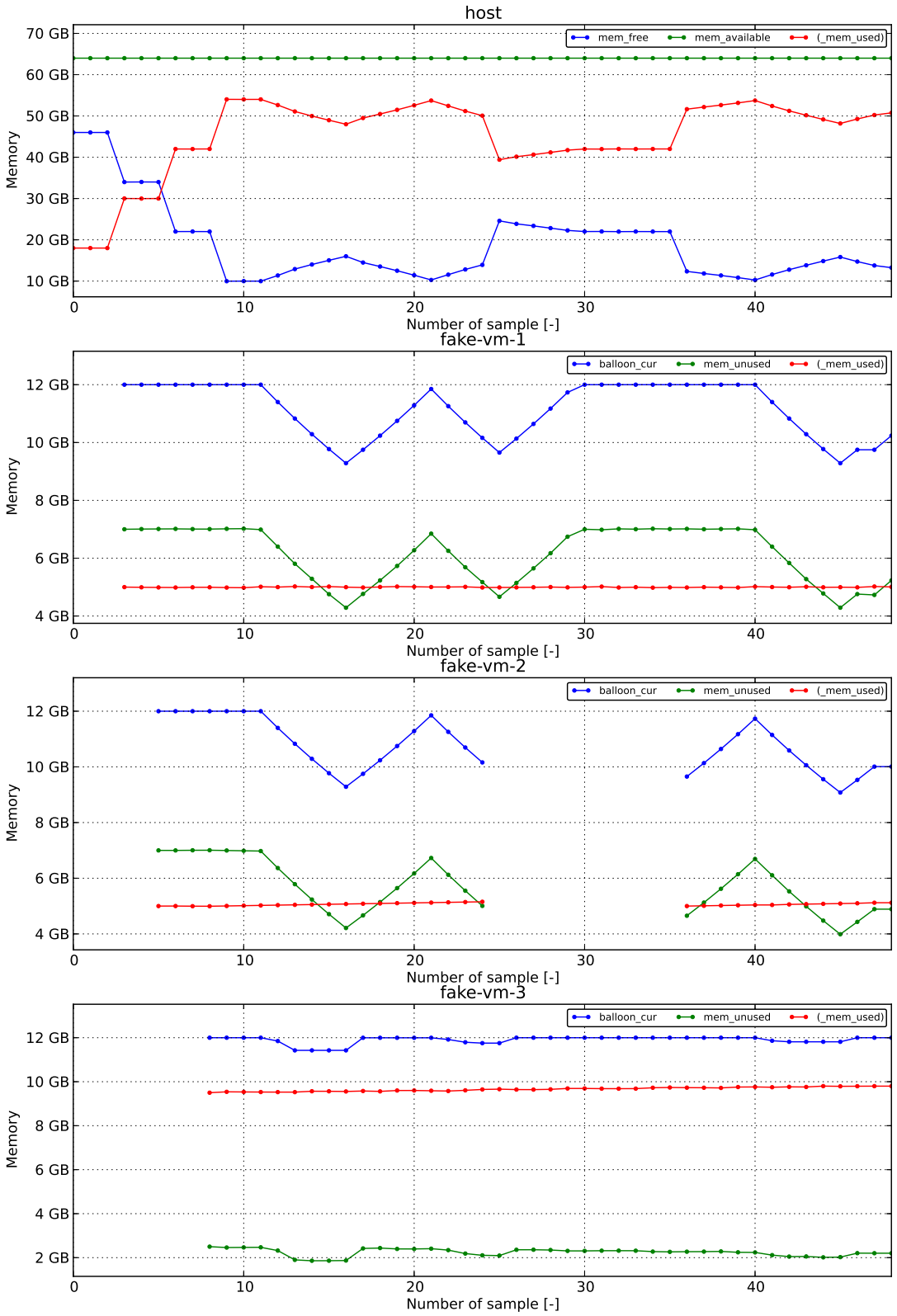
$$64 - 12.8 < 18 + 3 \cdot 12 \text{ [GB]}$$

Poslední VM využívá 9.5 GB a každý 5. vzorek zvýší jeho využití paměti o 40 MB, jinak se použije pseudokonstantní hodnota se středem použitým od posledního zvýšení (rozptyl 10 MB). Využití `fake-vm-2` roste s každým vzorkem o 10 MB a ve 25. vzorku je na dobu 10 vzorků vypnut. Po tuto dobu se host nenachází pod tlakem a to způsobí „nafukování“ guestů až po dosažení jejich maximálního individuálního objemu. Potom je `fake-vm-2` opětovně spuštěn a MoM tak znovu začne postupně utlačovat guesty ve prospěch hosta.

7.4.1 Výsledky

Po nastartování všech virtuálních strojů je vidět, že dochází k omezování guestů příliš brzy – politiky přejdou do režimu utlačování a udržují tak na hostu volné místo o objemu cca 10 GB. Takové množství se už rozhodně nedá nazvat nouze o paměť. V konečném důsledku takové chování má negativní vliv na výkon virtuálních strojů, protože tak mají velice omezené prostředky pro využití buffer cache. MoM zůstane v tomto stavu „zaseknut“ do doby uvolnění další paměti a teprve poté přejde k „nafukování“ guestů.

Na `fake-vm-3` je dobře vidět, že se míra stlačení guestů vypočítává individuálně každému guestu samostatně. Toto VM využívá velký objem paměti a proto je utlačeno výrazně méně, než tomu tak je u ostatních dvou virtuálních strojů.



Obrázek 9: Scénář 2 – využití paměti s původními politikami

7.5 Scénář 3: Kombinace guestů s/bez podpory mem-balloonig

Smyslem tohoto scénáře je sledovat chování MoMu s více virtuálními stroji, přičemž některé z nich nepodporují memory ballooning.

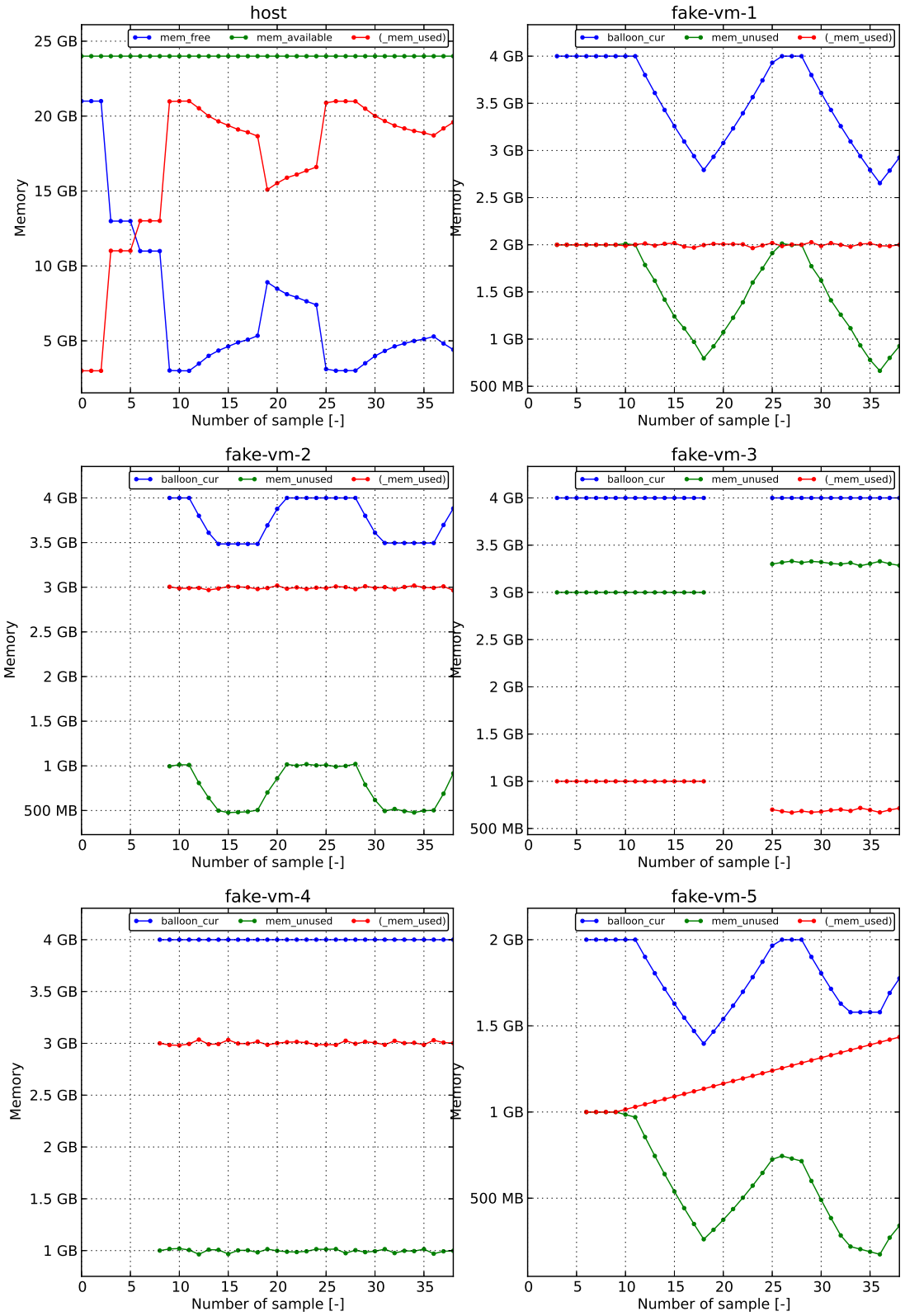
Parametry hosta 24 GB, pseudokonstantní zatížení paměti 3 GB (rozptyl 10 MB).

Parametry virtuálních strojů 5 × VM s maximální dostupnou pamětí 4, 4, 4 (bez podpory balónování), 4 (bez podpory balónování) a 2 GB, s pseudokonstantním zatížením paměti 2, 3, 1, 3 a 1 GB (rozptyl 15 MB).

Popis scénáře Virtuální stroje startují ve skupinách 2, 1 a 2 VM současně. `fake-vm-3` je mezi vzorky 19-24 zastaven. Poté je opětovně spuštěn s využitými 700 MB. `fake-vm-5` po 10. vzorku soustavně zvyšuje svou zátěž o 15 MB s každým vzorkem.

7.5.1 Výsledky

Balónování paměti probíhá korektně, následkem vypnutí `fake-vm-3` dojde k uvolnění tlaku na hosta a to způsobí vrácení části paměti guestům zpět. Změny ve využití paměti guestů `fake-vm-3` a `fake-vm-4` nemají na MoM žádný vliv, protože se jedná z pohledu hosta o konstantní zátěž, kterou nelze žádným způsobem regulovat. Na `fake-vm-5` je patrná změna objemu balónu v závislosti na využité paměti. Vzhledem k zátěži hosta, ani žádné VM ani host sám nepotřebuje využít swap pro získání další paměti.



Obrázek 10: Scénář 3 – využití paměti s původními politikami

7.6 Scénář 4: Host je pod tlakem, restarty VM

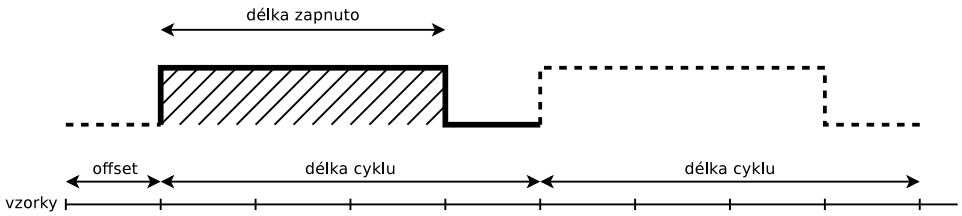
Poslední scénář je pro MoM náročnou situací – virtuální stroje často procházejí restarty, takže běží pouze v krátkých časových úsecích. MoM tak má k dispozici pouze malý objem statistik pro svou činnost.

Parametry hosta 16 GB, konstantní zatížení paměti 3 GB.

Parametry virtuálních strojů 5 × VM, všechny s maximální dostupnou pamětí 3 GB.

Popis scénáře Virtuální stroje jsou spuštěny podle následující tabulky:

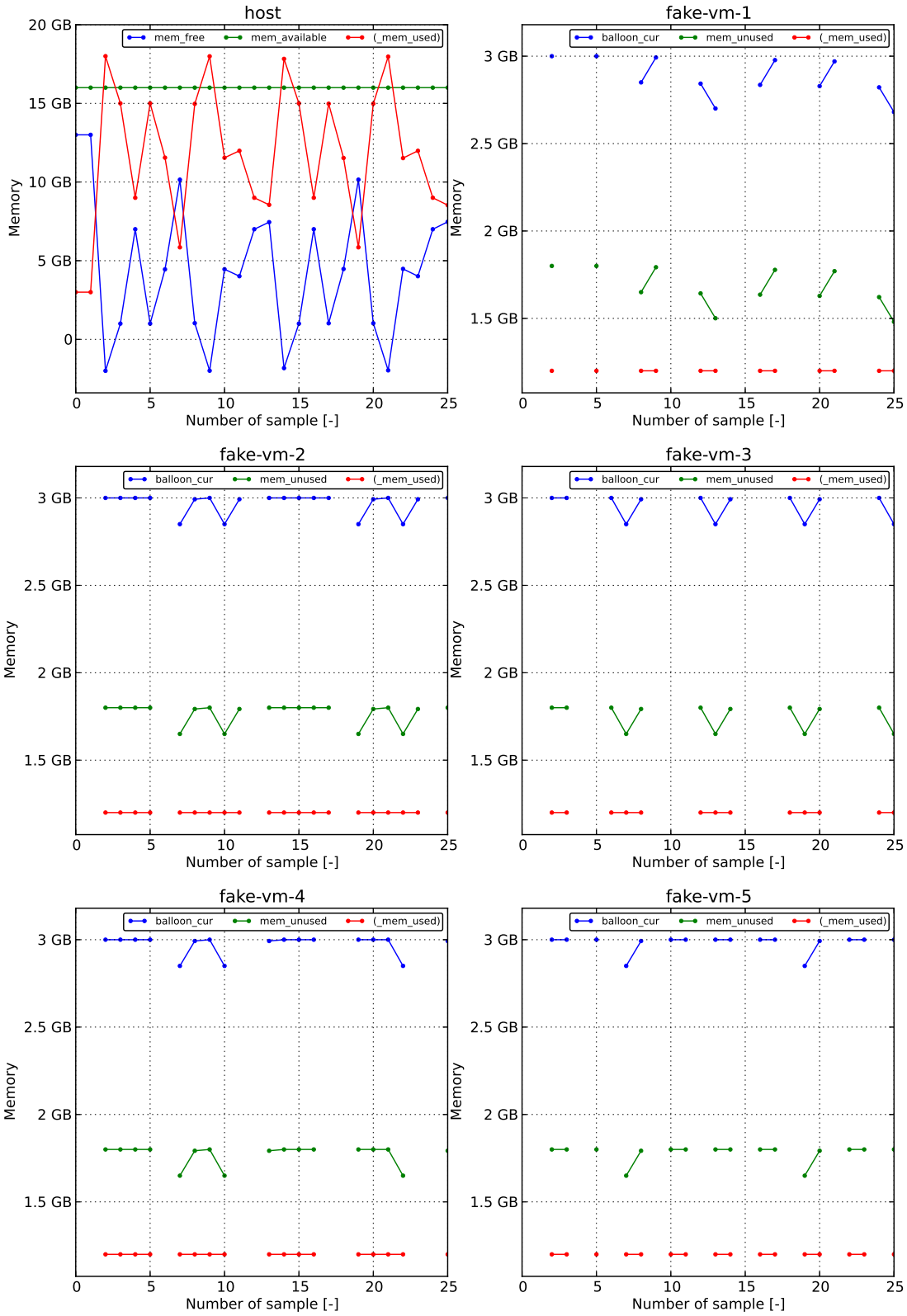
	offset	délka zapnuto	délka cyklu
fake-vm-1	1	2	4
fake-vm-2	0	5	6
fake-vm-3	1	3	6
fake-vm-4	0	4	6
fake-vm-5	0	2	3



Obrázek 11: Scénář 4 – schéma intervalů restartů (konkrétně: 1 | 3 | 4)

7.6.1 Výsledky

Vlivem restartů dochází na hostovi k výrazným skokovým změnám ve využité paměti. Pokud dojde na hostovi ke swapování, MoM na to reaguje velmi pomalu a paměť tak získává pouze v omezeném množství. Efekt balónování je tak více patrný, pouze pokud je guest spuštěn na delší časový úsek. Tento fakt je způsoben výpočtem průměrných hodnot za posledních 5 vzorků (`sample-history-length`), kde časté restarty poskytují nedostatečný počet hodnot pro efektivní funkci politik.



Obrázek 12: Scénář 4 – využití paměti s původními politikami

8 Návrh a evaluace nových politik

V kapitole 7 bylo poukázáno na vybrané problémy, které stávající politiky neřeší vůbec, nebo málo efektivně. Následuje rozbor každého z problémů s navrženým řešením. Všechny výsledky jsou získány ze stejných vstupních dat (scénářů) jako v předešlé kapitole. Jedinou změnou jsou pouze použité politiky, které s těmito daty pracují.

8.1 Agresivnější stlačení stabilních guestů

V části 7.3 (Scénář 1: Vynucení swapování na hostovi) je zřejmé, že i když guest má mnoho prostoru pro stlačení, stávající politiky tento prostor ubírají pouze po malých částech. To znamená dlouhou dobu, kdy je host pod tlakem – což je nežádoucí.

Byla navržena úprava politik, které v případě potřeby budou agresivněji stlačovat guesty s vhodným chováním. Vhodné chování je v tomto případě průběh využití paměti, který se mění pouze minimálně a dá se tak považovat za konstantní. Míru konstantnosti popisují statistické charakteristiky *rozptylu* a *směrodatná odchylka*.

V tomto případě je však vhodnější použití *relativní směrodatné odchylky* (někdy je také označována jako *variační rozpětí* či *variační koeficient*), protože její hodnota není v původních jednotkách měřené veličiny jako skalár, ale v procentech.

Rozptyl, označovaný jako s^2, je definován jako průměrná kvadratická odchylka měření od aritmetického průměru, přičemž při průměrování této odchylky dělíme číslem $(n-1)$.

$$s^2 = \frac{\sum_{i=o}^{N}(x_i - \bar{x})^2}{n-1}$$

Směrodatná odchylka (běžně označovaná jako s) je odmocnina z rozptylu a vrací míru rozptýlenosti do měřítka původních dat. Jestliže chceme posoudit relativní velikost rozptýlenosti dat vzhledem k průměru, použijeme *variační koeficient*. Lze ho použít, pokud chceme porovnat rozptýlenost dat skupin měření stejné proměnné s různým průměrem nebo v těch případech, kdy se mění velikost směrodatné odchylky tak, že je přímo závislá na úrovni měřené proměnné $s = k \cdot \bar{x}$, kde k je konstanta.

$$v_x = \frac{\sqrt{s^2}}{\bar{x}}$$

(Hendl, 2012)

8.1.1 Implementace v politikách

```
# V případě, že guest využívá téměř konstantní objem paměti,
# můžeme jej stlačit mnohem agresivněji. Tato konstanta
# vyjadřuje objem paměti v procentech o který bude guest
# stlačen.
(defvar aggresive_balloon_change_percent 0.5)

# Treshold použitý k rozeznání malých změn využité paměti
# guesta. Je to hodnota relativní směrodatné odchylky
# využité paměti. Pokud se guest dostane pod tuto hodnotu,
# bude aplikováno agresivnější stlačování.
# Konkrétně o aggresive_balloon_change_percent procent.
(defvar max_std_dev 0.003)

# ... v těle metody shring_guest

(defvar std_deviation (guest.StatStdDeviation '_guest_used'))

# Agresivní stlačení aplikujeme pouze pokud máme k dispozici
# dostatek vzorků pro výpočet směrodatné odchylky.
(if (> std_deviation 0) {
  # Výpočet hodnoty relativní směrodatné odchylky pro srovnání
  (defvar avg_guest_used (guest.StatAvg '_guest_used'))
  (defvar rel_std_deviation (/ std_deviation avg_guest_used))
  (if (< rel_std_deviation max_std_dev) {
    # Guest se chová "poklidně", proto jej stlačíme rychleji.
    (set balloon_size (*
      guest.balloon_cur
      aggresive_balloon_change_percent))
    }
    0
  )
}
0
)
```

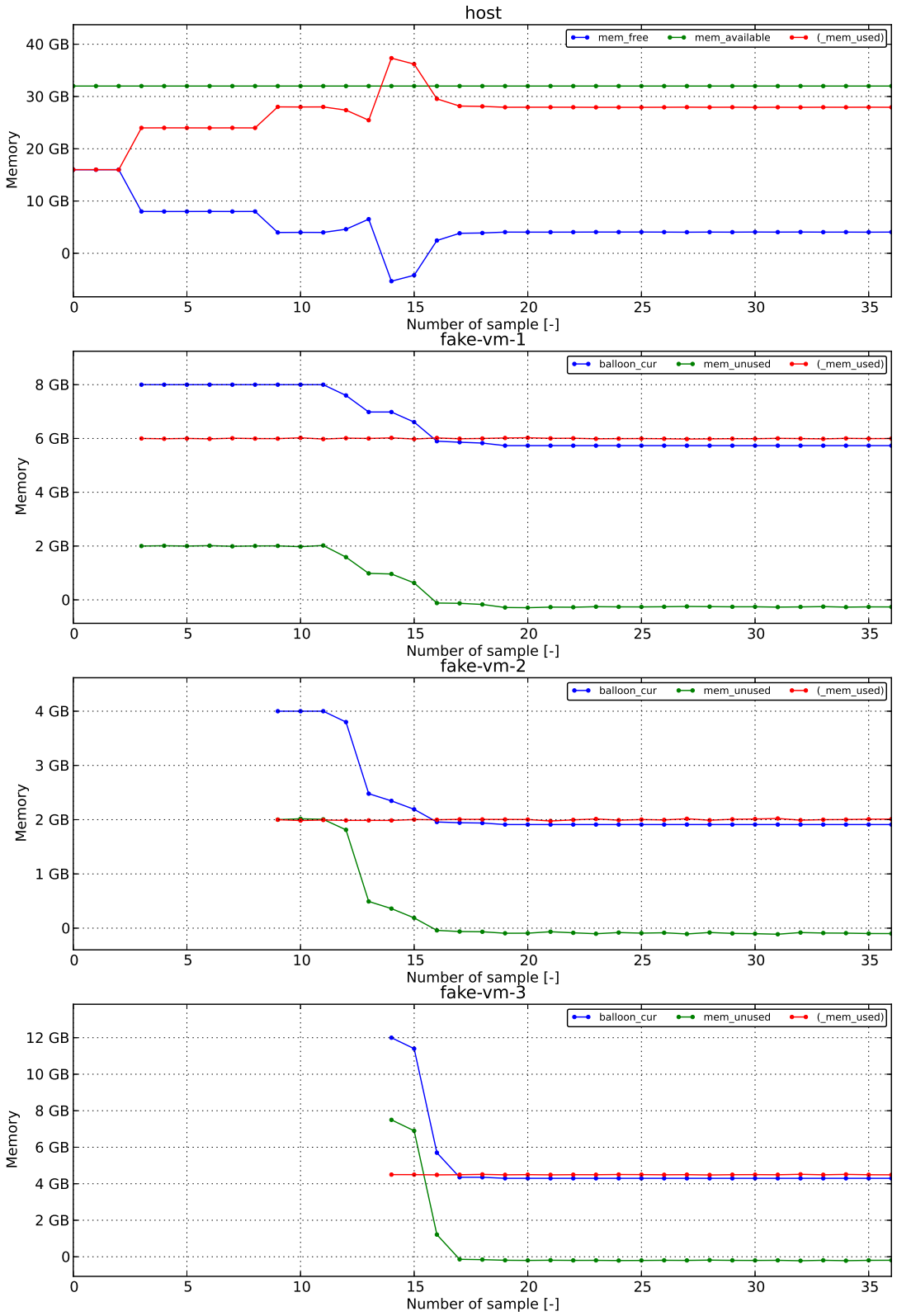
8.1.2 Výsledky scénářů

Scénář 1 Guesti v použitém scénáři nemění své využití paměti nijak výrazným způsobem, což je politikami rozeznáno, a proto se aplikuje výrazně agresivnější stlačení. Guesti v malé míře využívají swap pro uspokojení politik, ale nejde o výrazný objem. Takto se zkrátí doba tlaku na hosta a jedná se tak o žádoucí chování.

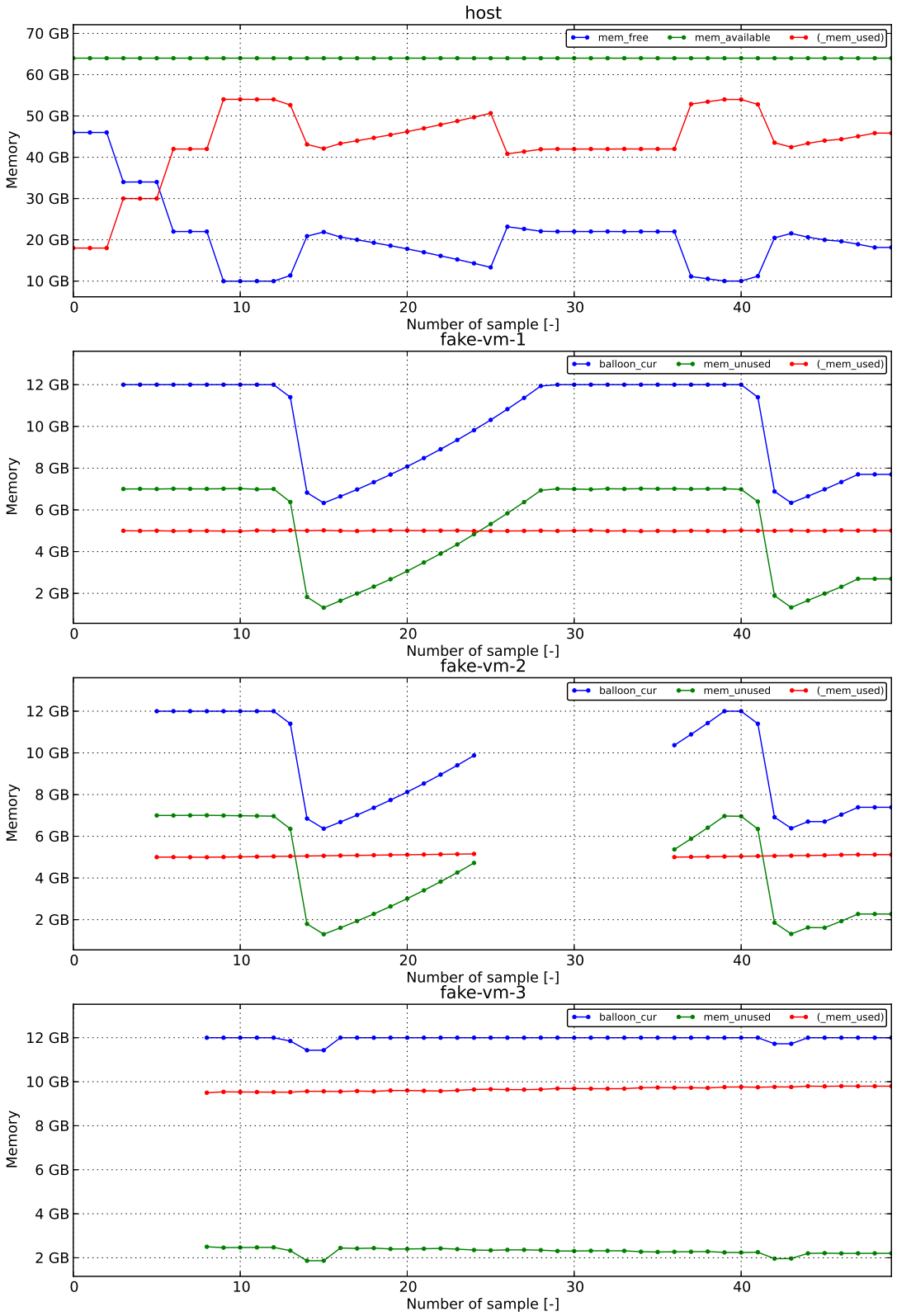
Scénář 2 Stlačování sice rychle omezí dostupnou paměť všech guestů, ale u žádného z nich nedojde ke swapování. Proto se v tomto případě nejedná o problém. Vypnutí `fake-vm-2` uvolní tlak na hosta a ostatní běžící guesti dostanou zpět svůj dříve odebraný objem paměti. `fake-vm-3` nedisponuje velkým množstvím dostupné paměti a nemá tak na efekt balónování významný vliv.

Scénář 3 Efekt změny pravidel je dobře patrný z využití paměti `fake-vm-1` a `fake-vm-5`. První z nich využívá paměť pouze v malém rozptylu a proto jej politiky stlačí agresivněji během několika vzorků, zatímco u dalšího využití souvisle roste a rychlé odebrání velkého množství paměti by brzy způsobilo silné využití swapu tohoto guesta. Balónování je proto provedeno podle původních pravidel, tzn. *férově*. Krátké vypnutí `fake-vm-3` způsobí uvolnění tlaku na hosta a během této doby dochází u všech virtuálních strojů k znovunavracení paměti.

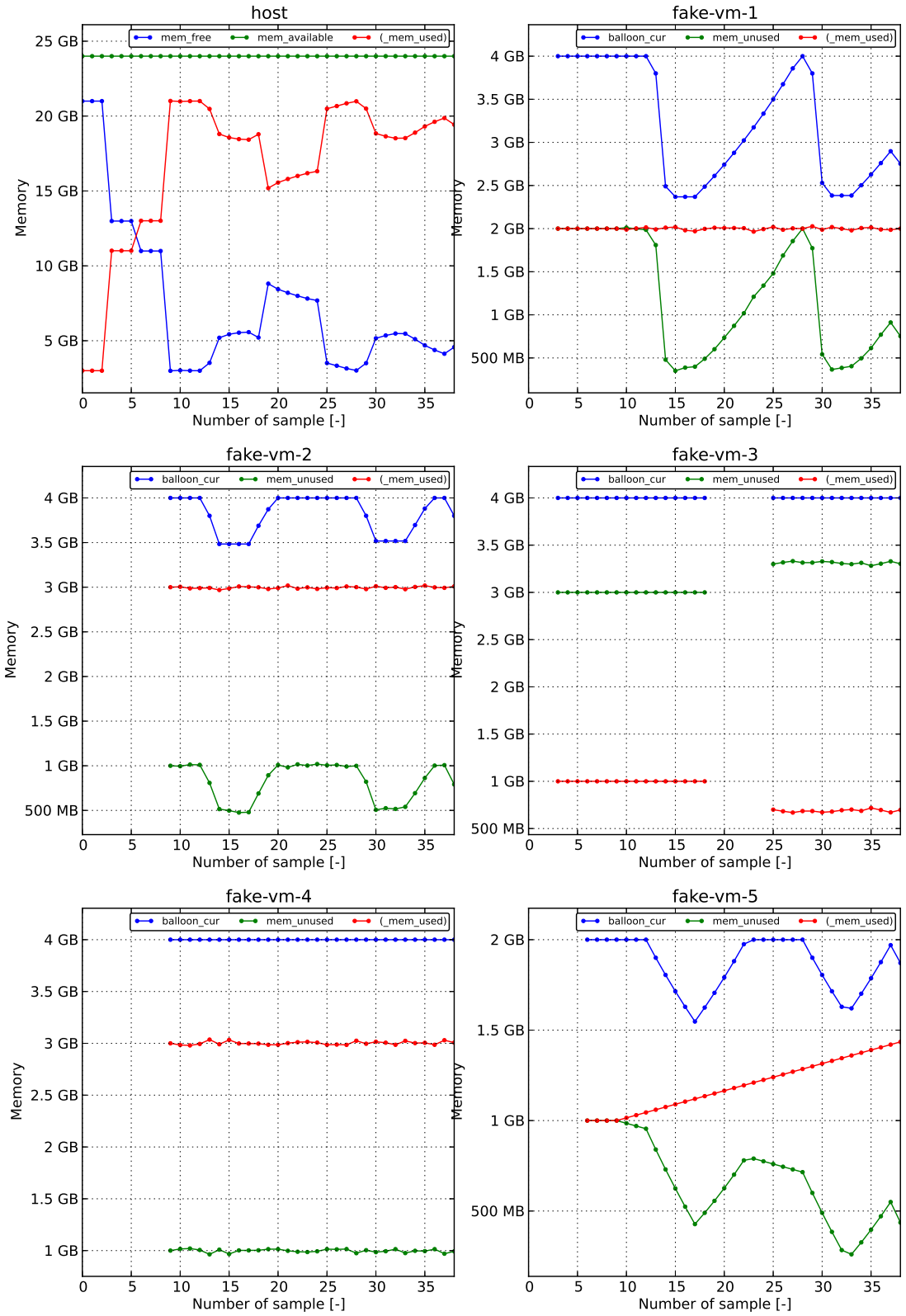
Scénář 4 V tomto scénáři se téměř balónování neuplatní vlivem krátké doby běhu virtuálních strojů. To způsobuje silný tlak na hosta, ale ten není souvislý. Proto se zde změna pravidel pro rychlé stlačení neprojeví.



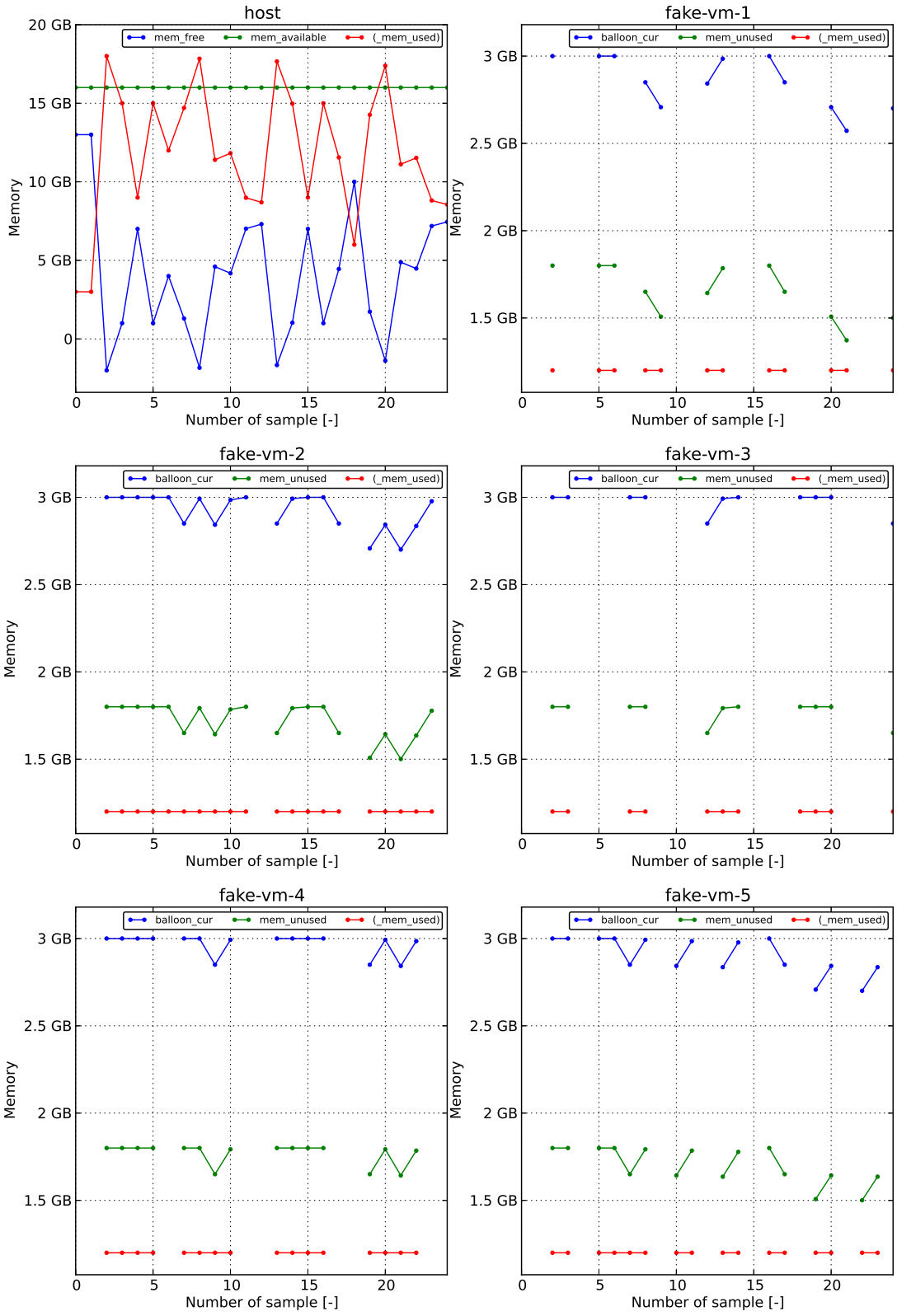
Obrázek 13: Scénář 1, využití paměti s novými politikami 1



Obrázek 14: Scénář 2, využití paměti s novými politikami 1



Obrázek 15: Scénář 3, využití paměti s novými politikami 1



Obrázek 16: Scénář 4, využití paměti s novými politikami 1

8.2 Spuštění balónování při skutečném nedostatku

Aktuální implementace politik rozeznává hranice míry nedostatku paměti hosta jako procentní část celkové fyzické paměti. Pokud ale použijeme hosta s velkým množstvím paměti (například 64 GB), reprezentují tato procenta nepřiměřené objemy a dochází tak k předčasnému balónování.

Upravené politiky definují hodnoty hranic dvojím způsobem: procentem (tzn. původní přístup) a konkrétní hodnotou. Podle konkrétní hodnoty využité paměti se porovná vždy to nižší číslo bez ohledu na jeho původ (z procent nebo konkrétní hodnoty).

8.2.1 Implementace v politikách

Původní konstanty byly pro odlišení doplněny o suffix `_percent`.

```
(defvar pressure_threshold_percent 0.20)
# 20% or 2GB
(defvar pressure_threshold 2000000)

(defvar pressure_critical_percent 0.05)
# 5% or 500MB
(defvar pressure_critical 500000)
```

Následně jsou přepočítány procentní hranice na konkrétní objem paměti v `kB`, které jsou dále porovnávány.

```
(defvar host_free_percent (/ (Host.StatAvg "mem_free") Host.mem_available))
(defvar host_free_kilobytes (Host.StatAvg "mem_free"))
(defvar host_pressure_kilobytes (* Host.mem_available
  pressure_threshold_percent))
(defvar host_critical_pressure_kilobytes (* Host.mem_available
  pressure_critical_percent))
```

Pomocná metoda (tzv. *helper*), jejímž smyslem je rozlišit, zda je překročena hranice udaná v % nebo v `kB`. Podle této volby se pak provede běžné porovnání volné paměti hosta s hodnotou hranice.

```
(def host_is_under_pressure (foo) {
  # find out which treshold is closer (in % or kB), equivalent in C:
  # if(treshold_kB < treshold_percent) { /* use treshold_kB */ } else
  #   { /* use treshold_percent */ }
  (if (< pressure_threshold host_pressure_kilobytes) {
    (debug 'normal pressure - use kB, because ' pressure_threshold
      ' < ' host_pressure_kilobytes ' (% < kB)')
    (if (< host_free_kilobytes pressure_threshold) 1 0 )
  } {
```

```
    (debug 'normal pressure - use percentage, because '
      pressure_threshold ' > ' host_pressure_kilobytes ' (% > kB)')
    (if (< host_free_percent pressure_threshold_percent) 1 0)
  })
})
```

Naprosto stejným principem byla vytvořena i druhá metoda s podobným názvem `host_is_under_critic_pressure`, která však operuje s relevantními proměnnými (s `_critic` v názvu).

Metody `host_is_under_pressure` a `host_is_under_critic_pressure` nepřijímají žádný použitý parametr, pouze vrací hodnotu `0` či `1`. Číslo `0` jako jediný parametr není v metodě nikde použit, jedná se o dodržení syntaxe v aktuální implementaci interpretu použitého dialektu jazyka Lisp. Používají se proto následujícím způsobem:

```
(if (host_is_under_critic_pressure 0) {
    (debug 'Host is under critical pressure')
} {
    (debug 'Host is NOT under critical pressure')
})
```

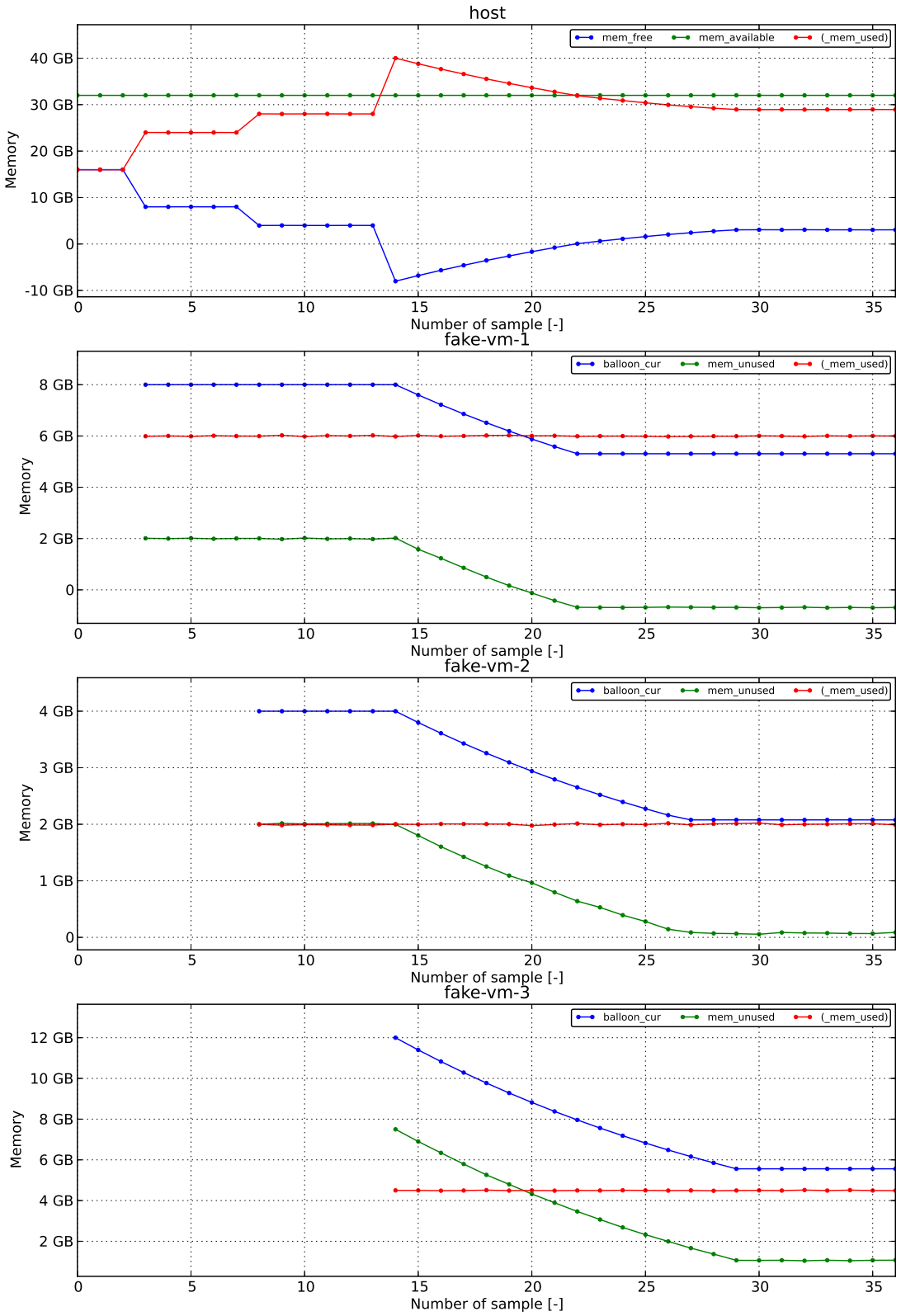
8.2.2 Výsledky scénářů

Scénář 1 Pro hosta s 32 GB pamětí se již aplikují upravená pravidla, a proto MoM začne s balónováním až při poklesu volného místa pod 2 GB. Než se spustí `fake-vm-3` tato hranice překročena není a nedochází k balónování. Následný start způsobí značný tlak na hosta a bez použití MoMu by host odložil na swap 8 GB. S použitými pravidly stlačuje všechny guesty tak dlouho, než tlak ustoupí. Guest `fake-vm-1` je balónován takovou měrou, že u něj dojde k swapování výrazně více než u `fake-vm-2`. Je to způsobeno větší pamětí guesta a tím v důsledku větší skutečnou hodnotou z `max_balloon_change_percent`, ale především délkou samotného balónování.

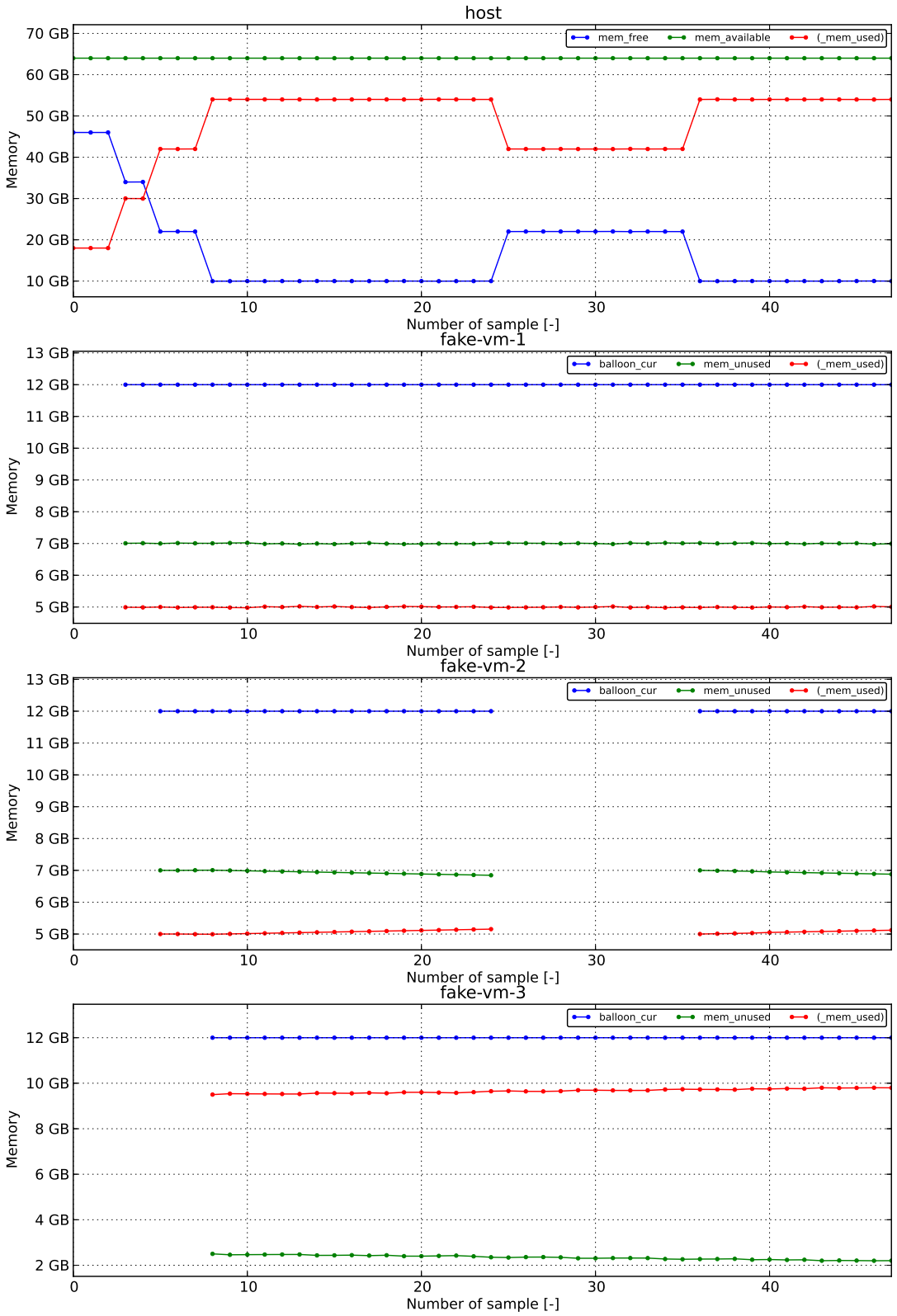
Scénář 2 Na tomto scénáři je dobře patrná změna v politikách, protože vůbec k balónování nedojde. Host má stále k dispozici nejméně 10 GB dostupné paměti a jedná se tak o žádané chování.

Scénář 3 Podobně jako v předcházejícím scénáři ani zde vůbec nedochází k balónování, protože to ještě není nezbytné (nejméně cca 3 GB volného místa). Využití paměti guestů se žádným způsobem neprojeví. Jedná se o korektní chování.

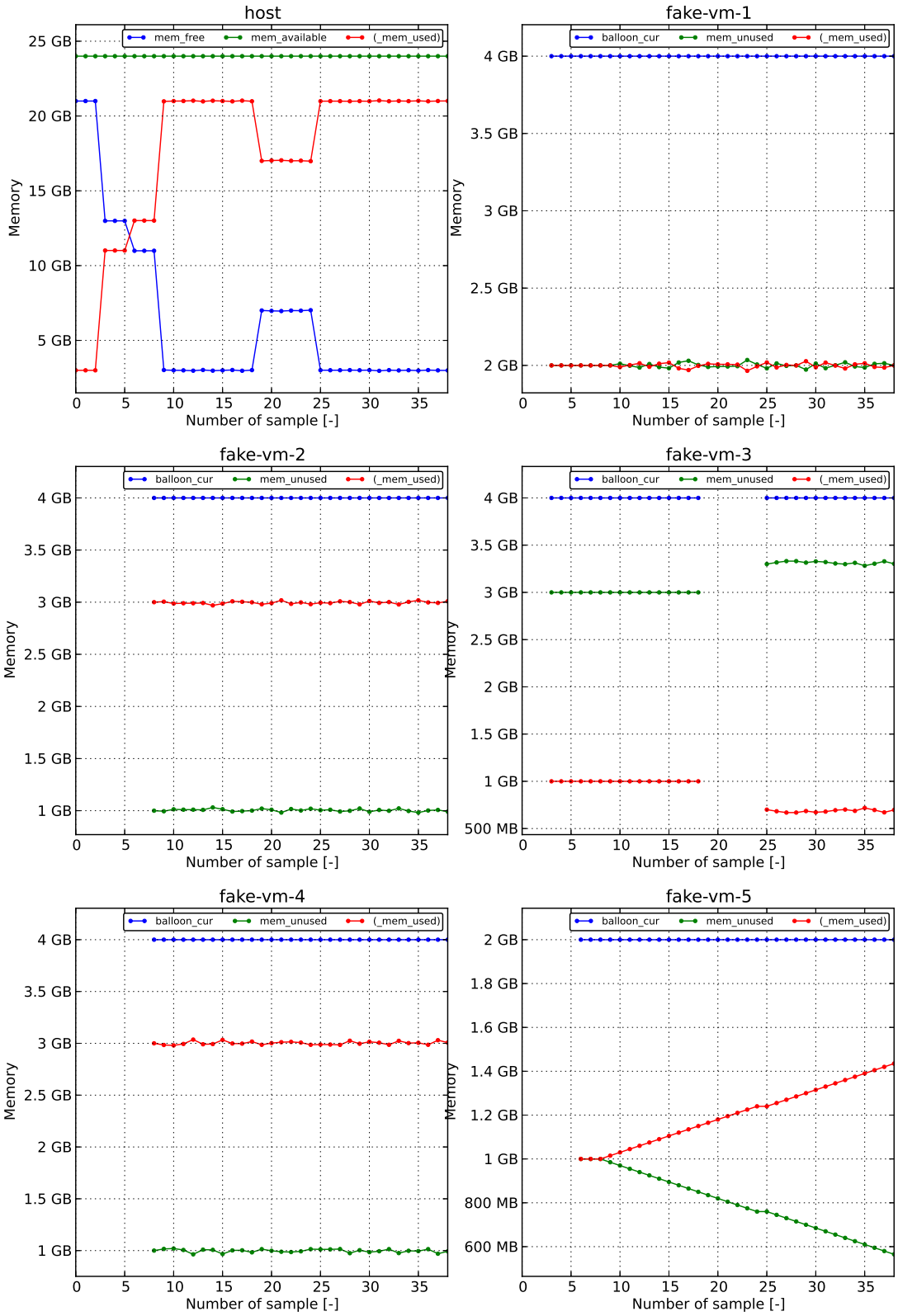
Scénář 4 Host se podle modifikovaných pravidel nachází pod tlakem po dobu nejdéle dvou vzorků, což je zcela nedostatečná doba pro rozeznání tlaku v politikách a k balónování tak vůbec nedojde. Jelikož je host pod tlakem pouze v krátké úseky času, je toto chování ještě akceptovatelné.



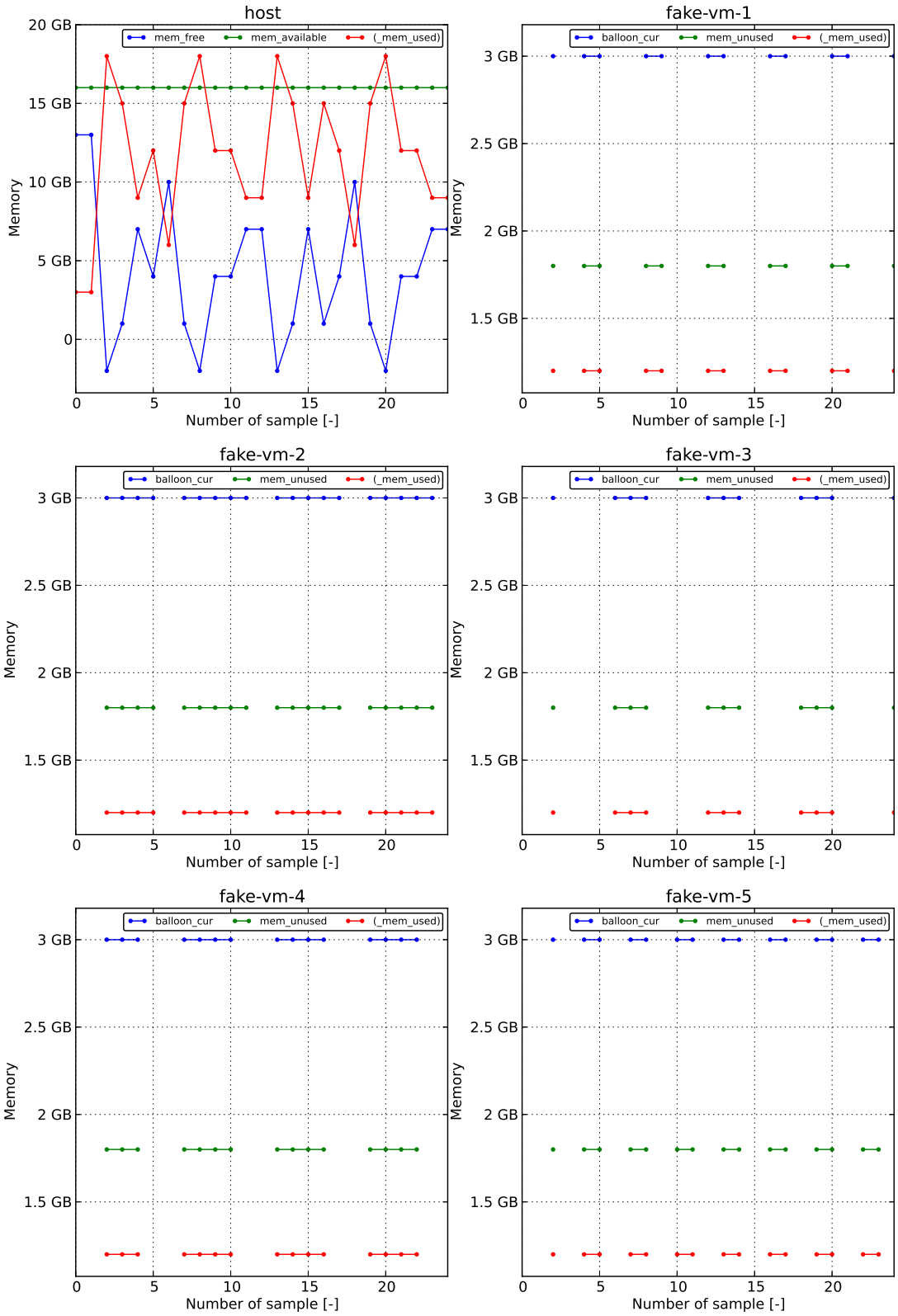
Obrázek 17: Scénář 1, využití paměti s novými politikami 2



Obrázek 18: Scénář 2, využití paměti s novými politikami 2



Obrázek 19: Scénář 3, využití paměti s novými politikami 2



Obrázek 20: Scénář 4, využití paměti s novými politikami 2

9 Nástroj pro správu politik

V současné době je z pohledu politik pohlíženo na všechny virtuální stroje stejně (hodnoty tresholdů, velikost minimální dostupné paměti atd.). Vznikl tak nástroj, který dovoluje definovat a spravovat konstanty v politikách individuálně pro každý virtuální stroj samostatně. Tímto způsobem je tak možné i aplikovat na různé guesty zcela odlišné politiky.

9.1 Popis nástroje

Nástroj pracuje s konstantami (klíč – hodnota) a tzv. plány, které konstanty sdružují do jednoho celku. K virtuálnímu stroji může být přiřazen právě jeden plán.

Přiřazení plánu je aplikováno následovně: Do XML definice virtuálního stroje libvirtu jsou skrze API zapsány hodnoty konstant a identifikátor plánu. `GuestConstantsOptional` kolektor v MoMu si tyto hodnoty přečte a v politikách jsou uplatněny hard-coded hodnoty těchto položek jako výchozí hodnoty. Pokud je konstanta pochází z `GuestConstantsOptional` kolektoru, použije se. V opačném případě bude aplikována výchozí hodnota (definovaná přímo v politikách).

Vzhledem k povaze cílové skupiny uživatelů (systémoví administrátoři) byl nástroj vytvořen jako neinteraktivní aplikace do příkazové řádky, která se ovládá pomocí parametrů. Dalším způsobem jak nástroj `mtool` použít, je jako běžnou objektovou třídu v pythonu. Práce s nástrojem se tak dá snadno automatizovat ve skriptech. Skript operuje se dvěma příkazy `vm` a `plan`, které definoují další podmnožiny parametrů. Všechny přepínače zobrazí `--help`. Příklady použití:

mtool vm --set --plan bronze virt-1 zapíše do virtuálního stroje `virt-1` konstanty z plánu `bronze`, veškeré dříve zapsané konstanty budou ztraceny,

mtool vm --clear virt-1 zcela vyčistí všechna data zapsaná parametrem `--set` ve virtuálním stroji `virt-1`,

mtool vm --get virt-1 přečte název použitého plánu ve virtuálním stroji `virt-1`,

mtool plan --list vypíše názvy všech aktuálně definovaných plánů,

mtool plan --create newplan const_1=12 treshold=0.17 vytvoří nový plán s konstantami `const_1` a `treshold`. Konstanty není nutné definovat ihned při vytvoření plánu, jsou volitelnými parametry libovolného počtu. Pokud už plán `newplan` existuje, je vyvolána vyjímka a k přepisu hodnot nedojde,

mtool plan --update newplan const_1=4 treshold=0.31 zapíše nové hodnoty konstant do plánu `newplan`. Pokud plán `newplan` neexistuje, je automaticky vytvořen. Jedná se tak vlastně o méně bezpečnou variantu `--create`,

mtool plan --delete oldplan odstraní plán `oldplan` včetně všech jeho konstant. Upozornění: tato metoda nemanipuluje s virtuálními stroji, pro odstranění plánu z VM je nutné použít `--clear`.

9.2 GuestConstantsOptional kolektor

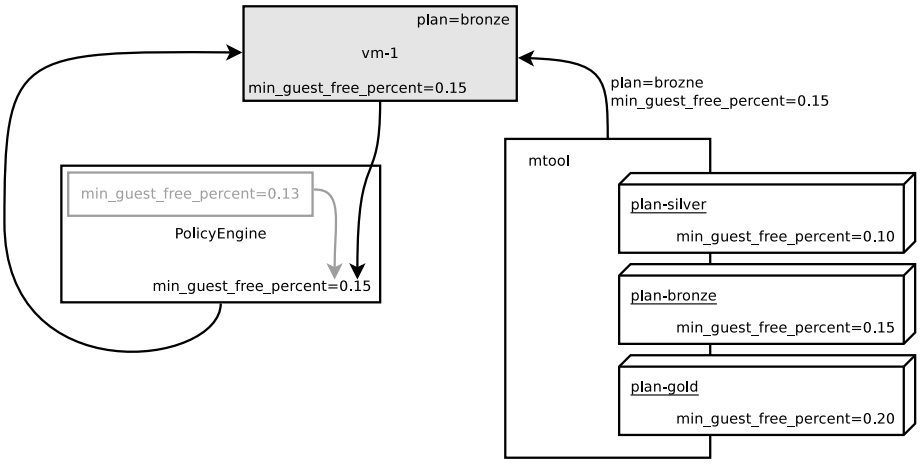
Smyslem tohoto kolektoru je získat z XML definice virtuálního stroje `libvirtu` (konkrétně z elementu `<metadata>`) konstanty, které je pak možné použít v politikách a dosáhnout tak individuálního zpracování pro toto VM. Konstanty jsou poskytovány jako tzv. volitelné položky (optional fields). To znamená, že jejich existence v poskytovaných datech není nezbytná.

9.3 Implementace v politikách

Objekt reprezentující guesta poskytuje metodu `Stat`, která dovoluje definovat výchozí hodnotu, pokud není požadovaná položka (field) získána z kolektorů. Tak se dá docílit přepsání (override) výchozí hodnoty jinou, která pochází přímo z metadat definice virtuálního stroje.

```
# Výchozí hodnota, která se použije pokud guest nemá definovanou
# vlastní hodnotu.
(defvar default_min_guest_free_percent 0.20)

(def shrink_guest (guest) {
# ...
(defvar min_free guest.Stat('const_min_guest_free_percent'
    default_min_guest_free_percent))
# ...
})
```



Obrázek 21: mtool a jeho interakce s MoMem

10 Diskuze a závěr

Tato poslední kapitola je shrnutím celé diplomové práce – obsahuje přehled o používaných metodách správy paměti v prostředí virtualizace a popis známých problémů politik nástroje MoM spolu s návrhem a implementací řešení.

10.1 Diskuze

Tato diplomová práce se zabývá politikami nástroje Memory overcommit Manager a přehledu metod správy paměti guestů hypervisory. Byla identifikována některá nevhodná chování politik a vznikly tak jejich úpravy, které si kladly za cíl tyto situace zlepšit.

Navržené úpravy politik však nebylo možné žádným jednoduchým a rychlým způsobem vyzkoušet. Tato činnost vyžaduje celou řadu přípravných kroků (konfigurace VM atd.) a ověření politik probíhá v reálném čase – což často znamená dobu řádově jednotky minut (průměrná doba na jeden scénář popsaná v této práci by vycházela kolem 5 minut). Tato doba a malá pružnost testování je ale ve fázi návrhu politik nepoužitelná. Byla tak motivací k vytvoření simulátoru, který těmito problémy z velké části netrpí. Umožňuje ověřit funkci pravidel při použití balónování a získat výsledky za třetinový čas v porovnání s reálným nasazením (vhodnou konfigurací lze tento čas ještě více zkrátit).

Ruční příprava dat pro simulátor je svým charakterem zdlouhavý proces vyžadující pečlivost a je tak náchylný k chybám z nepozornosti. Byl tak vytvořen nástroj umožňující jednoduchým způsobem generovat vstupní data programově, přímo v prostředí jazyka Python.

S výstupem simulátoru dále souvisí reprezentace jeho výsledků. Po drobné úpravě by bylo možné použít i aplikace typu Openoffice Calc. Byl ale navržen vlastní vizualizační nástroj, který je možné si zcela přizpůsobit podle potřeby. Je navržen jako samostatná aplikace, která dobře spolupracuje se simulátorem a stále ji lze bez problémů využít i kdekoli jinde.

Tato práce může být odrazovým můstkem pro další rozšíření:

- Podpora dalších metod správy paměti v simulátoru – především KSM (Kernel Same page Merging),
- použití strukturovaného formátu vstupních dat simulátoru – bylo by tak možné popsat virtuální stroj dalšími parametry (metadaty) a více se tak přiblížit reálnému provozu (např. latence splnění požadavku na změnu velikosti balónu),
- podpora RPC[12] v nástroji pro správu konstant umožňující jeho ovládání z dalších aplikací.

[12]Remote Procedure Call (RFC 5531, 2009)

10.2 Závěr

Cílem této diplomové práce bylo navrhnout politiky pro efektivní autoballooning operační paměti virtuálních strojů, které jsou použity v nástroji MoM (Memory overcommit Manager). Stávající implementace politik však v některých situacích nepracují vhodným způsobem, a proto byly navrženy takové změny, jejichž cílem bylo odstranit tato nežádoucí chování. Pro ověření chování politik vznikl MoM Simulator společně s dalšími pomocnými nástroji. Simulátor podporuje *memory ballooning* v MoMu, což je technika implementovaná v mnoha hypervisorech (viz Tabulka 1 v kapitole 3) a její použití tak není omezeno pouze na KVM.

Pro účely vyhodnocení funkce byly navrženy a vytvořeny čtyři scénáře (případy užití). Scénáře 1 a 2 přímo simulují situace, kde jsou patrná slabá místa stávajících politik. Ty byly pozměněny takovým způsobem, aby tato slabá místa co nejvíce eliminovaly.

První úprava využívá statistické metody relativní směrodatné odchylky a způsobuje tak agresivnější stlačení virtuálního stroje v případě, že pro toto rozhodnutí vhodný. Druhá úprava politik umožňuje MoMu se správně rozhodnout, kdy je vhodné aplikovat stlačování. Dříve byla hranice definována jako procentní část volné paměti hosta, které se ale v některých případech ukázalo jako nevhodné a balónování tak bylo prováděno předčasně.

Pro účely srovnání dopadu změn byly scénáře samostatně aplikovány na původní politiky a poté na obě změny samostatně. Nově navržené politiky v obou případech plní svůj účel – rozpoznají definované neefektivní chování a reagují na něj vhodným způsobem (rychle stlačují VM, která využívají stabilní objem paměti a spuštění balónování až když má host *skutečně* málo paměti). Dle výsledného chování MoMu lze usoudit, že v celkovém pohledu zlepšují jeho funkci.

Souběžně s MoM Simulátorem vznikl také nástroj pro generování scénářů, které jsou vstupem simulátoru. Použití generátoru není jedinou cestou jak připravit data, ale poskytuje v porovnání s dalšími možnými způsoby (např. OpenOffice Calc) řadu podstatných výhod.

Simulátor produkuje množství dat, jež mohou být vizualizovány pomoci dalšího nově vytvořeného nástroje, který je schopen sledovat *živý* průběh simulace nebo exportovat svůj výstup do externího souboru různých formátů (`eps`, `png`, …).

Dále byl vytvořen nástroj pro správu politik MoM. Ten umožňuje spravovat konstanty použité v politikách MoM a jejich uplatnění na virtuální stroje individuálně. Tímto způsobem se dá docílit i aplikování určitých částí politik pouze pro vybrané virtuální stroje.

11 Reference

Applications using libvirt [online]. 2014 [cit. 2014-12-14]. Dostupné z: <http://libvirt.org/apps.html>.

Abels T., Dhawan P., Chandrasekaran B. *An Overview of Xen Virtualization* [online]. 2005 [cit. 2015-01-01]. Dostupné z: <http://www.dell.com/downloads/global/power/ps3q05-20050191-Abels.pdf>.

Banerjee I., Guo F., Tati K., Venkatasubramanian R. *Memory Overcommitment in the ESX Server* [online]. 2013 [cit. 2014-12-14]. Dostupné z: <https://labs.vmware.com/vmtj/memory-overcommitment-in-the-esx-server>.

Berrangé, D. P. *Manage virtual machines with virt-manager* [online]. 2009 [cit. 2014-12-14]. Dostupné z: <http://virt-manager.org>.

Fediuck D. *Engine Core* [online]. 2012 [cit. 2015-01-01]. Dostupné z: <http://www.ovirt.org/images/d/d9/Ovirt-engine-core_beijing_2012.odp>.

Hagem W. *Using KVM virtualization: Getting started with kernel-based virtual machine, the latest generation of Linux virtualization techniques* [online]. 2014 [cit. 2015-01-02]. Dostupné z: <http://www.ibm.com/developerworks/library/l-using-kvm/l-using-kvm-pdf.pdf>.

Hagen W. *Professional Xen virtualization*. Indianapolis: Wiley Publishing, 2008, xxiii, 405 s. ISBN 978-0-470-13811-3.

Hendl J. *Přehled statistických metod: analýza a metaanalýza dat*. Vyd. 4. rozš. a přeprac. vyd.. Praha: Portál, 2012, 734 s. ISBN 978-80-251-2084-2.

IBM Corporation *Best practices for KVM* [online]. 2012 [cit. 2014-12-14]. Dostupné z: <http://www-01.ibm.com/support/knowledgecenter/api/content/nl/en-us/linuxonibm/liaat/liaatbestpractices_pdf.pdf>.

Jelínek L. *Jádro systému Linux: kompletní průvodce programátora*. Vyd. 1. Brno: Computer Press, 2008, 686 s. ISBN 978-80-251-2084-2.

Jones R. *Virtio balloon* [online]. 2010 [cit. 2014-12-14]. Dostupné z: <http://rwmj.wordpress.com/2010/07/17/virtio-balloon/>.

Kolovson C., Muir S., Tavilla R. *VMware technical journal* [online]. 2013 [cit. 2014-12-31]. Dostupné z: <http://www.researchgate.net/publication/237088461_An_Anomaly_Event_Correlation_Engine_Identifying_Root_Causes_Bottlenecks_and_Black_Swans_in_IT_Environments/file/e0b4951b7361f47e94.pdf>.

KUSNETZKY, D. *Virtualization - Trend or Buzzword?* [online]. 2009 [cit. 2014-12-14]. Dostupné z: <http://www.zdnet.com/article/virtualization-trend-or-buzzword>.

MAGENHEIMER, D. *Transcendent Memory ("tmem"): a new approach to physic* [online]. 2009 [cit. 2014-12-29]. Dostupné z: <http://old-list-archives.xenproject.org/archives/html/xen-devel/2009-01/msg00253.html>.

MAGENHEIMER, D. *Transcendent Memory on Xen* [online]. 2009 [cit. 2014-12-29]. Dostupné z: <http://www-archive.xenproject.org/files/xensummit_oracle09/xensummit_transmemory.pdf>.

MENAGE, P. *CGROUPS* [online]. 2014 [cit. 2014-12-14]. Dostupné z: <https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt>.

OpenStack: The Open Source Cloud Operating System [online]. 2014 [cit. 2014-12-14]. Dostupné z: <http://www.openstack.org/software/>.

PARISEAU B. *KVM reignites Type 1 vs. Type 2 hypervisor debate* [online]. 2011 [cit. 2014-12-14]. Dostupné z: <http://searchservervirtualization.techtarget.com/news/2240034817/KVM-reignites-Type-1-vs-Type-2-hypervisor-debate>.

REDHAT *Chapter 2. Memory allocation* [online]. 2014 [cit. 2015-01-02]. Dostupné z: <https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_MRG/1.3/html/Realtime_Reference_Guide/chap-Realtime_Reference_Guide-Memory_allocation.html>.

REDHAT *Category:Vdsm* [online]. 2012 [cit. 2015-01-01]. Dostupné z: <http://www.ovirt.org/Vdsm>.

RFC 2212 Specification of Guaranteed Quality of Service [online]. 2009 [cit. 2014-12-14]. Dostupné z: <http://tools.ietf.org/html/rfc2212>.

RFC 4122 A Universally Unique IDentifier (UUID) URN Namespace [online]. 2005 [cit. 2014-12-14]. Dostupné z: <http://www.ietf.org/rfc/rfc4122.txt>.

RFC 5531 RPC: Remote Procedure Call Protocol Specification Version 2 [online]. 2009 [cit. 2014-12-14]. Dostupné z: <http://tools.ietf.org/html/rfc5531>.

RAFFIC M. *VMware Memory Management Part 3 – Memory Ballooning* [online]. 2009 [cit. 2014-12-14]. Dostupné z: <http://www.vmwarearena.com/wp-content/uploads/2014/05/VMware-Memory-Terminology.jpg>.

RICHARDSON L., RUBY S. *RESTful web services*. 1st ed. Sebastopol: O'Reilly, 2007, xxiv, 419 s. ISBN 978-0-596-52926-0.

SYMANTEC CORPORATION *copy-on-write* [online]. 2014 [cit. 2014-12-14]. Dostupné z: <http://www.symantec.com/security_response/glossary/define.jsp?letter=c&word=copy-on-write>.

TAKEMURA C., CRAWFORD L. *The book of Xen: a practical guide for the system administrator*. San Francisco: No Starch Press, 2010, xxiv, 281 p. ISBN 15-932-7186-7.

TATABOJS *Nanoalbum* [online]. 2004 [cit. 2014-12-14]. Dostupné z: <http://tatabojs.cz/tatabojs/OLD/OLD2/neweb/nanoalbum.htm>.

TORVALDS, L. *CPU Accounting Controller* [online]. 2010 [cit. 2014-12-14]. Dostupné z: <https://www.kernel.org/doc/Documentation/cgroups/cpuacct.txt>.

VMWARE *Understanding Memory Resource Management in VMvare® ESX™ Server* [online]. 2009 [cit. 2014-12-14]. Dostupné z: <http://www.vmware.com/files/pdf/perf-vsphere-memory_management.pdf>.

VMWARE *Understanding Full Virtualization, Paravirtualization, and Hardware Assist* [online]. 2007 [cit. 2015-01-01]. Dostupné z: <http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf>.

VMWARE *Understanding Memory Resource Management in VMware vSphere® 5.0* [online]. 2009 [cit. 2014-12-14]. Dostupné z: <http://www.vmware.com/files/pdf/mem_mgmt_perf_vsphere5.pdf>.

VMWARE *VMware ESXi* [online]. 2013 [cit. 2015-01-01]. Dostupné z: <http://www.vmware.com/products/esxi-and-esx/overview>.

VMWARE *vSphere Web Access* [online]. 2014 [cit. 2014-12-14]. Dostupné z: <http://pubs.vmware.com/vsphere-4-esx-vcenter/topic/com.vmware.vsphere.dcadmin.doc_41/vsp_dc_admin_guide/vsp_interfaces/c_vi_web_access.html>.

WATTS, S. *Virtualization: More than Just a Technology Buzzword* [online]. 2014 [cit. 2014-12-14]. Dostupné z: <http://idtus.com/blog/virtualization-and-software-testing-technology/>.

WIRZENIUS L., OJA J., STAFFORD S., WEEKS A *The buffer cache* [online]. 2004 [cit. 2014-12-14]. Dostupné z: <http://www.tldp.org/LDP/sag/html/buffer-cache.html>.

XEN PROJECT *Xen Orchestra* [online]. 2013 [cit. 2014-12-14]. Dostupné z: <http://xenproject.org/directory/directory/consulting/33-xen-orchestra.html>.