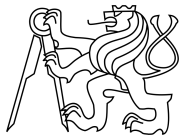
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Optimalizace paralelního zpracování cyklické fronty
Student:	Bc. Josef Kučera
Vedoucí:	Ing. Jiří Kašpar
Studijní program:	Informatika
Studijní obor:	Počítačové systémy a sítě
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Cyklická fronta CQ byla vyvinuta v předmětu MI-MCS, inspirována lock-less frontou z [1], rozšířena pro více písařů a čtenářů, realizována pomocí atomických operací architektury x64.
1. Nastudujte funkci a implementaci cyklické fronty CQ.
2. Změřte závislosti propustnosti fronty na délkách fronty a zprávy.
3. Navrhněte a implementujte další optimalizace paralelních přístupů ke sdíleným datovým strukturám fronty.
4. Změřte a vyhodnoťte přínosy jednotlivých typů optimalizací na víceprocesorovém serveru s vícejádrovými procesory architektury x64.

Seznam odborné literatury

[1] Herlihy, Maurice and Shavit, Nir: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., 2008
[2] Daniel J. Sorin, Mark D. Hill, and David A. Wood.: A Primer on Memory Consistency and Cache Coherence (1st ed.). Morgan & Claypool Publishers, 2011

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 15. února 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra počítačových systémů a sítí

Diplomová práce

Optimalizace paralelního zpracování cyklické fronty

Bc. Josef Kučera

Vedoucí práce: Ing. Jiří Kašpar

8. května 2017

Poděkování

Poděkovat bych chtěl především mému vedoucímu diplomové práce Ing. Jiřímu Kašparovi za příkladné vedení, konzultace a cenné rady, které mi pomohly v tvorbě této práce. V neposlední řadě bych chtěl poděkovat celé rodině a přítelkyni za velkou podporu během mého studia a textovou korekturu diplomové práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 8. května 2017 ……

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na tuto práci

KUČERA, Josef. *Optimalizace paralelního zpracování cyklické fronty.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Cílem této práce je představit a porovnat varianty implementace cyklické fronty *CQ* v jazyce C pro více písařů a čtenářů s paralelním přístupem pomocí zámků a atomických operací na architektuře x86-64. Následně navrhnout sadu optimalizací pro vkládání a vybírání zpráv z cyklické fronty a otestovat jejich výkonnostní přínos na základě různých parametrů, jako jsou délka fronty, velikost zprávy, způsob zpracování zpráv, počty písařů a čtenářů a jejich rozestavení na jádra procesorů.

Klíčová slova cyklická fronta, datová struktura s paralelním přístupem, lock-free, sdílená paměť, pipeline, optimalizace paměti, vícejádrové systémy

Abstract

The purpose of this work is to outline and compare variants of implementation of circular queue CQ in language C for multiple writers and readers with concurrent access using locks and atomic operations on x86-64. Next, design a set of optimizations for inserting and retrieving messages from a circular queue and measure their performance by varying parameters such as length of the queue, message size, message processing, number of writers and readers and their deployment on processor cores.

Keywords circular queue, concurrent data structure, lock-free, shared memory, pipeline, optimization cache memory, multicore systems

Obsah

Seznam obrázků

Seznam ukázek zdrojového kódu

Úvod

V dnešní době je nesmírně těžké přijít v diplomové práci s nějakým neotřelým a originálním úvodem, který by nebyl převyprávěn v téměř každé bakalářské, diplomové či dizertační práci zabývající se paralelním výpočtem nebo zpracováním – neustálé vylepšování technologie procesorů pomocí vzrůstající frekvence dorazilo na své hranice, a musela se tak hledat jiná cesta pro zvyšování výkonu. Velcí výrobci procesorů, jako jsou Intel nebo AMD, se proto zaměřili na umístění více výpočetních jader do jednoho procesoru, a vznikly tedy vícejádrové procesory.

Navzdory tomuto ustálenému klišé, které zajisté slyšel každý absolvent školy zaměřené na informační technologie, výroba vícejádrových procesorů zapříčinila vznik nepřeberného množství prací věnujících se paralelizaci výpočtu. Jedním z hlavních důvodů pro věnování se této oblasti také je, že implementovat aplikaci běžící korektně a efektivně na jednom jádře není jednoduché, ale pokud má běžet na vícejádrovém procesoru, tak náročnost implementace ještě obrovsky narůstá.

Nejvíce problémů při paralelizaci aplikace leží ve správném návrhu a také především v efektivním využívání sdílených datových struktur. Proto se programátoři často uchylují k jednoduchému konzervativnímu řešení, kdy se celá sdílená datová struktura zamkne a úpravy může provádět v jednom čase pouze jedno vlákno. Takový způsob následně vede při častém zamykání

k vytvoření „úzkého hrdla“, které může být ještě umocněno zvyšujícím se počtem vláken.

Tato práce se zaměřuje právě na jednu ze základních datových struktur – cyklickou frontu, která se v práci optimalizuje pro sdílený přístup z více vláken na architektuře x86-64. Nejdříve jsou vysvětleny všechny nezbytné teoretické pojmy související s implementací cyklické fronty. V další části je popsána již samotná implementace cyklické fronty *CQ* založená na jednoduché cyklické frontě od Maurice Herlihy a Nir Shavit [1] s doplněním podpory více písařů a více čtenářů a s rozdělením sdílených ukazatelů. Dále jsou porovnány její varianty implementované pomocí zámků (lock-based) vůči variantě bez zámků pouze s atomickými instrukcemi (lock-free). Součástí stejné kapitoly je taktéž analýza dostupných projektů podobných *CQ*. Následně jsou navrženy optimalizace cyklické fronty pro efektivnější využívání cache paměti a výpočetního času s představením jejich úkolů a předpokládaných přínosů. U všech navržených a implementovaných optimalizací je v poslední kapitole při přístupech z více vláken změřen jejich vliv na rychlost a propustnost cyklické fronty v závislosti na různých parametrech a je okomentováno, zda skutečně došlo k očekávaným vylepšením či nikoliv.

Během návrhu, implementace a testování se primárně práce zabývá scénářem, kdy testovací aplikace intenzivně využívá cyklickou frontu z více vláken a způsobuje její maximální vytížení. Pro ostatní případy, kdy využívání cyklické fronty není tak vysoké, poskytují většinou „naivní“ implementace dostatečnou propustnost, a proto takové situace nejsou v návrzích a testech optimalizací zohledňovány.

Kapitola 1

Analýza

První část této kapitoly analyzuje různé vzorové modely paralelizací, se kterými lze počítat ve světě vícejádrových systémů, a podrobně představuje jeden z nich, který je stěžejní pro tuto práci. Druhá část popisuje základní vlastnosti sdílených datových struktur a možné způsoby práce s nimi. Zároveň obsahuje představení některých druhů synchronizačních technik a vysvětlení různých způsobů zajištění postupu výpočtu, které lze použít při paralelním přístupu do sdílených datových struktur. Součástí analýzy je také krátké vysvětlení významu cache pamětí v procesoru spolu s důležitostí jejich koherenčních protokolů. V závěru je také uveden problém konzistence paměti ve vícejádrovém systému vyskytující se v implementaci cyklické fronty *CQ* spolu se specifickými vlastnostmi architektury x86-64, pro kterou je *CQ* primárně určena.

1.1 Vzory paralelního zpracování

Při vývoji paralelních systémů se po analýze problematiky a možnosti paralelního zpracování může dospět k použití některého z různých vzorů pro paralelní zpracování. Většinou to nejsou hotové předpřipravené vzory, které lze přímo transformovat do zdrojového kódu, ale spíše jde o popis nebo šablonu sloužící pro vyřešení problému s paralelizací zpracování. Vzory lze také formalizovat jako „best practices“, kterých se může programátor či analytik držet při vytváření designu aplikace nebo systému. Vždy je

důležité vypozorovat, jak je možné celkový problém rozdělit do menších částí, jaký tok mají data v algoritmu a jaké mají mezi sebou závislosti, například hierarchickou, sekvenční nebo žádnou. Velmi často se problém paralelizace nevyřeší pouze uplatněním jednoho ze vzorů, ale až jejich společnou kombinací.

Nejčastější vzory pro paralelní běh algoritmů lze rozdělit do tří různých kategorií s podskupinami [2]:

- *paralelizace řízená úlohami*
  - nezávislé (lineární) – Do této kategorie spadají algoritmy, které lze rozdělit na samostatné úlohy (parallel tasks). Tento vzor pomáhá programátorovi kontrolovat mapování úloh na výpočetní prostředky a vypořádat se s některými závislostmi mezi nimi. Ve výjimečném případě se může stát, že všechny úlohy jsou kompletně mezi sebou nezávislé – „embarrassingly parallel tasks“, tedy překvapivě či trapně paralelní úlohy.
  - závislé (rekurzivní) – Paralelizace typu „rozděl a panuj“ (divide and conquere) se uplatňuje v situacích, kdy jsou úlohy rekurzivní a vznikají z výsledku předchozího dokončení úlohy, například u velmi známého algoritmu mergesort na setřídění pole.
- *paralelizace řízená strukturou dat*
  - nezávislé segmenty (lineární) – Zástupcem této oblasti je vzor nazvaný geometrická dekompozice, která se používá v případech, kdy lze data rozdělit do oddělených částí a výsledek v jedné části vzniká použitím dalších několika okolních částí. Vzor pomáhá programátorovi dobře organizovat výpočet a zajistit, že potřebná data leží přesně tam, kde jsou skutečně potřeba. Příkladem může být násobení matic.
  - závislé segmenty (rekurzivní) – Tyto vzory se použijí při pohledu na data, kde nejde přímočaře použít přístup „rozděl a panuj“ z povahy algoritmu. Typické pro tento problém je časté sdílení

výsledků po každém kroku algoritmu. Příkladem pro použití takového vzoru může být problém, kdy máme les stromů a hledáme pro každý uzel jeho kořen.

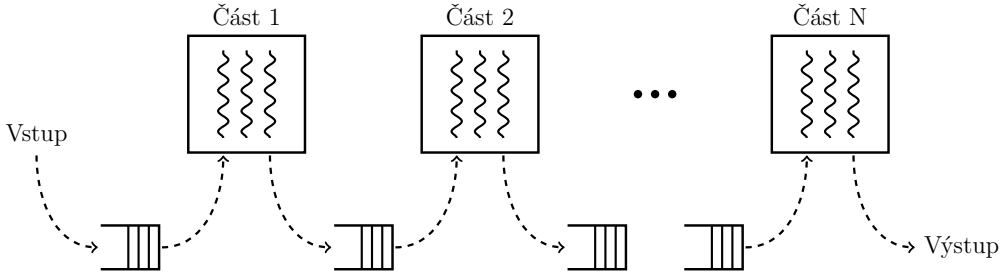
- *paralelizace řízená tokem dat*

  - řízené statickým tokem dat – Tento vzor se uplatňuje, pokud lze algoritmus výpočtu nad daty přeorganizovat do několika po sobě následujících kroků. V případě statického toku dat, který proteče vždy všemi částmi a žádnou z nich nevynechá, se používá pipeline model (roura, řetěz nebo kolona). Příkladem použití může být různorodé filtrování dat, zpracování telemetrických dat z radarů nebo zpracování signálu v reálném čase.
  - dynamický tok dat – V případě nepravidelné a asynchronní obdoby pipeline modelu, kdy data nemusí procházet všemi částmi a mohou některé vynechat, se jedná o paralelizaci řízenou událostmi („event-based coordination“). Zde ovšem oproti statickému toku dat dochází k častějšímu problému se synchronizací kroků a vyhnutí se zahlcení v některé z částí.

Posledním z vyjmenovaných paralelních vzorů je paralelizace řízená statickým tokem dat. Pipeline model rozděluje zpracování do několika etap výpočtu, přičemž mezi každou je vytvořena sdílená datová struktura (buffer, cyklická fronta) pro předávání mezivýsledků. V každé etapě může současně běžet až několik paralelních vláken pracujících s různými daty. Potenciál škálování tohoto modelu výpočtu neleží v navyšování pracujících vláken v každé z částí, ale v rozdělení výpočtu na větší a větší počet celků. Škálování aplikace s pipeline modelem je tedy především limitováno strukturou výpočetního problému, a to do jaké míry je ho možné rozdělit do samostatných částí.

Další kapitola této práce se zabývá pouze cyklickou frontou *CQ*, která je především vyvinuta a optimalizována pro paralelní model pipeline a slouží hlavně jako sdílená datová struktura pro předávání mezivýsledků mezi jeho stupni výpočtu.

Obrázek 1.1: Pipeline model paralelního výpočtu složený z N částí. Pro předávání mezivýsledků se používají sdílené fronty.



1.2 Sdílené datové struktury

Sdílené datové struktury na vícejádrových počítačích jsou používány současně z několika vláken, a proto se s nimi musí zacházet opatrně a podle předem určených pravidel. V zásadě platí, že správnost a postup výpočtu se sdílenou datovou strukturou jsou zajištěny splněním dvou základních vlastností definovaných Leslie Lamportem již v roce 1977 – živostí (liveness) a bezpečností (safety) [3].

Vlastnost bezpečnosti zajišťuje, že se neobjeví něco nesprávného nebo neočekávaného během výpočtu. Přesněji, že přechody mezi jednotlivými stavy aplikace způsobené algoritmem nesmí vytvořit nežádoucí nebo nepředpokládaný stav. Živost zase zabezpečuje, že se stále něco produktivního v aplikaci děje. Přesněji, že je posloupností stavových přechodů postupně dosahováno žádoucího stavu.

Další odstavce popisují, jak je možné docílit splnění těchto dvou vlastností pomocí využití různých synchronizačních metod – blokující nebo neblokující přístup.

1.2.1 Lock-based

Nejjednodušší způsob zabezpečení sdílených datových struktur je použití zámků. Zámky poskytují synchronizaci přístupu ke sdíleným datům pomocí vzájemného vyloučení ostatních vláken. Dalšími nástroji fungujícími na podobném blokujícím principu jsou například označení kritických sekcí

v programu nebo použití semaforů. Tyto prostředky jsou nezbytné pro zajištění bezpečnosti, protože omezují současnou modifikaci sdílených dat z více vláken. Bez tohoto vzájemného vyloučení by docházelo k neočekávaným obsahům ve společné paměti, a tedy i poškození sdílených datových struktur.

S používáním klasických zámků na ochranu sdílených prostředků je také spojeno mnoho problémů. Některé z nich jsou zde vyjmenované:

- Zámky způsobují blokování ostatních vláken, jelikož v jeden čas může pouze jedno vlákno pokračovat ve výpočtu a držet si zámek, zatímco ostatní musejí počkat, dokud se zámek opět neuvolní.
- Během implementace je nutné se zámky zacházet s opatrností. Jejich špatné používání a nevracení zpět může vyústit ve vznik deadlocku, který způsobuje kompletní zastavení postupu výpočtu v programu.
- Pokud vlákno, které momentálně drží zámek, ukončí svoji existenci, selže nebo je zablokováno, nemusí se již ostatní vlákna dočkat vrácení tohoto zámku a opět může dojít k deadlocku.
- Zámky zapřičiňující zvýšenou komunikační režii v systému, která způsobuje zahlcení a limituje možnost škálování.

1.2.2 Lock-free

Neblokující nebo lock-free algoritmy byly vyvinuty pro eliminaci některých problémů způsobených zámky. Implementace bez zámků zajišťuje, že vlákno modifikující sdílenou datovou strukturu neblokuje ostatní vlákna kvůli vzájemnému vyloučení z kritické sekce. Dalším benefitem oproti zamykání je zajištění postupu výpočtu celé aplikace, jelikož nemůže dojít k situaci, kdy některé z vláken neuvolní zámek pro zamčení jiným vláknem.

Zbavení se zámků je nejčastěji dosaženo pomocí použití hardwarových primitiv typu *read-modify-write*. Nejpoužívanější takovou operací je nejspíše *compare_and_swap*, jejíž pseudokód je vysvětlen v ukázce zdrojového kódu 1.1 a také se používá v následující kapitole o cyklické frontě *CQ*.

Compare_and_swap potřebuje celkem tři parametry. Prvním je adresa modifikované paměti, druhým je její očekávaná hodnota na této adrese a posledním je nová hodnota, která má nahradit tu stávající, pokud se shoduje s očekávanou hodnotou. Všechny tyto kroky jsou provedeny nerozdělitelně za sebou v hardwaru a žádný jiný proces nemůže do této posloupnosti kroků vstoupit. Například mezi porovnáním a uložením nové hodnoty nemůže jiný proces jakkoliv modifikovat obsah sdílené struktury.

```
bool sync_bool_compare_and_swap (address, expected, new_val) {
  if (value(address) == expected) {
    value(address) = new_val;
    return true;
  } else {
    return false;
  }
}
```

Ukázka kódu 1.1: Pseudokód popisující konstrukci atomické instrukce *compare_and_swap*.

Většina lock-free algoritmů používá *compare_and_swap* pro zajištění vlastností bezpečnosti a živosti, jelikož garantují postup výpočtu celého programu. Velmi podobné atomické operaci *compare_and_swap* je použití dvojice operací *Load-link/Store-conditional* na architekturách RISC, kde není možné v jedné instrukci pojmout 3 operandy. Na rozdíl od *Load-link/Store-conditional* ale instrukce *compare_and_swap* zajišťuje postup alespoň jednoho vlákna.

V některých případech není nutné používat komplexní operaci *compare_and_swap*, ale vlákno si může vystačit například s *fetch_and_add*, které atomicky zkopíruje hodnotu a až poté ji inkrementuje v paměti. Místo operace sčítání lze použít několik jiných operací (bitové operace, odčítání) [4] a také prohodit pořadí činností kopírování a modifikace dat. S pomocí těchto všech primitiv jsou lock-less algoritmy schopny překonat některé problémy spojené se zámky a vytvořit ve většině případů efektivnější alternativu oproti lock-based algoritmům.

Existují také algoritmy zvané wait-free, které dokonce garantují postup výpočtu na úrovni každého z vláken, jelikož u lock-free algoritmů hrozí, že se vláknu nemusí několikrát za sebou povést operace *compare_and_swap* (vždy se očekávaná a stávající hodnota liší) a může následně docházet k dočasnému hladovění vlákna. Tyto algoritmy jsou především vhodné v situacích, kde má docházet k dobrému škálování aplikace. V této práci se ovšem s tímto typem algoritmu neoperuje, jelikož není záměrem škálování na úrovni jedné instance cyklické fronty. To by například vyžadovalo až dvě vlákna navíc starající se pouze o začleňování nových zpráv od všech písařů, nebo o rozdělování zpráv všem čtenářům, jak je popsáno v diplomové práci „Scalable and Performance-Critical Data Structures for Multicores“ od Mudit Verma [5].

1.3 Koherence cache paměti

Cache paměť v procesoru je hardwarová součást počítače, která uchovává kopie dat přečtených z adresy v operační paměti, a tím může být následující přístup k těmto datům rychlejší. Výhody použití pamětí cache jsou založeny na principu lokality v čase (ke stejným datům přistupujeme opakovaně) a na lokalitě v prostoru (při přístupu k datům je velká pravděpodobnost, že se bude v krátké době přistupovat k datům v jejich blízkosti).

V současné době se nejvíce vyskytují v procesorech až tři úrovně cache pamětí. Nejmenší, nejrychlejší a nejblíže k jádru je L1 cache. O trochu vzdálenější, větší a pomalejší je L2 cache, která taktéž patří k jednomu jádru. Nejvzdálenější je L3 cache, která je již sdílená všemi jádry procesoru a většinou má velikost až několik megabajtů. Každý výrobce ještě používá různé typy vnoření dat. Výrobce procesoru Intel například používá *inkluzivní* typ. To znamená, že vyšší úroveň cache paměti obsahuje všechna data z nižších menších úrovní. Na druhou stranu výrobce procesorů AMD zase používá *exkluzivní* typ, kdy vyšší úroveň cache paměti neobsahuje žádná data z nižších úrovní.

Koherencí dat v hierarchii cache pamětí se rozumí transparentní synchronizace obsahu různých pamětí cache při paralelním přístupu k téže

buňce. Pro implementaci koherence cache paměti bylo vyvinuto několik velmi podobných protokolů založených na stavovém automatu (MEI, MESI, MOESI, MESIF). Každé z písmen v názvu protokolu indikuje možný stav jedné řádky cache paměti (modified, invalid, exclusive, shared, owned, forwarded). Mezi těmito stavy se přechází kvůli paralelním operacím čtení a zápisu do stejné řádky cache paměti.

Každý protokol pro koherenci cache paměti musí ale splňovat dva základní invarianty. To jsou podmínky, které musí být splněny po celou dobu běhu programu [6]:

- *Single-Writer, Multiple-Read (SWMR) Invariant* – Pro každý cache řádek A v libovolném daném (logickém) čase smí existovat buď jediné jádro, které může zapisovat do řádku A a také ho číst, anebo více jader, které smějí řádek A pouze číst. Tento invariant lze také vyjádřit tak, že život každého řádku cache paměti je rozdělen do epoch, kde v každé z nich může mít buď jedno jádro přístup *read-write*, nebo více jader *read-only* přístup.

- *Data-Value Invariant* – Obsah cache řádky A je na začátku epochy stejný jako na konci poslední read-write epochy.

Dodržením těchto dvou invariantů při implementaci některého z protokolu pro hierarchii cache se zajistí její „transparentnost“ – koherence se týká všech cache pamětí procesoru, ale již ne samotné architektury. Neovlivňuje tedy provádění instrukcí jinak než na rychlosti přístupu do paměti. Na druhou stranu pouze splnění dvou invariantů koherence pro zajištění celkové konzistence paměti nestačí. Koherence zajišťuje řízení přístupu do každého cache bloku zvlášť a interakci mezi různými bloky neřeší. Reálné programy totiž pracují s proměnnými v různých cache blocích a koherenční protokol mezi nimi nevidí vztahy, například datový ukazatel do jiné části paměti. Implementace konzistence paměti velmi často používá koherenci cache pamětí jako stavební blok a již se nestará o její realizaci a druh protokolu.

1.4 Konzistence paměti

Ve vícejádrovém systému se sdílenou pamětí je nutné zajistit při paralelních přístupech do paměti konzistentní pohled na její obsah. Proto je nezbytné, aby implementace sdílené paměti zaručila naplnění některého z formálně definovaných modelů konzistence paměti a programátor se jí mohl držet při vývoji aplikace. Nejintuitivnějším a nejpřísnějším modelem je *striktní konzistence*, která zaručuje, že jakékoliv čtení z adresy X vrátí uloženou hodnotu při posledním zápisu do adresy X. Jelikož mají být všechny zápisy okamžitě viditelné, je pro tento model nutné absolutní časové uspořádání, které je ve vícejádrovém systému se sdílenou pamětí velmi těžce dosažitelné. Proto byly v literatuře navrženy reálnější modely pro vícejádrové systémy. Účelem paměťového modelu je, aby bez ohledu na všechna možná zpracování paralelních vláken výpočtu byl výsledek vždy správný, ale nemusí být pokaždé stejný. V roce 1979 představil Leslie Lamport [7] model *sekvenční konzistence*, který je jedním z nejšíře přijatých modelů.

Model *sekvenční konzistence* zajišťuje, že výsledky jakéhokoliv provedení operací jsou stejné, jako kdyby byly všechny operace všech procesorů provedeny v nějakém sekvenčním pořadí a operace každého procesoru jsou v této sekvenci v pořadí specifikovaném programem. *Sekvenční konzistence* je o něco slabší model konzistence oproti *striktní konzistenci*, avšak je snadno implementovatelná a jednoduchá pro používání v paralelních aplikacích. Jestliže procesy běží na různých procesorech, je povoleno libovolné prokládání jejich instrukcí, avšak s podmínkou, že všechny procesy vidí stejné pořadí změn paměti.

Při vývoji víceprocesorových systémů se *sekvenční konzistence* ukázala jako velmi neefektivní, a proto vzniklo mnoho dalších slabších konzistencí paměti, kde se pohled na pořadí čtení a zápisů může výrazně lišit od normální intuice, například *kauzální konzistence* nebo *slabá konzistence*. Systémy s takovými zjednodušenými konzistentními modely poskytují speciální strojové instrukce – bariéry, které vynutí specifikované pořadí operací zápisu a čtení a zabrání jejich posunutí přes tuto bariéru. Záměrem je, aby mohly být vidět paměťové operace konzistentně v nějakém

očekávaném pořadí i jinými procesory. Tyto instrukce musejí být používány pouze ve vhodných situacích, protože jejich nadměrné použití výrazně snižuje možnost optimalizovat pořadí přístupů do paměti.

Každá z architektur vícejádrových systému má implementovanou jinou konzistenci paměti, konkrétně architektura x86-64, pro kterou je vytvořena implementace cyklické fronty *CQ*, používá model zvaný *total store order* [8]. Tento model se velice přibližuje k silné *sekvenční konzistenci* a také bývá zařazen do skupiny silných modelů konzistencí paměti. Jeho podstatným rozdílem je možnost upravit posloupnost operací čtení se staršími operacemi zápisu, pokud se netýkají stejné adresy v paměti („Store-Load ordering“). V *sekvenční konzistenci* není možné upravovat pořadí žádných operací zápisu nebo čtení. Velice pěkný popis a definice všech pravidel s podrobnými ukázkami, co vše je možné u konzistence *total store order* očekávat, jsou dostupné z oficiálního dokumentu „Intel® 64 Architecture Memory Ordering White Paper“ [8].

Kapitola 2

Vlastnosti implementace *CQ*

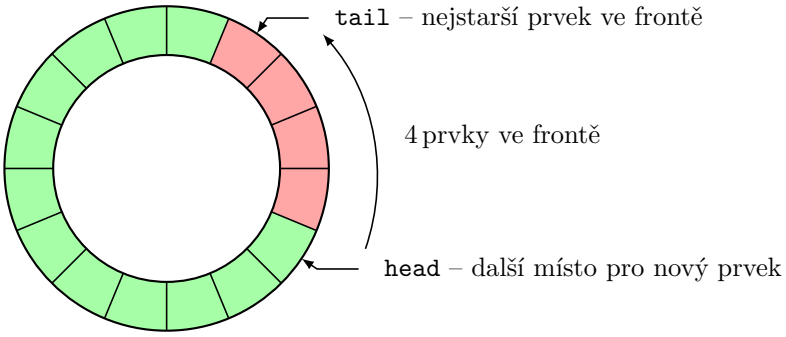
Fronty jsou fundamentální datové struktury ve výpočetních scénářích s modelem producent – konzument nebo pipeline. Základní dvě operace nad každou frontou jsou *enqueue* a *dequeue* a bývají vysvětlovány již v základních kurzech programování. *Enqueue* operace zařadí nejnovější prvek na konec fronty a *dequeue* naopak vyzvedne nejstarší prvek ze začátku fronty. Aby se zabránilo nekonzistencím a chybám, operace se sdílenou frontou musejí být celistvé (atomické) – buď se provedou celé, nebo vůbec, ale nic mezi tím. Pokud narůstá počet vláken přistupujících ke sdílené frontě, výkon celého systému klesá právě kvůli jejich vzájemnému boji o sdílené prostředky. Je tedy nutné vymýšlet a implementovat stále efektivnější koordinace paralelních přístupů.

Cyklická fronta je jedním z mnoha způsobů, jak lze implementovat sdílenou frontu pro paralelní přístup. Jejím základem je zacyklené pole, ve kterém se po zápisu na posledním prvku opět pokračuje od prvního, pokud již byl mezitím odebrán, a tvoří se tak dojem nekonečného pohybu. K fungování cyklické fronty je v nejjednodušším případě potřeba si pamatovat pouze dva ukazatele – kde se právě nachází začátek a konec fronty. Pokud ukazují na stejnou pozici v poli, je fronta prázdná, pokud by se takového stavu mělo dosáhnout po vložení dalšího prvku, pak naopak dochází k tomu, že do plné fronty se již další prvek nevejde. Oproti jiným implementacím front má cyklická fronta omezenou velikost a jejím dalším

benefitem je, že se nemusí posouvat její obsah, ale pouze ukazatelé, jak je patrné z obrázku 2.1.

Vždy je nutné si také ujasnit, co vlastně znamená začátek a konec fronty. V této práci se ukazatel `head` považuje za začátek fronty a označuje místo, kam písaři vkládají nové zprávy. Na druhý konec analogicky ukazuje `tail`, odkud čtenáři vyzvedávají nejstarší zprávy. V jiných pracích a implementacích cyklických front může být význam těchto ukazatelů obrácený.

Obrázek 2.1: Diagram s ukázkou základního a nejrozšířenějšího modelu cyklické fronty se dvěma ukazateli na začátek a konec fronty.



Tato kapitola představuje dvě varianty implementace cyklické fronty *CQ* v jazyce C, které vznikly během výuky magisterského předmětu Vícejádrové systémy (MI-MCS) na FIT ČVUT. První z nich používá pro koordinaci paralelních přístupů klasické zámky (lock-based varianta) a druhá pouze atomické instrukce (lock-free varianta). Následně se v měřeních těchto dvou variant cyklických front ukazují kladné nebo záporné stránky každé z nich a na základě naměřených výsledků se od lepší varianty odvíjí i další kapitoly v této práci. V závěru této kapitoly je také analýza jiných dostupných cyklických front podobných *CQ*.

V textu práce se velice často vyskytuje kurzívou zkratka *CQ*, která označuje konkrétní implementaci cyklické fronty vzniklou během výuky předmětu na FIT ČVUT, a neznamená tedy pouze zkratku pro „circular queue“.

2.1 Vlastnosti

Následující odstavce popisují společné vlastnosti obou variant implementací cyklické fronty *CQ* a všechny důležité implementační detaily, které jsou podstatné pro vysvětlení celé funkčnosti *CQ*. Hlavním záměrem pro vytvoření této neobvyklé implementace cyklické fronty je její použití jako stavebního kamene pro výpočetní model proudového zpracování ve více stupních, což si lze představit jako spojení výpočetních kroků do série a mezi každým krokem dochází k předávání mezivýsledků pomocí front.

2.1.1 Konfigurace *CQ*

Během inicializace je nutné specifikovat, kolik písařů a čtenářů má přistupovat do cyklické fronty a v jakém módu má probíhat čtení zpráv při vícero čtenářích. Výsledkem je výběr celkem ze čtyř konfigurací pokrývající všechny možné počty písařů a čtenářů:

- *W1R1* – Základní konfigurace cyklické fronty, kde se vyskytuje pouze jeden čtenář a jeden písař.
- *WnR1* – Varianta pro více písařů a pouze jednoho čtenáře.
- *W1Rnx* – Varianta pro jednoho písaře a více čtenářů.
- *WnRnx* – Komplexní konfigurace pro více písařů a více čtenářů.

Pro konfiguraci s více čtenáři je také nutné vybrat způsob, kterým se mají zprávy zpracovávat (označené písmenem x v předešlém seznamu konfigurací). Na výběr je ze tří možností:

- *1=1* (*single*) – Základní varianta, kdy každou vloženou zprávu přečte pouze jeden čtenář a po přečtení je odstraněna.
- *N* (*broadcast*) – Každá vložená zpráva musí být přečtena všemi čtenáři a až při posledním přečtení je odstraněna.

- *B* (*barrier*) – Velice podobný režim čtení jako *broadcast*, ale jak z názvu vyplývá, tak všichni čtenáři na sebe čekají na vytvořené bariéře, dokud zprávu nepřečte i nejpomalejší z nich a až poté se může pokračovat se zpracováním další zprávy.

Podle zvolené konfigurace při inicializaci pak *CQ* vybere nejefektivnější způsob vkládání a vyzvedávání zpráv. Pro každou konfiguraci jsou připravené optimalizované procedury, které nemusí obsahovat všechny potřebné techniky pro synchronizaci přístupů, jako například u konfigurace *WnR1*, kdy jeden čtenář nepotřebuje žádné zámky nebo atomické operace pro vyzvednutí zprávy, a tudíž operace mohou probíhat mnohem efektivněji. Písaři a čtenáři pak následně používají funkční ukazatele, které se při inicializaci podle vstupní konfigurace nastaví na správné optimalizované procedury.

2.1.2 Datové struktury

Základem celé implementace *CQ* je klasické pole obsahující ukazatele na jinou datovou strukturu `CQentry`, která slouží jako obalovací struktura a je tvořena pouze dvěma položkami – ukazatelem na buffer pro uložení zprávy a počítadlem zbývajících přečtení bufferu čtenáři. V případě režimu čtení *single* se používá jednodušší datová struktura `CQsimpleentry`, jelikož neobsahuje počítadlo zbývajících přečtení, které je v tomto případě módu čtení nevyužité.

Cyklická fronta využívající atomické instrukce byla popsána například již v knize „The Art of Multiprocessor programming“ od Maurice Herlihy a Nir Shavit [1], ale podporovala pouze jednoho písaře a čtenáře. Pro rozšíření této cyklické fronty na podporu více písařů a čtenářů muselo dojít k rozdělení na šest různých ukazatelů místo klasických dvou zažitých ukazatelů reprezentujících začátek a konec fronty. Díky tomuto vylepšení je stále možné používat atomické instrukce pro vkládání a vybírání zpráv a také dochází ke zkrácení času v kritické sekci, která je v jiných situacích řízena pomocí zámků. Bez tohoto rozdělení na šest různých ukazatelů by nebylo možné používat atomické instrukce při paralelních přístupech z více vláken.

Přehled všech rozdělených sdílených ukazatelů uvnitř cyklické fronty *CQ*:

Ukazatelé na začátku cyklické fronty

- *allocated* – Ukazatel reprezentuje první volné místo na vložení nového záznamu do cyklické fronty.
- *queued* – Ukazatel na poslední záznam, u kterého se zapisují data do bufferu (`CQentry`).
- *head* – Ukazatel na poslední záznam, u kterého se právě zapisuje adresa naplněného bufferu (`CQentry`).

Ukazatelé na konci cyklické fronty

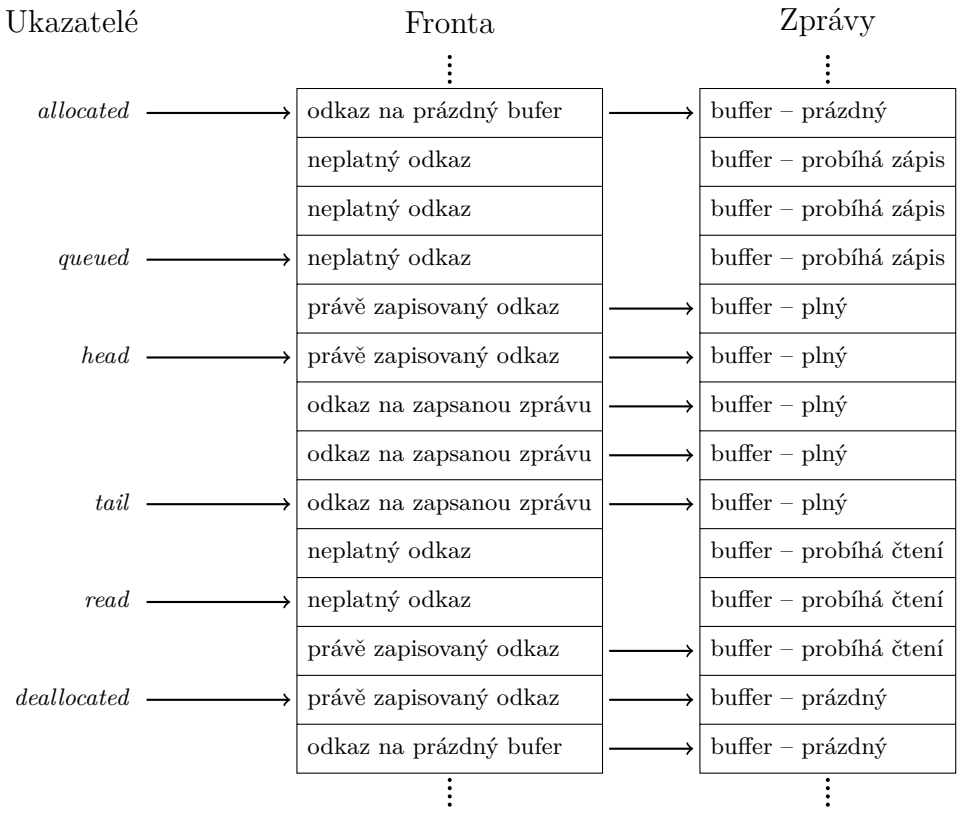
- *tail* – Ukazatel na první nepřečtený (nejstarší) záznam v poli.
- *read* – Ukazatel na poslední záznam, u kterého dochází ke čtení dat z bufferu (`CQentry`).
- *deallocated* – Ukazatel reprezentující místo v poli, kde probíhá poslední zápis adresy přečteného (vyprázdněného) bufferu v `CQentry`.

2.1.3 Použití *CQ*

Použití cyklické fronty *CQ* je velice přímočaré a snadné. Písaři nebo čtenáři musejí využívat pouze několik málo různých funkcí s velice jednoduchým rozhraním. Celý životní cyklus *CQ* začíná inicializací pomocí procedury `CQ_init`. Každý z čtenářů nebo písařů musí nejprve zavolat proceduru `open`, která vrací vytvořenou datovou strukturu `CQhandle`. Díky ní se každé z vláken identifikuje při vkládání nebo vybírání z fronty. Zároveň také obsahuje některé lokální kopie ze sdílených ukazatelů, jejichž počet se v následující kapitole ještě rozšiřuje.

Dále následuje část s vkládáním zpráv písaři a vybíráním zpráv čtenáři a životní cyklus zase končí zavoláním procedury `CQ_close`, při které dochází k odstranění všech vytvořených datových struktur.

Obrázek 2.2: Diagram s vysvětlením funkcionality všech rozdělených ukazatelů uvnitř cyklické fronty *CQ*.




```
typedef struct {
  struct CQ *cq;      // pointer to one instance of CQ
  long rdwr;          // 0 = reader, 1 = writer
  long id;            // ID of reader or writer
  long cpu;           // ID of fixed CPU core
  long head;          // local copy of head pointer
  long tail;          // local copy of tail pointer
  long deallocated;   // local copy of deallocated pointer
} CQhandle;
```

Ukázka kódu 2.1: Ukázka datové struktury `CQhandle`, kterou vlastní každý čtenář nebo písař a používá ji pro vkládání a vybírání zpráv ze *CQ*.

2.1.4 Vložení prvku do fronty

Jak bylo zmíněno v předcházející části, *CQ* obsahuje místo klasických dvou ukazatelů celkem šest. To umožňuje rozdělit operaci vložení zprávy na získání bufferu s rezervováním místa a na zařazení zprávy do fronty. Druhou možností je vložit zprávy klasickým způsobem, tedy pomocí jediného zavolání procedury `writeMsg` zařadit připravený buffer bez předchozí rezervace místa a obdržení bufferu.

Pokud písař chce využívat vkládání pomocí dvou fází, tak nejprve musí zavolat proceduru `getBuffer`, která vrací ukazatel na připravený buffer pro zápis a také posune sdílený ukazatel *allocated* na další položku. Písař si ještě při získávání bufferu může vybrat, zda má procedura počkat na uvolnění místa při zahlcení fronty, nebo vrátit ihned hodnotu `NULL`, a tudíž k žádnému vyčkávání na volné místo nedojde. Pro druhou, nečekající variantu je připravena jiná procedura `getBufferNW` (`NW` označuje anglický výraz „no wait“).

Do obdrženého bufferu si může písař libovolně dlouho vkládat data. Po dokončení zapisování do bufferu musí písař zavolat proceduru `putMsg`, pomocí které zařadí buffer již s jistotou, že fronta vložení zprávy neodmítne z důvodu nedostatečné kapacity volných míst.

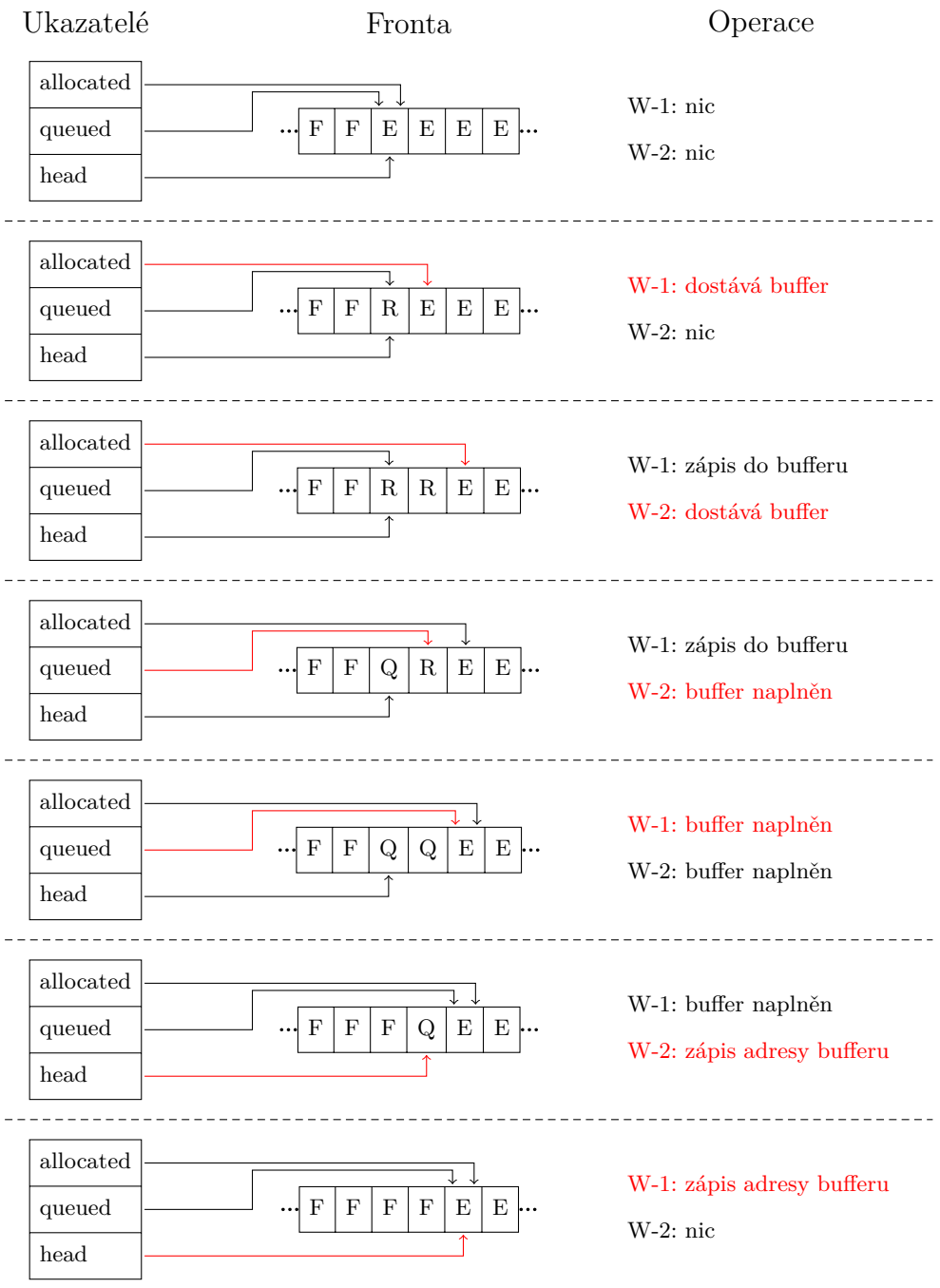
Celý rozdělený proces vkládání nových zpráv od dvou písařů označených jako *W-1* a *W-2* je podrobně rozkreslen na obrázku 2.3.

2.1.5 Odebrání prvku z fronty

Odebrání prvku z fronty probíhá v podstatě podobným mechanizmem jako vkládání. Čtenář si může zase vybrat mezi rozdělením na dvě fáze nebo zavoláním jediné procedury `readMsg`, která obsah zprávy překopíruje do předaného bufferu vytvořeného čtenářem a odstraní ji z fronty nebo dekrementuje její počítadlo zbývajících přečtení zprávy.

Pokud čtenář zvolí rozdělené vybírání zpráv z fronty, musí nejprve zavolat funkci `getMsg`, která vrací ukazatel na buffer k přečtení. Pokud ve frontě neexistuje žádná zpráva, kterou by mohl čtenář přečíst, tak aktivně vyčkává, dokud se některá nová neobjeví ve frontě. Tak jako u vkládání, i zde

Obrázek 2.3: Diagram s postupem vkládání nových zpráv a právě prováděných operací od písařů *W-1* a *W-2*. Červená barva označuje posun ukazatele spolu s iniciující operací. Písmena uvnitř fronty označují stav, ve kterém se právě daný ukazatel v poli nachází (E – volné místo, R – místo rezervováno, Q – zapisování adresy, F – obsazeno).



existuje funkce bez čekání `getMsgNW` vracející ukazatel na `NULL` v případě prázdné fronty.

Čtenář si může libovolně dlouho pracovat s bufferem a po dokončení čtení musí zavolat funkci `putBuffer`, kterou opět zařadí přečtený buffer zpět do fronty, a tím dovolí využít místo pro novou zprávou od písaře.

Jak bylo zmíněno v předešlém textu, tak fronta umožňuje čtenářům číst zprávy v různých módech. Základním je *single*, kde se zpráva odstraní ihned po přečtení. Druhým je *broadcast*, u kterého funkce na vrácení přečteného bufferu `putBuffer` pouze dekrementuje počítadlo zbývajících přečtení uvnitř struktury `CQentry` a pokud se dostane až na nulu, tak je zpráva odstraněna. Posledním módem je *barrier*, kde na sebe všichni čtenáři počkají uvnitř funkce `putBuffer` a zpráva je vymazána až po přečtení nejpomalejším čtenářem.

2.1.6 Další implementační detaily

V této části jsou rozebrány a vysvětleny některé důležité implementační detaily, které se vyskytují v implementaci cyklické fronty *CQ*.

2.1.6.1 Logické operace

I když se přesná velikost cyklické fronty předává parametrem při inicializaci, tak realita je jiná. Pro efektivnější využívání fronty a kontrolu cykličnosti se velikost zvětší až na nejbližší mocninu dvojky. Ukazatelé v této implementaci *CQ* jsou pouze celočíselné indexy do pole a inkrementují se stále dokola. V této implementaci ale přetečení hodnoty přes maximální hodnotu datového typu `long` nevadí, jelikož plnost fronty se kontroluje přes vzdálenost ukazatelů `allocated` a `deallocated` od sebe. Díky tomu je také možné odstranit „drahou“ operaci modulo (počítání zbytku po dělení) a kontrolu cykličnosti provádět až při vkládání nebo vybírání zpráv pouze nad svou lokální kopií sdíleného ukazatele pomocí logické operace *AND*, která zpravidla zabere několikanásobně méně instrukčních cyklů než dělení.

Jelikož kontrolu cykličnosti je nutné provádět při každém vkládání nebo vybírání zprávy z fronty, tak se nejprve při inicializaci *CQ* připraví bitová

maska z navýšené velikosti fronty na mocninu dvojky, pomocí které vždy vlákno provádí operace *AND*. Výkonnostní přínos tohoto nahrazení logickou operací *AND* místo operace modulo je velmi znatelný, jak je patrné z měření v článku „Modulo and Division vs Bitwise Operations“.[9]

2.1.6.2 Volatile proměnné

Na vícejádrovém procesoru má každé jádro vlastní sadu registrů, a proto se v nich kopie hodnot sdílených proměnných vyskytují, pokud k nim vlákno přistupuje (čtení i zápis). Jestliže ke sdílené proměnné přistupuje více vláken a některé z nich upraví obsah, tak se vyskytuje možnost, že jiné vlákno může stále vidět starou hodnotu uloženou ve svých registrech. Toto je poměrně důležitá část v implementaci *CQ*, jelikož se vyskytuje hned několik proměnných, které jsou velice často upravovány a čteny z různých vláken.

```
#define SHARED volatile
⋮
  SHARED long allocated;
  SHARED long queued;
  SHARED long head;
  SHARED long tail;
  SHARED long read;
  SHARED long deallocated;
⋮
```

Ukázka kódu 2.2: Použití direktivy `volatile` pro načítání hodnoty z hlavní paměti.

Direktiva `volatile` (v překladu nestálý) u proměnné zajišťuje, že si vlákno při každém čtení takto označené proměnné vyžádá nejnovější kopii až z paměťového subsystému (cache paměť, hlavní paměť) místo použití své lokální kopie v registrech. Direktiva tedy zajišťuje, že kompilátor neudržuje (neoptimalizuje) lokální kopie proměnných v registrech procesoru déle, než skutečně potřebuje k realizaci jednoho příkazu.

Ke sdíleným ukazatelům v instanci *CQ* se neustále přistupuje z několika vláken, a proto jsou všechny označeny jako `volatile`, aby se eliminovaly chyby kvůli použití dříve načtených již neaktuálních hodnot v registrech.

2.2 Představení variant *CQ*

CQ je od začátku implementována v lock-less variantě pomocí atomických instrukcí. Ovšem kvůli dostatečnému porovnání výkonnosti byla připravena i druhá lock-based varianta používající jeden a čtyři zámky. Jejich výkonnostní porovnání lze vidět v části o srovnání variant *CQ*.

2.2.1 Lock-based varianta

Jediným synchronizačním mechanizmem u lock-based varianty jsou zámky implementované pomocí proměnných typu `pthread_mutex_t` ze standardu POSIX [10], jelikož pro vícero čtenářů a písařů se v *CQ* používají POSIXová vlákna. V nejjednodušším řešení lze použít pouze jeden zámek na celou frontu, ovšem při vkládání nové zprávy je zbytečné zamezovat čtenářům přístup na opačné straně. *CQ* používá několik ukazatelů u začátku a konce fronty, takže pouze jeden zámek pro všechny ukazatele by nutně a zbytečně blokoval i ostatní. Pro synchronizaci se tedy používají celkem čtyři zámky, vždy po dvou na začátku a na konci fronty. Pro srovnání výkonnosti s lock-less variantou jsou připraveny obě verze, tedy s jedním i čtyřmi zámky.

Během všech operací vkládání a vybírání zpráv z fronty je nejprve před modifikací uzamčen příslušný zámek k ukazateli a po úpravě zase uvolněn. To poskytuje mezivláknově bezpečný mechanizmus, který ovšem může způsobit problém při případném neuvolnění zámku a ostatní vlákna by se již teoreticky nikdy nedostala k modifikaci ukazatele.

2.2.2 Lock-less varianta

Jak již vyplývá z názvu, tak lock-less nebo lock-free varianta neobsahuje žádné zámky pro synchronizaci přístupu. Nahrazení zámků nejen způsobuje navýšení rychlosti, ale také zajišťuje garanci postupu výpočtu celého programu a nemůže se stát, že by jedno vlákno zablokovalo celé zpracování

(deadlock). Většina jiných lock-less algoritmů je založena na atomických instrukcí `compare_and_swap` nebo `fetch_and_add` a to platí i pro tuto variantu cyklické fronty. Atomické instrukce jsou ve své podstatě miniaturní zámky v paměti nebo sběrnici, aby jiné vlákno nemohlo modifikovat obsah, dokud se instrukce nedokončí. Obě atomické operace mají v názvu ještě předponu `__sync_`, která označuje vložení paměťové bariéry, aby žádné jiné paměťové operace nemohly být posunuty před nebo za tuto bariéru.

Pokud tato varianta fronty pracuje v konfiguraci s více čtenáři, je nutné používat atomické instrukce jak pro obdržení volného bufferu, tak i pro jeho zpětné zařazení. Nejprve je nutné rezervovat místo ve frontě pomocí funkce `getBuffer`. Písař si potřebuje vytvořit aktuální kopii hodnoty ukazatele *allocated*, zkontrolovat volné místo vůči ukazateli *deallocated* a případně vyčkávat, dokud se místo neuvolní. Pokud je ve frontě volné místo, pokusí se písař posunout ukazatel *allocated* pomocí instrukce `compare_and_swap`, do které vloží adresu sdíleného ukazatele pro porovnání s hodnotou lokální kopie a pokud se stále rovnají, tak dojde k jeho posunutí na další volné místo ve frontě.

Porovnání aktuální hodnoty a kopie může selhat, což znamená, že mezitím jiný písař již ukazatel upravil. Je tedy nutné celý proces od vytvoření kopie aktuální hodnoty znovu opakovat, dokud instrukce `compare_and_swap` neuspěje. Existuje také druhá varianta `getBufferNW`, kde se nevyčkává na uvolnění místa ve frontě a případný pokus o posunutí ukazatele se provádí pouze jednou. Pokud dojde někde k chybě, funkce skončí a vrátí ukazatel na `NULL`. I když atomické instrukce garantují postup výpočtu celého programu, může se stát, že některému z vláken se nepodaří několikrát za sebou rezervovat místo a může docházet k jeho hladovění.

Po naplnění bufferu písař zavolá funkci `putBuffer`, která si vystačí jen s jednodušší atomickou instrukcí `fetch_and_add`. Tato instrukce pouze atomicky přečte aktuální hodnotu ukazatele *queued* a posune ho o místo dále. Díky tomu již písař ví, na které místo ve frontě má vložit adresu naplněného bufferu. Po konci zápisu adresy bufferu písař velice krátkou dobu aktivně vyčká, než předcházející písaři doposouvají ukazatel *head* až na jeho úroveň. Následně může na závěr posunout i on ukazatel *head*

```
// function for getting free buffer with active waiting
char* Wn_getBuffer(CQhandle * handle) {
  CQ * cq;
  CQentry * entry;
  char * buffer;
  long old, new, index, success;

  cq = handle->cq;
  do {
    while ((old = cq->allocated) - cq->deallocated ==
    ↪  cq->queuesize) {} // test free place
    new = old + 1;
    success = __sync_bool_compare_and_swap(&cq->allocated,
    ↪  old, new); // compare old values and store new value
  } while (!success);
  index = old & cq->mask;
  entry = cq->queue + index;
  buffer = entry->buffer;
  return buffer;
}
// function for inserting full buffer
void Wn_putMsg(CQhandle * handle, char * buffer) {
    CQ * cq;
    CQentry * entry;
    long old, index;

    cq = handle->cq;
    // shift pointer queued to the next element
    old = __sync_fetch_and_add(&cq->queued, 1);
    index = old & cq->mask;
    entry = cq->queue + index;
    entry->buffer = buffer;
    entry->counter = cq->readers;
    // wait until all threads finish their writes
    while (cq->head < old) {}
    cq->head = old + 1;  // only this thread can update head
}
```

Ukázka kódu 2.3: Procedura pro získání bufferu a jeho vložení do fronty při více písařích. Tyto procedury jsou skryté za voláním *getBuffer* a *putMsg*.

bez potřeby atomických instrukcí, a tím je nová zpráva úspěšně vložena a zařazena do fronty.

Funkce pro vkládání nových zpráv jsou v ukázce zdrojového kódu 2.3. Obdobný mechanizmus funguje i pro čtenáře, kde jsou taktéž obě fáze prováděny pomocí stejných atomických instrukcí.

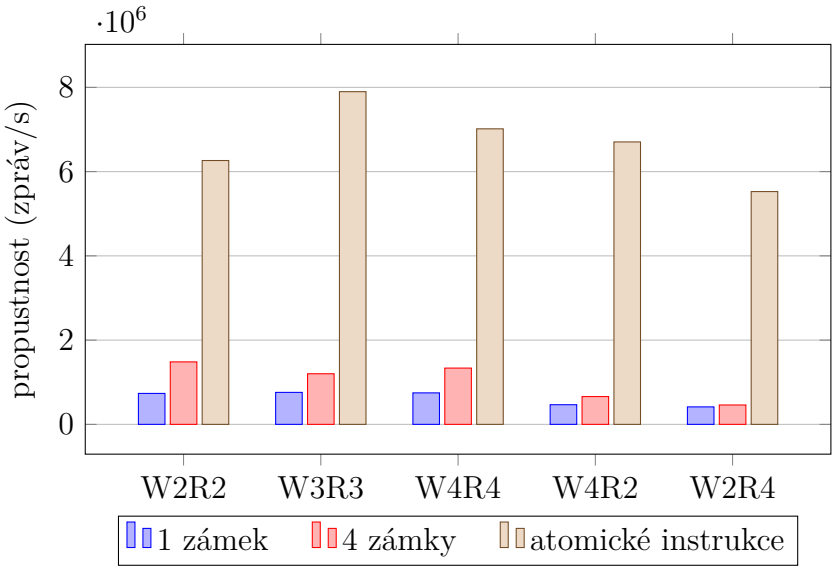
2.3 Srovnání variant *CQ*

Všechny implementační detaily a různé varianty *CQ* již byly představeny dopodrobna a ještě zbývá změřit, zda lock-less varianta skutečně výkonnostně převyšuje implementaci s klasickými zámky. Pro měření je připravena testovací aplikace, která vytvoří definované počty písařů a čtenářů a simuluje jejich práci, kdy písař sekvenčně zapíše zprávu, vloží ji do fronty, čtenář vyzvedne a opět přečte celou zprávu. Podrobněji je aplikace popsána až v poslední kapitole o měření výsledků optimalizací.

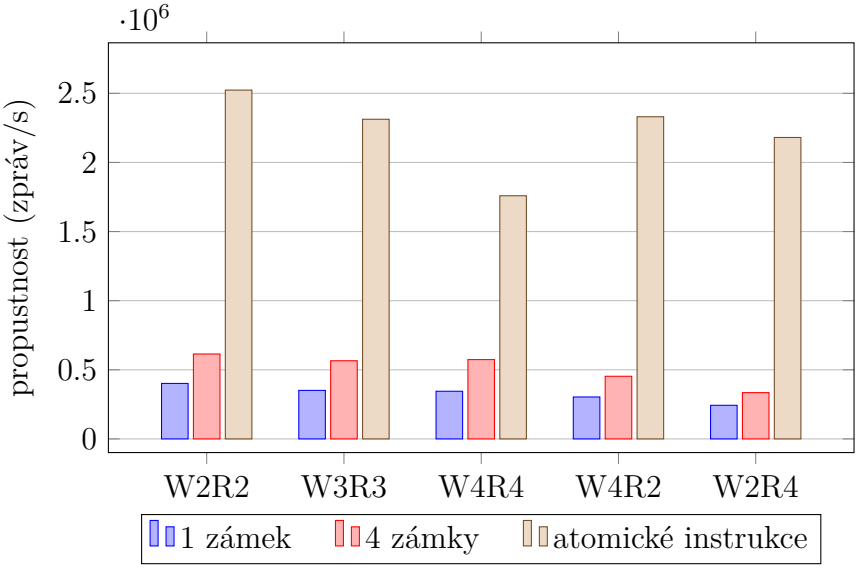
Nejprve probíhá měření v konfiguraci se stejnými počty písařů a čtenářů a následně i pro srovnání situace, kdy písaři nebo čtenáři výrazně nestíhají. Pro měření je v testovací aplikaci nastavena velikost fronty na 4096 zpráv, celkem se zapisuje 12 miliónů zpráv o velikosti 256 bytů a čtenáři pracují v módu *single*. Zároveň všichni písaři a čtenáři běží na stejném procesoru. Dle naměřených výsledků z grafu 2.4 lock-less varianta s atomickými instrukcemi skutečně výrazně převyšuje obě varianty založené na jednom a čtyřech zámcích.

Samozřejmě naměřené hodnoty se vždy velmi odvíjejí od způsobů rozmístění vláken na procesory při výpočtu. Vytvořená testovací aplikace umožňuje přesně specifikovat, které jádro kterého procesoru má písař nebo čtenář obsadit a zůstat tam po celou dobu běhu programu. V grafu 2.5 je taktéž patrné, že při odstěhování všech čtenářů na druhý procesor dochází k výraznému snížení celkové propustnosti fronty u všech variant *CQ*, ale fronta s atomickými instrukcemi stále výrazně dominuje oproti ostatním. V měření se použilo stejné nastavení parametrů jako u předchozího měření, jen se tedy všichni čtenáři odstěhovali na druhý procesor.

Obrázek 2.4: Propustnost různých variant *CQ* při několika konfiguracích počtů písařů (W) a čtenářů (R). Hodnoty jsou naměřeny v milionech zpráv za sekundu, Všechna vlákna využívají jeden procesor.



Obrázek 2.5: Propustnost různých variant *CQ* při několika konfiguracích počtů písařů (W) a čtenářů (R). Hodnoty jsou naměřeny v milionech zpráv za sekundu a všichni čtenáři jsou na jednom procesoru a všichni písaři na druhém procesoru.



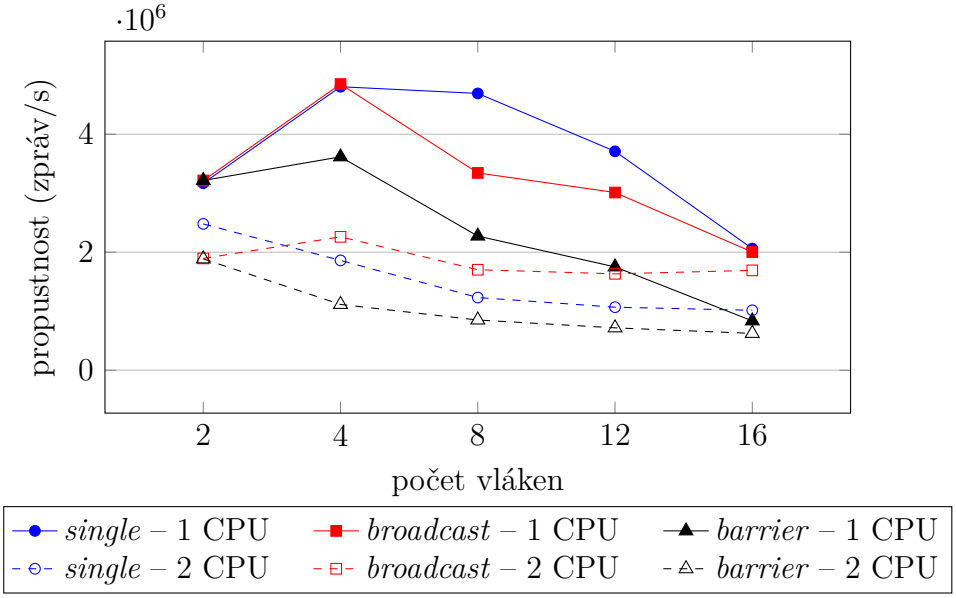
Z měření propustností lock-based a lock-free cyklických front *CQ* vyšla jako jasný vítěz varianta bez zámků. V následujících kapitolách se proto práce zabývá pouze *CQ* implementovanou pomocí atomických operací, pro kterou probíhá návrh dalších optimalizací, jejich implementace a následné měření výkonnostních přínosů. Pro ilustraci chování atomické varianty fronty při různých parametrech a způsobech rozmístění písařů a čtenářů jsou vytvořeny grafy s hodnotami propustností v miliónech zpráv za sekundu, od kterých se odráží testování zrychlení optimalizací v poslední kapitole této práce.

Jak je patrné z grafu propustností 2.6, tak při navyšování počtu vláken operujících s *CQ* dochází ke značnému zpomalování, které již plyne z podoby cyklické fronty, kde se vlákna perou o modifikaci sdílených ukazatelů. Primárním účelem *CQ* není podpora škálování přístupů z desítek vláken současně, ale aby fronta sloužila jako základní kámen pro řetězové zpracování (pipeline model) a pro předávání mezivýsledků. Nejvhodnější model pro běh této cyklické fronty je model výpočtu, kdy v jeden okamžik s jednou instancí *CQ* v řetězci pracuje pouze několik vláken, ale nad celým řetězcem již dohromady operují i desítky vláken.

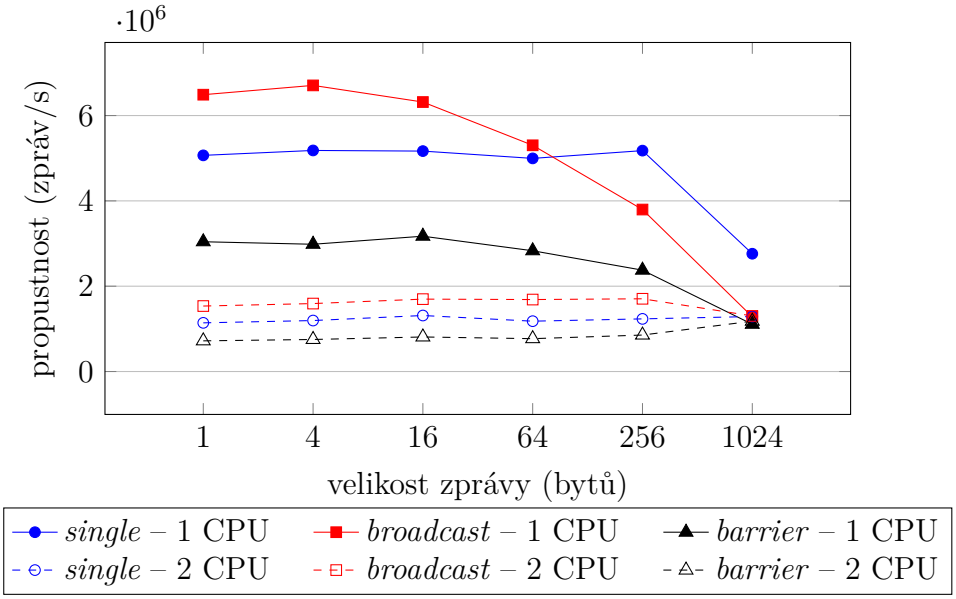
Z dalšího ukázkového grafu propustností 2.7 plyne, že při narůstání délky zprávy se také prodlužuje potřebný výpočetní čas vlákna ke zpracování delší zprávy, a to samozřejmě vede k celkovému snižování propustnosti *CQ*. Stále je také patrný rozdíl, pokud se využívá rozmístění vláken s jedním procesorem nebo se dvěma procesory, a vlákna si tedy předávají všechna data až přes hlavní paměť a nemají žádnou společnou L3 cache paměť.

Posledním grafem ilustrujícím chování atomické varianty je obrázek 2.8, který sleduje propustnost v závislosti na zvyšujícím se počtu míst uvnitř *CQ*. Zajímavým zjištěním ale je, že i při navyšujícím se počtu míst uvnitř fronty nedochází ani k sebemenšímu náznaku snižování propustnosti cyklické fronty, které by se dalo očekávat kvůli nutnosti nahrávat do cache paměti více a více rozdílných částí fronty a s tím související i menší pravděpodobnost, že data zůstanou v cache paměti déle.

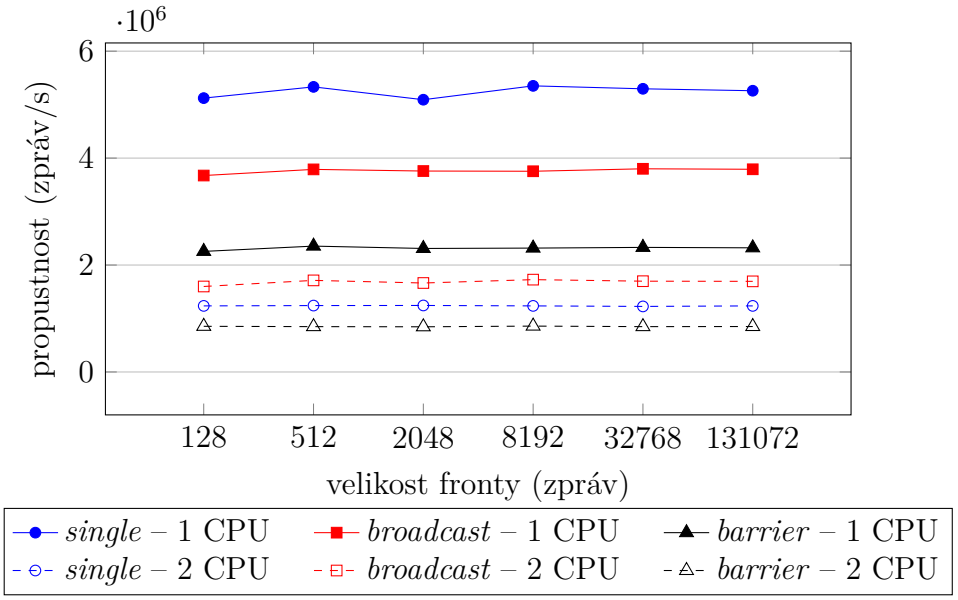
Obrázek 2.6: Propustnost atomické varianty *CQ* v závislosti na rostoucím počtu vláken a způsobu rozmístění písařů a čtenářů. Hodnoty jsou naměřeny v milionech zpráv za sekundu, velikost fronty je nastavena na 8192 míst, celkem se zapisuje 12 miliónů zpráv o velikosti 256 bytů.



Obrázek 2.7: Propustnost atomické varianty *CQ* v závislosti na rostoucí velikosti zprávy a způsobu rozmístění písařů a čtenářů. Hodnoty jsou naměřeny v milionech zpráv za sekundu při konfiguraci W4R4, velikost fronty je nastavena na 8192 míst a celkem se zapisuje 12 miliónů zpráv.



Obrázek 2.8: Propustnost atomické varianty *CQ* v závislosti na rostoucí velikosti fronty a způsobu rozmístění písařů a čtenářů. Hodnoty jsou naměřeny v milionech zpráv za sekundu při konfiguraci W4R4 a celkem se zapisuje 12 miliónů zpráv o velikosti 256 bytů.



2.4 Dostupná řešení podobná *CQ*

Různých implementací cyklických front podporujících paralelní přístup již bylo na světě vytvořeno nepřeberné množství. Většina implementací se specializuje pouze na jeden určený problém nebo výpočetní model a je na něj přímo optimalizovaná. Jiné se zase snaží pokrýt co nejvíce různorodých případů užití a být co nejvíce univerzální, ale v některých situacích za cenu nižšího výkonu. Naše implementace cyklické fronty *CQ* se zaměřuje především na předávání mezivýsledků uvnitř výpočetního řetězce (pipeline model) a umožňuje rozdělit operaci vkládání zpráv do fronty na dopřednou rezervaci místa s obdržením bufferu a na následné zařazení naplněného bufferu se zprávou. Taktéž operace vybírání zpráv je rozdělena na obdržení plného bufferu se zprávou a na následné vrácení zpracovaného bufferu zpět do fronty.

Během hledání na Internetu a analyzování dostupných prací podobných *CQ*, se povedlo nalézt mnoho jiných lock-free cyklických front, které

podporují model pouze pro jednoho písaře a čtenáře. Existuje i několik málo řešení pro podporu vícero písařů a čtenářů. Takovou implementací je například lock-free cyklická fronta v C++ knihovně Boost [11] či cyklická fronta od Chaoran Yang a John Mellor-Crummey [12]. Téměř v žádném z dostupných projektů není implementována možnost si nejdříve rezervovat místo v cyklické frontě s bufferem a až poté do něj začít produkovat data. Díky této vlastnosti je možné zavést optimalizaci *work-stealing*, která je popsána až v následující kapitole.

Povedlo se také nalézt jediný projekt, který se přibližuje myšlenkou na rozdělení operací, ale používá tento mechanizmus pouze u písařů. Taktéž se nepovedlo nalézt lock-free cyklickou frontu s možností si vybrat způsob zpracování zpráv čtenáři z cyklické fronty (*single*, *broadcast* nebo *barrier*).

2.4.1 Cyklická fronta od Faustino Frechilla

Jedinou nalezenou podobnou implementací lock-less cyklické fronty je veřejně dostupný projekt od Faustino Frechilla na serveru CodeProject [13]. Autor používá celkově tři ukazatele místo klasických dvou na začátek a konec fronty. Jedním z rozdělených ukazatelů na začátku fronty je `writeIndex` sloužící pro rezervaci místa ve frontě a téměř kopíruje funkcionalitu ukazatele `allocated`. Podstatným rozdílem oproti `allocated` je, že ukazatel `writeIndex` slouží k rezervaci přesného místa ve frontě, na který právě ukazuje, ale `allocated` slouží pouze k rezervaci nespecifikovaného místa ve frontě, a pořadí se tedy v *CQ* ještě může změnit mezi rezervací místa a vložením bufferu. Druhým ukazatelem je `maximumReadIndex`, který se posouvá až po dokončení zápisu adresy vkládané struktury do fronty a kombinuje dohromady vlastnosti ukazatelů `queued` a `head` v *CQ*.

Dalším důležitým rozdílem je, že tato fronta nevlastní předem vytvořené buffery, ale pouze ukládá ukazatele na strukturu alokovanou a předanou od písaře. Čtenář se následně stará při vyzvedávání zprávy o její správnou dealokaci. Na jednu stranu se jedná o nemalou úsporu paměti oproti *CQ*, kde může být velice podstatná část paměti zabrána a dlouho nevyužita, ale je nutná dodatečná časová režie na alokování a dealokování struktur

v paměti. Vždy ale záleží na konkrétní situaci a různých parametrech (dostupná paměť, velikost zprávy, počet míst, …), který z přístupů lze vyhodnotit jako vhodnější pro použití.

V této implementaci cyklické fronty ale mohou čtenáři zpracovávat zprávy pouze v režimu *single* a není tu možnost předat stejnou zprávu všem čtenářům jako u *CQ*. Tato funkcionalita je již od začátku zamýšlena a bude nutná při práci na budoucích projektech v dalších bězích magisterského předmětu Vícejádrové systémy na FIT ČVUT.

Kapitola 3

Návrh optimalizací

Tato kapitola se věnuje návrhům optimalizací pro paralelní běh již implementované lock-free varianty cyklické fronty *CQ* a analýze jejich očekávaných přínosů. Většina optimalizací se týká efektivnějšího využívání paměti cache v procesorech a snížení četnosti situací, kdy musí docházet k zneplatňování řádků v cache paměti a k opětovnému načítání z hlavní paměti. Již během prvotní implementace *CQ* vznikl návrh na implementaci první a třetí optimalizace v této kapitole.

3.1 Zarovnání sdílených struktur na velikost řádku cache paměti

První z navržených optimalizací je zarovnání všech sdílených datových struktur a velikosti každé zprávy přesně na násobek velikosti jednoho řádku cache paměti, tedy násobek 64 bytů. Díky tomu lze předejít jevu false-sharing (doslova falešnému sdílení) vyskytujícímu se v situacích, kdy vlákna bežící na jiných jádrech procesoru modifikují data, která mají rozdílné adresy v paměti, ale sdílejí stejnou řádku v cache paměti. Správa paměti na vícejádrovém systému zpravidla poskytuje některý z protokolů pro koherenci cache pamětí a ten zajišťuje, že je celá řádka v cache paměti zneplatněna i v ostatních jádrech vykonávajících kód jiných vláken, i když přistupují k jiným částem paměti uvnitř stejné řádky. Tento jev tedy

způsobuje zbytečnou režii a zpomalování výpočtu, které může být snadno odstraněno za cenu o trochu větší paměťové náročnosti.

V *CQ* tento problém jistě velmi často nastává, jelikož ukazatelé do fronty leží uvnitř jedné velké sdílené struktury a přistupuje se k nim z mnoha vláken bežících na různých jádrech. Eliminování false-sharingu je velmi snadné pomocí vložení atributu `aligned(64)` k proměnné nebo rovnou k celé datové struktuře, a tím se zabezpečí, že začátek proměnné či struktury bude zarovnán přesně na násobek 64 bytů, respektive na začátek jedné řádky v cache paměti. Zbylé místo uvnitř stejné řádky je bohužel nevyužité. Druhou částí této optimalizace je zvětšení velikosti každé zprávy ve frontě přesně na nejbližší větší násobek 64 bytů. Díky tomu by se mělo předcházet dalšímu false-sharingu, který v *CQ* vzniká nejvíce při práci s malými zprávami.

```
#define SHARED  volatile
#define ALIGNED __attribute__((aligned(64)))
⋮
  SHARED long ALIGNED allocated;
  SHARED long ALIGNED queued;
  SHARED long ALIGNED head;
  SHARED long ALIGNED tail;
  SHARED long ALIGNED read;
  SHARED long ALIGNED deallocated;
⋮
```

Ukázka kódu 3.1: Ukázka zdrojového kódu pro zarovnání sdílených ukazatelů na velikost jednoho řádku cache paměti.

3.2 Kopírování s obcházením cache paměti

Jak bylo představeno v předešlé kapitole, naše implementace cyklické fronty *CQ* umožňuje vkládat zprávy do fronty dvěma způsoby. Prvním je, že si vlákno může nejprve vyžádat od *CQ* rezervování místa ve frontě s poskytnutím volného bufferu pro zápis. Do něho může libovolně dlouho

zapisovat a na konci pouze předá zaplněný buffer zpět do *CQ*, která si buffer vloží do cyklické fronty.

Druhým způsobem je, že si vlákno vytvoří zprávu ve svém bufferu a ten si následně *CQ* uvnitř při vkládání překopíruje. Tato optimalizace se týká pouze tohoto druhého způsobu vkládání zpráv a její největší přínos lze očekávat u velice dlouhých zpráv v řádech kilobytů až megabytů, které při standardním kopírování způsobí vyklízení mnoha jiných řádků v cache paměti.

V této optimalizaci se standardní kopírování přes `memmove` či `memcpy` nahrazuje vlastní implementací kopírování pomocí non-temporal instrukcí. Ty zajišťují nahrání části hlavní paměti přímo do vektorových registrů a jejich následné uložení do hlavní paměti s obcházením všech úrovní cache paměti. Využívá se built-in funkcí překladače GCC [4], pro načtení 128 bitů je to `__builtin_ia32_movntdqa` a pro jejich uložení `__builtin_ia32_movntdq`. Obě funkce potřebují podporu sady instrukcí minimálně SSE 4.1. Bohužel testovací server již neobsahuje podporu sady instrukcí AVX2 nebo AVX512, které by umožnily přesun 256 nebo dokonce 512 bitů v jedné instrukci s obcházením cache paměti.

```
⋮
const __m128i xmm0 =
↪  __builtin_ia32_movntdqa((__m128i*)(src   ));
__builtin_ia32_movntdq((__m128i*)(dst   ), xmm0);
const __m128i xmm1 =
↪  __builtin_ia32_movntdqa((__m128i*)(src+16));
__builtin_ia32_movntdq((__m128i*)(dst+16), xmm1);
const __m128i xmm2 =
↪  __builtin_ia32_movntdqa((__m128i*)(src+32));
__builtin_ia32_movntdq((__m128i*)(dst+32), xmm2);
const __m128i xmm3 =
↪  __builtin_ia32_movntdqa((__m128i*)(src+48));
__builtin_ia32_movntdq((__m128i*)(dst+48), xmm3);
⋮
```

Ukázka kódu 3.2: Ukázka zdrojového kódu pro kopírování s obcházením cache paměti.

Dále se při tomto upraveném kopírování využívá i předešlé optimalizace, kdy jsou zprávy vždy stejně veliké a zarovnány přesně na velikost řádku cache paměti. Díky tomu se nemusí provádět zdržující počítání offsetu a kontrola zbývajících počtů bitů pro překopírování dat a je možné si vše předpočítat pouze jednou na začátku při inicializaci fronty. Následně dochází ke střídavému využívání všech dostupných osmi 128 bitových vektorových registrů.

3.3 Work-stealing

Jediná ze všech navržených optimalizací, která nemá hlavní úkol především v lepším využívání cache paměti, je work-stealing. Jak již z názvu plyne, jedná se doslova o „kradení práce“ jiným vláknům. Během využívání cyklické fronty může docházet k situacím, kdy je fronta plná nebo naopak úplně prázdná, tedy písaři nestíhají dostatečně zásobovat novými zprávami nebo čtenáři nestíhají jejich zpracování. Právě v těchto případech by se mělo uplatit vylepšení work-stealing.

Naší implementaci cyklické fronty *CQ* lze pospojovat do „řetězu“ a vytvořit model paralelního výpočtu pipeline, protože každá instance *CQ* může obsahovat ukazatele na předešlou (`prevqueue`) a následující (`nextqueue`) instanci *CQ*. Díky tomu je umožněno „krást“ nezpracované položky i z ostatních front a tím lépe využívat výpočetní kapacitu. Pokud z nejbližších instancí *CQ* nelze ukrást nějakou zprávu ke zpracování, může se vlákno podívat i do vzdálenějších instancí a pomoci se zpracováním zprávy až tam. Může prozkoumat fronty jak ve směru k začátku pipeline, tak i ve směru ke konci pipeline. Směr se vždy odvíjí od toho, zda konkrétní cyklická fronta je plná (prohledává se ke konci) nebo prázdná (prohledává se k začátku).

Důležitým kritériem pro zavedení work-stealingu je rozdělení operace vkládání na dopředné rezervování místa s obdržením bufferu a na zařazení naplněného bufferu. Bez této možnosti rezervace místa by nebylo možné zaměstnat čekající písaře zpracováním zpráv z jiných naplněných cyklických front, jelikož mají již připravená data čekající na vložení do fronty. Pokud se

jim nejdříve nepovede rezervovat místo, nezačnou ani produkovat data pro vložení do fronty a mohou se pokusit o krádež z jiných naplněných front.

Při inicializaci *CQ* je nutné předat také ukazatel na funkci, která se má vykonávat, pokud je fronta plně obsazená a písaři nemají kam vkládat další zprávy. Zároveň se při počáteční inicializaci musí rozhodnout, zda může být work-stealing vůbec povolen. Pokud má *CQ* fungovat v režimu *broadcast* nebo *barrier*, tak by krádež z jiných front porušila mód čtení, a proto musí být work-stealing v těchto situacích zakázán.

Samotná krádež funguje pomocí vytvoření dočasného handleru do jiné instance *CQ* a pomocí něho se zpracuje zpráva. Na konci každé krádeže je tento dočasný handler zrušen.

```
// scheduler - finds and processes element from next queue

void CQ_busywriter(CQhandle * handle) {
  char * buffer;
  CQ * cq;
  CQhandle handle2;

  cq = handle->cq;
  while (cq) {
    if ((cq->worker != 0) && (cq->tail != cq->head)) {
      handle2.cq = cq;
      handle2.rdwr = 0;
      handle2.id = -2;
      if ((buffer = cq->getMsgNW(& handle2))) {
        cq->worker(& handle2, b);
        cq->putBuffer(& handle2, b);
        return;
      }
    }
    cq = cq->nextqueue;
  }
  return;
}
```

Ukázka kódu 3.3: Ukázka zdrojového kódu pro vytížení čtenáře během čekání na příchozí zprávu od písaře.

3.4 Odkládání čtení sdílených ukazatelů

Další navrženou optimalizací je odložení čtení sdílených ukazatelů do cyklické fronty, které se používají v daném vlákně pouze pro čtení. Důsledkem této optimalizace je, že si vlákna kontrolují plnost nebo prázdnost fronty pouze přes své vlastní lokální kopie sdílených ukazatelů a pokud při zápisu nebo čtení dorazí na jejich hranice, tak si teprve zkopírují aktuální hodnotu ukazatele z paměti. Tuto optimalizaci lze použít jen pro některé sdílené ukazatele, u čtenáře je to ukazatel *head* a u písaře *deallocated* a *tail*.

Díky této úpravě by se mělo snížit nutné zapisování změněného obsahu řádku cache do hlavní paměti a znovu nahrávání do jiné cache paměti, aby si jiné vlákno mohlo přečíst aktuální hodnotu. Většinou ji při kontrole plnosti fronty ještě nepotřebuje a vystačí si se starou hodnotou ve vlastní cache paměti.

3.5 Mazání nepotřebných řádků cache paměti

Předpokládaným přínosem této optimalizace by mělo být zrychlení načítání řádky do cache paměti díky pravidelnému uvolňování již nepotřebných částí cache paměti. Účinek by se měl projevit jak u písařů, tak i čtenářů, protože každé vlákno zprávu zapisuje nebo čte pouze jednou. Jelikož řádka cache paměti je vždy 64 bytů dlouhá, vyplachování se provádí vždy po zapsání nebo přečtení 64 znaků ze zprávy.

Pro vyplachování řádku paměti se používá built-in funkce překladače GCC [4] `__builtin_ia32_clflush`, která ze všech úrovní cache paměti odstraní řádek obsahující data z adresy předané v parametru. Tato funkce vyžaduje podporu instrukční sady alespoň SSE2.

```
// gets buffer with message for mode xxRnn
char * Rnn_getMsg(CQhandle * handle) {
  CQ * cq;
  CQentry * entry;
  char * buffer;
  long index;

  cq = handle->cq;

  // compare local counters
  if (handle->tail >= handle->head) {
    while (handle->tail == cq->head) {
      // long waiting for new message from writer
      if (cq->eof) return 0;
      // during waiting, reader can help to writer
      if ((cq->prevqueue)
          && (cq->optimizations & OPT_WSTBW)) {
        cq->idlereader(h);
      }
    }
    handle->head = cq->head;  // late update
  }

  index = handle->tail & cq->mask;  // modulo via logical AND
  entry = cq->queue + index;
  buffer = entry->buffer;

  return buffer;
}
```

Ukázka kódu 3.4: Ukázka zdrojového kódu pro odkládání čtení sdíleného ukazatele u čtenáře.

3.6 Přednačítání ukazatelů

V této optimalizaci *CQ* se má zrychlit přístup na datovou strukturu, která obsahuje zprávu k přečtení. Pokud je předem jasné, že vlákno bude zapisovat nebo číst v pořadí následující položku ve frontě, lze použít další built-in funkci překladače GCC [4] `__builtin_prefetch`, které se v parametru předá adresa požadované paměti k přednačtení do cache paměti. Dalšími parametry lze také specifikovat, zda se přednačtená data mají použít pro zápis nebo pouze pro čtení a také lze definovat lokalitu dat, tedy jak daleko od jádra se mají data nahrát. Tento parametr určuje, do jaké úrovně cache paměti se data mají nahrát.

Jak bylo zmíněno v předešlém odstavci, je nutné dopředu vědět, kterou položku lze přednačíst. Proto této optimalizace lze využít pouze v některých situacích, kdy zapisuje jeden písař nebo čte pouze jeden čtenář. Pokud je čtenářů více, musejí fungovat v módu *broadcast* respektive *barrier*, protože každý čtenář musí přečíst postupně stejné zprávy, a je tedy možné předpovědět, jaká data bude v nejbližším čase určitě potřebovat.

3.7 Hromadné vkládání a vybírání zpráv z fronty

Posledním z návrhů optimalizací je hromadné posouvání sdílených ukazatelů na nejnovější a nejstarší prvek fronty. Zamýšleným účelem je více omezit zápis do sdílených datových struktur, které chce jedno z vláken modifikovat a tím zneplatňovat řádky cache paměti jiným vláknům. Písař si udržuje svůj lokální ukazatel na nejnovější záznam ve frontě a po zapsání definovaného balíku (`BATCH_SIZE`) nových zpráv konečně posune i sdílený ukazatel, aby čtenáři mohli číst nově zapsané zprávy.

Úplně totožný mechanizmus funguje i pro čtenáře, kde se ale posouvá ukazatel na nejstarší nepřečtenou zprávu. Mezitím se hromadí ve frontě již přečtené buffery, které písaři ještě nemohou použít. Nakonec také dochází k posunutí sdíleného ukazatele na nejstarší zprávu po přečtení definovaného balíku zpráv (`BATCH_SIZE`).

```
#define BATCH_SIZE 20

// put buffer with message in the queue
void W1_putMsg(CQhandle * handle, char * buffer) {
  CQ * cq;

  cq = handle->q;

  if (handle->write_batch == BATCH_SIZE) {
    cq->head += BATCH_SIZE;
    handle->write_batch = 0;
  }

  handle->head++;
  handle->write_batch++;
  return;
}
```

Ukázka kódu 3.5: Ukázka zdrojového kódu pro hromadný zápis zpráv.

Jelikož zde dochází k inkrementování sdílených ukazatelů až po určitém balíku zpráv, tak může při snížení četnosti příchozích zpráv docházet k problému, že písař příchozí zprávy úspěšně zpracoval, ale čtenář je stále nemůže vyzvednout z fronty. Důležitou podmínkou úspěchu této optimalizace je tedy velmi častý přísun nových zpráv, který omezuje nemožnost přečíst zprávy čtenářem na minimální dobu. Zároveň je nutné zajistit, aby písař po zpracování poslední zprávy také inkrementoval sdílený ukazatel o zbývající počet nevložených zpráv přes nově vytvořenou proceduru `CQ_flushPointers`.

Tuto optimalizaci lze ovšem použít jen v případě, kdy fronta pracuje v režimu maximálně jednoho čtenáře nebo jednoho písaře, tedy *W1R1*, *W1Rn* nebo *WnR1*. Další podmínkou je, že nesmí být zapnuta optimalizace work-stealing, protože nově příchozí vlákno by zapisovalo v cyklické frontě na místa, která již zprávu obsahují, nebo by četlo znovu zprávy, které již byly zpracovány jiným čtenářem. Obě situace se ověřují na začátku při inicializaci cyklické fronty v `CQ_init` a případně je hromadné vkládání a vybírání zpráv zakázáno.

Kapitola 4

Vyhodnocení přínosů optimalizací

Tato kapitola se zabývá experimentálním vyhodnocením všech navržených optimalizací z předcházející kapitoly. Popisuje konfiguraci testovacího serveru a implementaci testovací aplikace využívající lock-less variantu *CQ*. Z naměřených časů výpočtů a propustností jsou následně sestaveny grafy zrychlení při zapnutých optimalizacích oproti vypnutým.

4.1 Testovací architektura

Jak bylo popsáno v předešlých kapitolách, *CQ* je naimplementována pouze pro architekturu x86-64, proto také měření probíhalo na školním testovacím serveru používající stejnou architekturu.

Konfigurace testovacího serveru:

- model serveru: HP ProLiant DL380p Gen8
- procesor: 2x Intel Xeon E5-2690 (8 jader + Hyper-Threading)
- cache paměť: L1 – 512kB, L2 – 2MB, L3 – 20MB
- operační paměť: 32GiB
- operační systém: Ubuntu 12.10 Quantal Quetzal

Podrobnější informace o konfiguraci testovacího serveru jsou dostupné z výstupu programu `dmidecode` [1] v souboru `machine_information.txt` jako příloha k této práci na CD.

4.2 Testovací aplikace

Pro důkladné porovnání různých variant *CQ* a přínosů optimalizací je připravena parametrizovaná testovací aplikace používající cyklickou frontu *CQ* pro ukládání a vyzvedávání zpráv.

Pomocí parametrů lze měnit počty písařů a čtenářů a jejich zamknutí během výpočtu na určené jedno jádro procesoru. Zamknutí vlákna na jádro probíhá pomocí uvedení seznamu CPU ID oddělených čárkami a jako druhý parametr slouží počet písařů, kteří jsou uvedeni na úplném konci seznamu. Přehled všech CPU ID je například dostupný v operačním systému Ubuntu v souboru `/proc/cpuinfo`. Každé měření pro srovnání běhu s optimalizací a bez optimalizace vždy probíhá na totožné topologii a při stejném rozmístění čtenářů a písařů.

Dalšími parametry je možné specifikovat mód, ve kterém mají zpracovávat zprávy čtenáři, velikost zprávy a počet míst uvnitř fronty. Posledním parametrem je výčet optimalizací oddělených čárkou, které mají být použity při běhu testovací aplikace. Bez uvedení optimalizací jsou v základu všechny vypnuty. V testovací aplikaci je také připravený jednoduchý návod na použití (usage), který se vypíše na konzoli při spuštění bez žádného zadaného parametru.

Pro měření optimalizace s non-temporal instrukcemi je testovací aplikace upravena tak, aby písaři a čtenáři používali klasické funkce `writeMsg` a `readMsg`, které musejí obsahy bufferů překopírovat a nepoužívají dopředné obdržení bufferu s rezervací místa ve frontě.

Simulovaná práce pro písaře tvoří zápis všech znaků ASCI abecedy vzestupně stále dokola, dokud se buffer nenaplní a také pro kontrolu zapíše výsledek operace *XOR* všech zapsaných znaků. Počáteční znak se odvíjí od

[1] `http://www.nongnu.org/dmidecode/`

pořadového čísla zprávy. Čtenář po obdržení zprávy také provádí operaci *XOR* přes všechny znaky v bufferu a na závěr ověřuje, zda jsou výsledky *XOR* operací totožné.

Kompilace testovací aplikace probíhala pomocí překladače GCC verze 4.7.4 s parametrem `-msse4.1` pro povolení non-temporal instrukcí sady SSE4.1 a s parametrem `-O2` pro zapnutí druhé úrovně optimalizací zdrojového kódu v překladači GCC. Pro jednoduché spuštění kompilace aplikace je také vytvořený soubor `Makefile`.

4.3 Výsledky měření

Kvůli eliminaci výkyvů ve výsledcích je každá implementovaná optimalizace změřena pětkrát za sebou a v grafech zrychlení se vždy používá minimum ze všech opakování měření. Zrychlení optimalizace se vždy vztahuje ke spuštění testovací aplikace se shodnými parametry a topologií, ale pouze s nepoužitou optimalizací. V popiscích grafů se objevuje rozdělení na „1 CPU“ a „2 CPU“, což definuje topologii, na které běží aplikace. Při označení „2 CPU“ jsou tedy všichni písaři odděleni od všech čtenářů na druhém procesoru, a tedy nesdílejí spolu L3 cache paměť.

V předešlých kapitolách je představeno, že *CQ* umožňuje čtení ve vícero módech. Během testování se používají pouze módy *single* nebo *broadcast*, který je velice podobný módu *barrier*, až na přidanou synchronizaci čtenářů na každé zprávě, a v testech ho proto může pro přehlednost zastoupit.

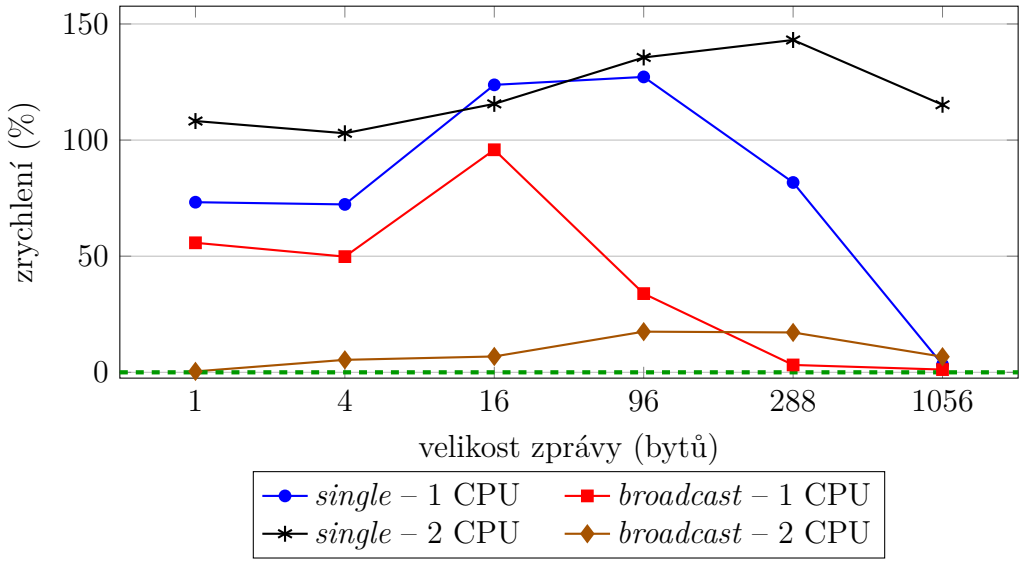
Kvůli přehlednosti všech grafů se v nich vyskytují pouze vypočítaná zrychlení po zapnutí optimalizace, ale ke všem naměřeným absolutním, minimálním a průměrným hodnotám je možné se dostat v textovém souboru `testing_output.txt`, který je výstupem testovacího skriptu `testing.sh`. Oba soubory jsou vloženy jako příloha k této práci na CD.

4.3.1 Zarovnání sdílených struktur na začátek řádku cache paměti

První z implementovaných optimalizací je zarovnání začátku sdílených struktur a velikosti zprávy na začátek řádku cache paměti. Očekávaným výsledkem je eliminace situace, kdy různé zprávy a ukazatele sdílejí stejnou řádku cache paměti a každá úprava v jedné části zneplatní celou řádku u jiných vláken pracujících s jinými zprávami a ukazateli.

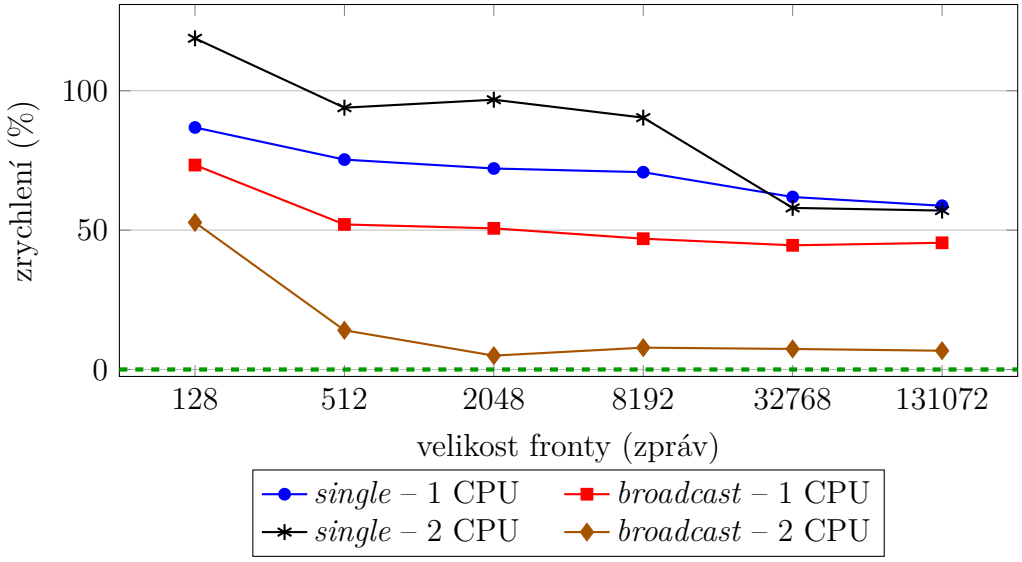
Při testování jsou použity velikosti zpráv, které zabírají necelé řádky cache paměti, tedy nejsou násobkem 64 bytů, aby se u nich projevilo sdílení řádků. Jak je z grafu 4.1 patrné, tak tato optimalizace má právě největší přínos při sdílení stejné řádky několika ukazateli a zprávami. Pokud se velikost zprávy zvětšuje, zmenšuje se podíl řádků v cache paměti sdílejících různé zprávy, a proto také ubývá jevu false-sharing, který má tato optimalizace za úkol eliminovat. Na druhou stranu ale počet řádků sdílejících různé ukazatele zůstává s narůstající délkou zprávy pořád stejný, tedy tížené zrychlení se projevuje nejvíce u nich.

Obrázek 4.1: Graf zrychlení optimalizace *zarovnání sdílených struktur* závislý na velikosti zprávy při různých módech čtení. Další parametry testu: konfigurace W4R4, velikost fronty 2048.



Pokud narůstá počet míst uvnitř fronty, tak samozřejmě přibývá i řádků cache paměti, kde se vyskytuje false-sharing. Z grafu 4.2 je ale zřejmé, že naopak ubývá situací, kde se během využívání cyklické fronty tento jev nakonec projeví a způsobí zdržení ostatních vláken. Při hodně vysokém počtu míst uvnitř fronty dokonce dochází k velmi nepatrnému zpomalení při módu čtení *single*. Jev false-sharing se v tomto případě nejspíše projevuje jen ve velmi málo případech a naopak nahrávání vždy nové řádky pro každou zprávu způsobuje zpomalení, protože při vypnuté optimalizaci se do jedné řádky vměstná až několik zpráv najednou a není nutné vyklízet z cache paměti tolik řádků navíc.

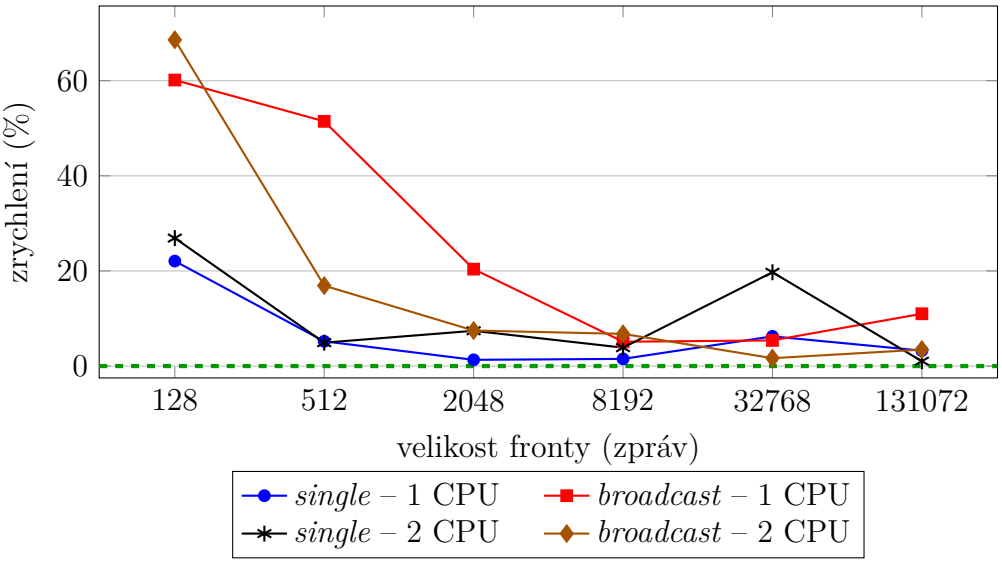
Obrázek 4.2: Graf zrychlení optimalizace *zarovnání sdílených struktur* závislý na velikosti fronty při různých módech čtení. Další parametry testu: konfigurace W4R4, velikost zprávy 4 byty.



Měření zrychlení v grafu 4.3 je provedeno pro srovnání vlivu zarovnání všech sdílených struktur včetně velikosti zpráv vůči zarovnání pouze sdílených ukazatelů. Tento graf prokazuje tvrzení z předešlého odstavce, že false-sharing se při zvětšující se frontě projevuje stále méně a také ze zrychlení ukrajuje nutné nahrávání celé řádky pro každou zprávu zvlášť.

Optimalizace na zarovnání začátku sdílených struktur dopadla v měřeních velice dobře a buď velmi výrazně pomohla běhu cyklické fronty, nebo

Obrázek 4.3: Graf zrychlení optimalizace *zarovnání velikosti zprávy* závislý na velikosti fronty při různých módech čtení vůči již zarovnaným ukazatelům. Další parametry testu: konfigurace W4R4, velikost zprávy 4 bytů.



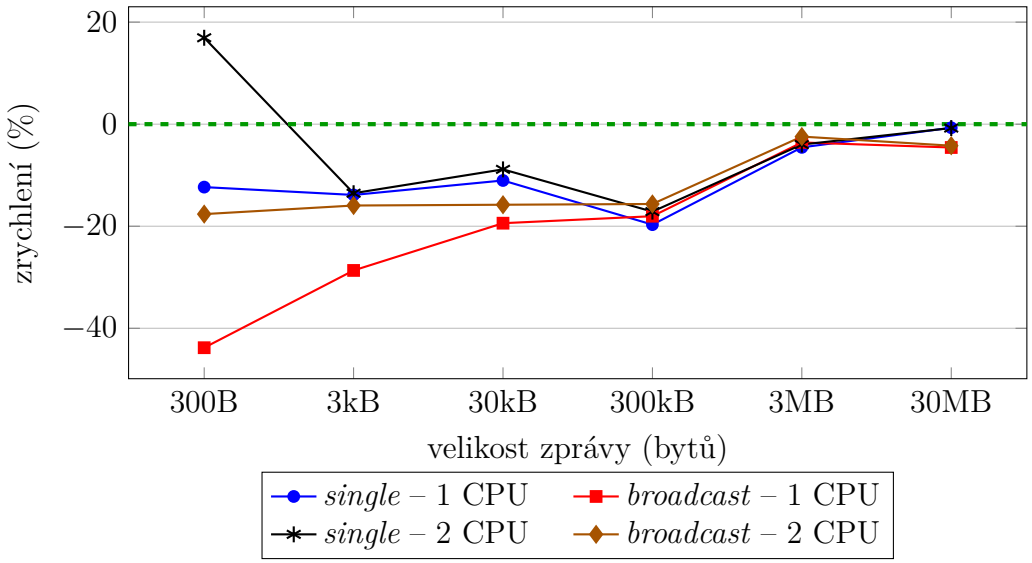
nezpůsobila žádný významný propad výkonnosti. Proto lze tuto optimalizaci skutečně považovat za přínos pro fungování *CQ*, především při módu čtení *broadcast*, kdy se zneplatnění jedné řádky s upravenou zprávou dotýká mnohem více vláken současně nežli v režimu *single*.

4.3.2 Kopírování bez ovlivnění cache paměti

Během klasického kopírování pomocí `memmove` se postupně načítají data do cache paměti a opět ukládají do jiných řádků. Tím dochází k soustavnému zaplavování daty malých cache pamětí, kterému umějí zabránit non-temporal instrukce na kopírování dat obcházející všechny úrovně cache paměti. Zároveň pro jednoduchost a rychlost kopírování musí být v aplikaci zapnuta předchozí optimalizace na zarovnání zprávy na velikost řádky cache paměti. Pro testování této optimalizace je také testovací aplikace upravena tak, aby nepoužívala rezervování místa ve frontě, ale použila klasický přístup s překopírováním bufferu.

Z naměřených výsledků v grafu 4.4 se ale ukazuje, že non-temporal instrukce nepřinášejí očekávané zrychlení, ba naopak dochází k nemalému

Obrázek 4.4: Graf zrychlení optimalizace *kopírování bez ovlivnění cache paměti* závislý na velikosti zprávy. Další parametry testu: konfigurace W4R4, velikost fronty 1024.



zpomalení ve všech testovaných konfiguracích, které se zmenšuje při zvětšení velikosti zprávy. Jedním z možných vysvětlení, proč ke zrychlení nedochází, je problém, že non-temporal instrukce okamžitě vytěžují sdílenou paměťovou sběrnici s operacemi zápisu do hlavní paměti a blokují ostatní operace jiných vláken. Na druhou stranu se propagace změn do hlavní paměti při zápisu přes cache paměť odloží až na pozdější dobu, kdy to bude skutečně nutné provést.

Zajímavé by bylo také srovnání, pokud by se místo klasického kopírování přes cache paměť použila non-temporal instrukce pro přesun 256 nebo 512 bitů v jedné instrukci. Pro takové porovnání ale v současné době nedisponujeme žádnou serverovou architekturou x86-64, která by podporovala sadu instrukcí AVX2 nebo AVX512.

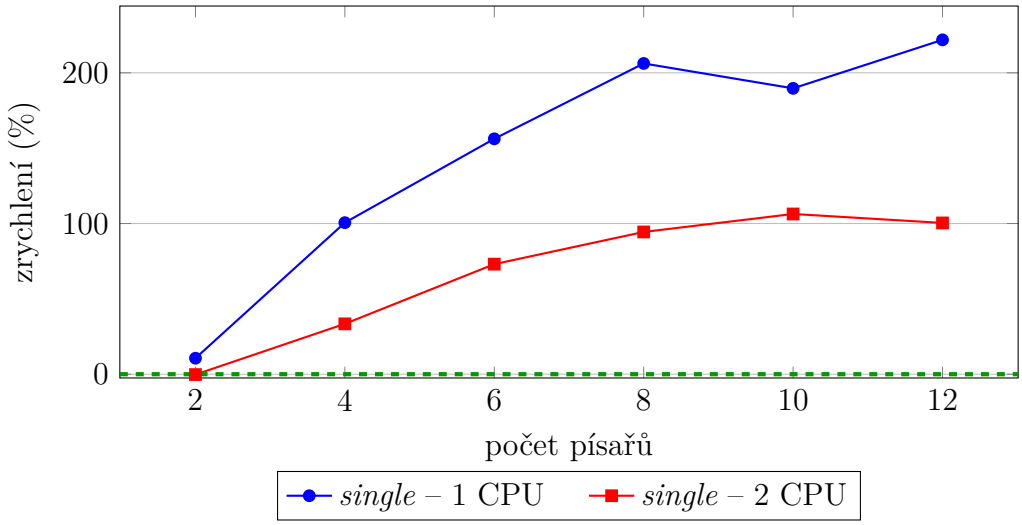
4.3.3 Work stealing

Další testovanou optimalizací je work-stealing, tedy doslova kradení práce jiným vláknům, pokud není zrovna žádná zpráva ke zpracování ve frontě.

Je jedinou z optimalizací, která nemá hlavní úkol v lepším využívání cache paměti, ale v lepším zaměstnání všech vláken aplikace.

Pro porovnání výsledků testy probíhaly pouze s kradením práce písařů od čtenářů, protože v tomto případě si lze vystačit pouze s jedinou instancí *CQ* a srovnání je názornější. Z grafu 4.5 je patrné, že pokud čtenáři nestíhají vyprazdňovat frontu, tak pomoc od nevytížených písařů skutečně velmi pomůže navýšit celkovou propustnost cyklické fronty.

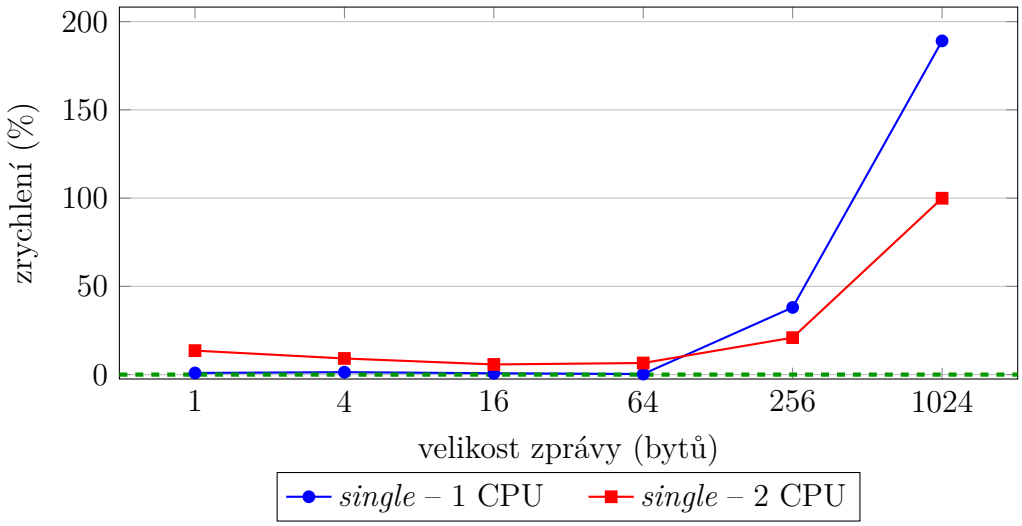
Obrázek 4.5: Graf zrychlení optimalizace *work-stealing* závislý na počtu písařů. Další parametry testu: 2 čtenáři, velikost zprávy 1024B, velikost fronty 2048.



Dále z grafu 4.6 lze vyčíst nárůst zrychlení při navyšující se velikosti zprávy, protože čas strávený nad zpracováním jedné zprávy se zvětšuje, a tedy i výpomoc od „nudících se“ písařů znamená vetší celkovou časovou úsporu běhu celé aplikace. Zrychlení má stejnou tendenci růstu i pro větší velikosti, než jsou uvedeny pouze v grafu.

Jak bylo vysvětleno v kapitole o návrzích optimalizací, tak jedinou slabinou work-stealingu je, že může fungovat pouze při módu *single*, jelikož při ostatních módech si čtenáři drží ukazatele na svůj konec fronty pouze u sebe lokálně. I přes tuto slabinu, při které ovšem nedochází ke zpomalení *CQ*, lze optimalizaci work-stealing považovat za velmi přínosnou.

Obrázek 4.6: Graf zrychlení optimalizace *work-stealing* závislý na velikosti zprávy. Další parametry testu: konfigurace W12R2, velikost fronty 2048.



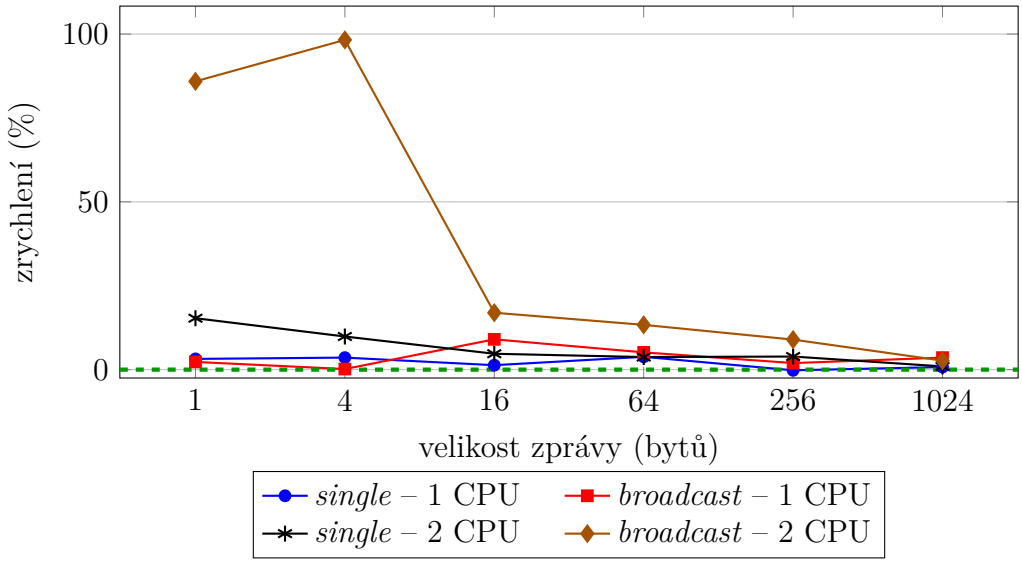
Další netestovaný potenciál se ale skrývá při zapojení několika instancí *CQ* do řetězu za sebe. Poté dochází i ke kradení práce z předchozích front, pokud čtenáři nemají nic na práci, nebo z následujících front v pipeline, pokud písaři nemají místo, kam zapisovat nové zprávy. Tento příklad použití bude využíván a podrobně testován v následujících běžích předmětu Vícejádrové systémy (MI-MCS) na FIT ČVUT, kdy se *CQ* bude používat v různých semestrálních pracích.

4.3.4 Odkládání čtení sdílených ukazatelů

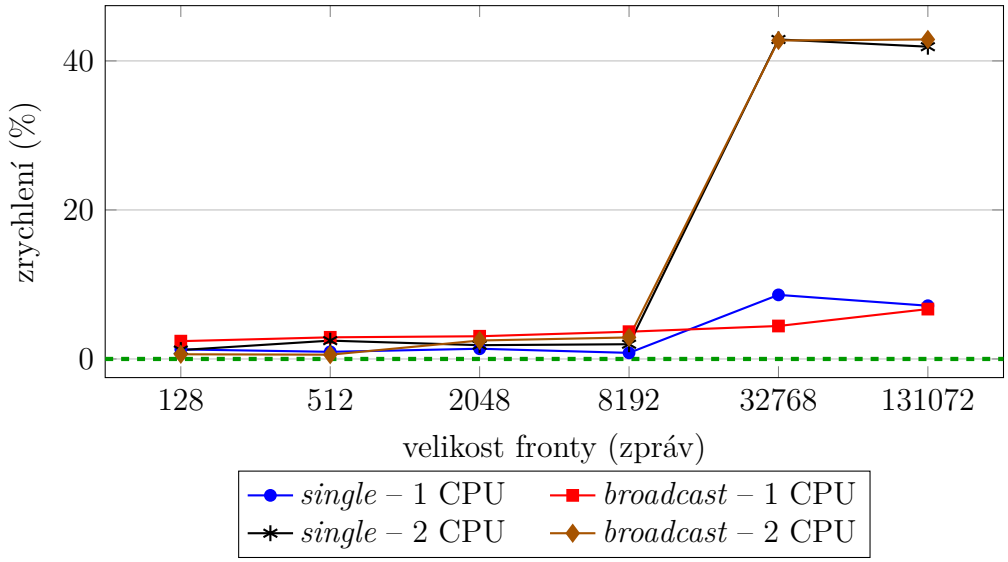
Další testovanou optimalizací je odkládání čtení sdílených ukazatelů, což znamená uchovat si lokální kopie sdílených ukazatelů, dokud to bude jen možné a zbytečně nevyžadovat vždy nejaktuálnější hodnotu až z hlavní paměti.

Při návrhu této optimalizace již bylo zřejmé, že jde o velice nadějné vylepšení a měření to v grafu 4.7 skutečně prokázalo především u velmi malých velikostí zpráv, kde se díky zpožděnému obnovení ukazatelů dosáhlo až dvojnásobného zrychlení. Bohužel s rostoucí velikostí zprávy se zrychlení velmi strmě snížilo na úplné minimum.

Obrázek 4.7: Graf zrychlení optimalizace *odkládání čtení sdílených ukazatelů* závislý na velikosti zprávy při různých módech čtení. Další parametry testu: konfigurace W4R4, velikost fronty 8192.



Obrázek 4.8: Graf zrychlení optimalizace *odkládání čtení sdílených ukazatelů* závislý na velikosti fronty při různých módech čtení. Další parametry testu: konfigurace W4R4, velikost zprávy 1024 bytů.



Taktéž v grafu 4.8 dochází k výraznému zrychlení, které se ukazuje ale až od velkých velikostí front. Bohužel důvod, proč se zrychlení dostavuje až při takto vysokých hodnotách, se nepodařilo objasnit a vyžaduje asi další a hlubší prozkoumání chování cache pamětí v této konkrétní situaci.

Netestovaný potenciál této optimalizace se možná skrývá v rozdělení na dvě samostatné optimalizace (nápad vznikl až během dokončování této práce), kdy si mohou uchovávat kopii sdíleného ukazatele pouze písaři nebo čtenáři. Mohlo by se tím docílit lepšího zaměření konkrétní optimalizace a neuchovávat kopie sdílených ukazatelů za každé situace.

4.3.5 Mazání nepotřebných řádků cache paměti

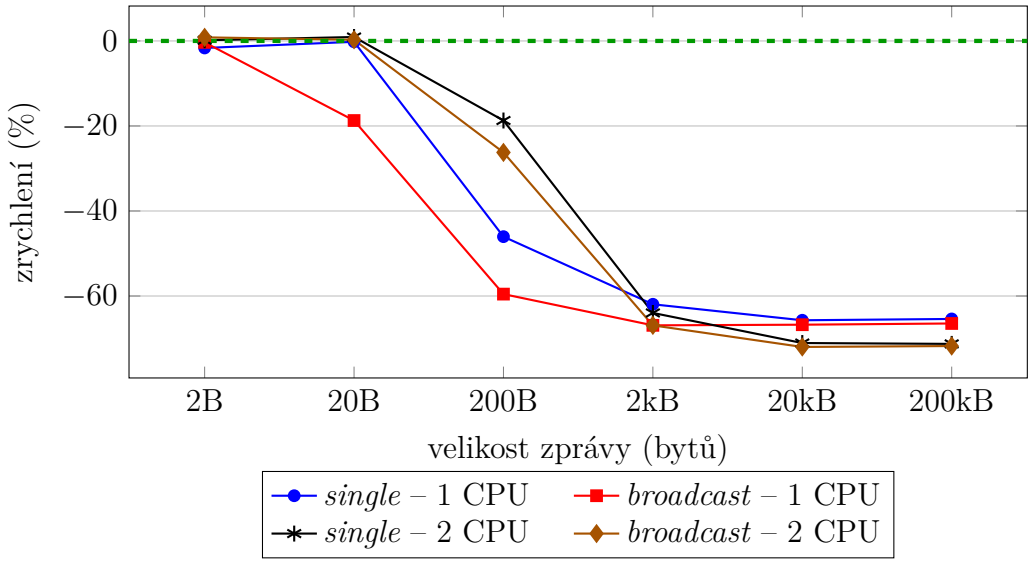
Zavedení optimalizace na mazání nepotřebných řádků cache paměti mělo způsobit efektivní uklízení řádků v paměti, které již nebude vlákno skutečně potřebovat a pro nahrávání dalších řádků bude co nejvíce volných míst. Z výsledků měření v grafu 4.9 ale jasně plyne, že tato operace zbytečně zatěžuje systém instrukcemi navíc a zdržuje běh aplikace. Během návrhu této optimalizace nebylo zřejmé, že by explicitně vyvolaná operace na mazání řádku cache paměti způsobovala v systému tak velikou režii. Rozhodnutí o odstranění konkrétní řádky z cache paměti by se tedy mělo přenechat až na rozhodnutí algoritmu pro správu cache paměti.

4.3.6 Přednačítání ukazatelů

Další testovanou optimalizací je přednačítání ukazatelů do cache paměti, kdy lze s jistotou předpovědět, jakou další zprávu bude čtenář číst nebo do kterého místa bude písař zapisovat. Této optimalizace lze pouze využít, pokud je v konfiguraci jen jeden písař, jeden čtenář nebo mód čtení je nastaven na *broadcast* či *barrier*.

Bohužel měření výsledků v grafu 4.10 ukazuje, že přednačítání ukazatelů nejblíže k jádru do L1 cache paměti způsobuje naopak velmi mírné zpomalení. Důvodem může být, že před zavoláním funkce `__builtin_prefetch` je nutné navíc předpočítat index v poli cyklické fronty, který bude vlákno potřebovat při dalším vkládání nebo vybírání zprávy. Zde jistě dochází ke

Obrázek 4.9: Graf zrychlení optimalizace *mazání řádků cache paměti* závislý na velikosti fronty při různých módech čtení. Další parametry testu: konfigurace W4R4, velikost fronty 8192.



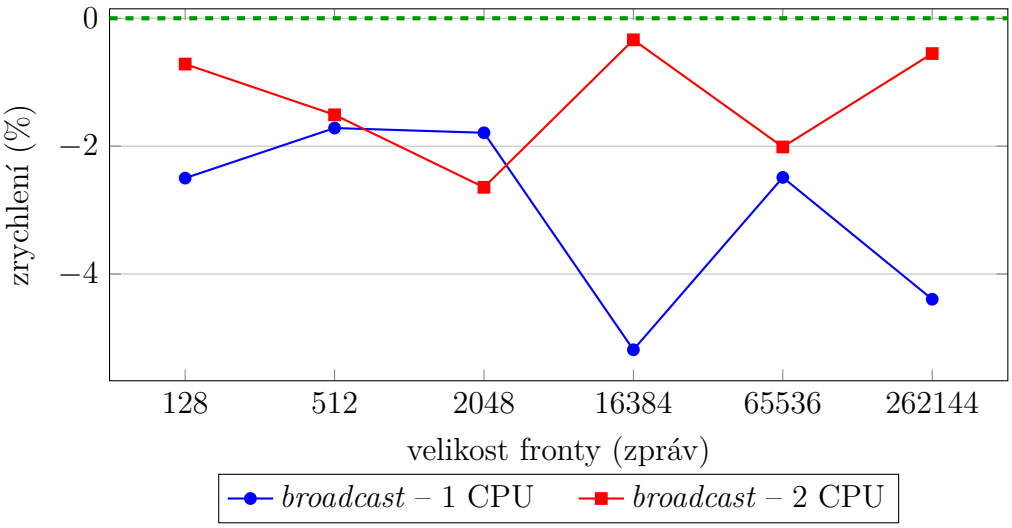
zbytečné režii navíc a mírnému zpomalení. Z měření tedy plyne, že by se měla organizace načítání řádku do paměti přenechat až na rozhodnutí algoritmu pro správu cache pamětí.

4.3.7 Hromadné vkládání a vybírání zpráv z fronty

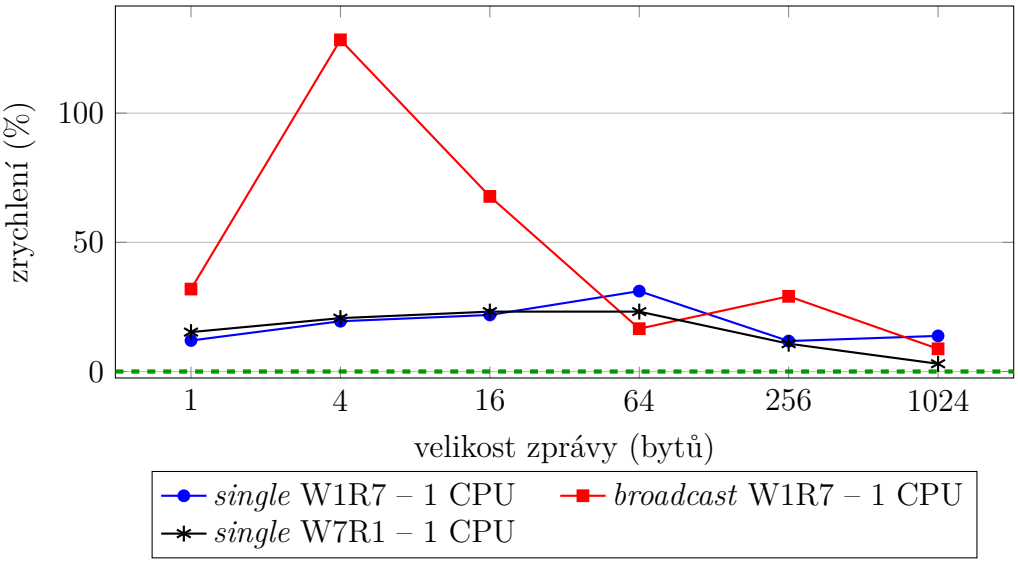
Poslední testovanou optimalizací je hromadné vkládání a vybírání zpráv z fronty. Pokud je přísun nových zpráv do fronty velmi nestálý nebo se v některých chvílích i zastavuje, tak tato optimalizace způsobuje nechtěné zdržení zpráv u písaře, než mohou být zpracovány čtenáři. Způsobuje také problém v opačné situaci, když je čtenář sám a ve frontě se nevyskytují žádné zprávy, tak se drží balík již přečtených a vrácených bufferů ve frontě a písaři je ještě nemohou použít pro zápis.

Ovšem pokud se takové situace vyskytují pouze výjimečně nebo mírné zdržení zpráv nevadí (v této testovací aplikaci jsou vkládány a vybírány zprávy neustále), tak optimalizace hromadného vkládání a vybírání poskytuje velmi výrazné celkové navýšení propustnosti cyklické

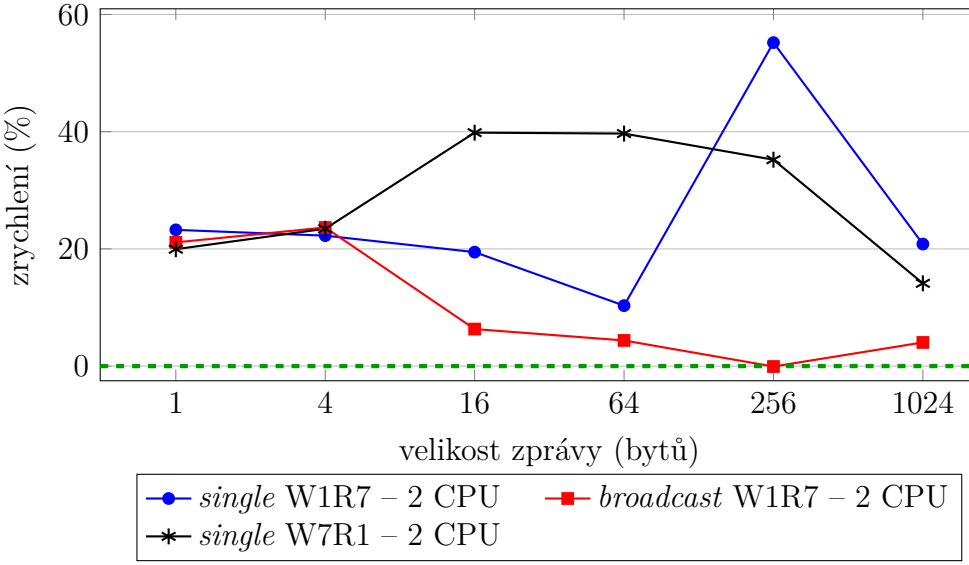
Obrázek 4.10: Graf závislosti zrychlení optimalizace *přednačítání ukazatelů* na velikosti fronty. Další parametry testu: konfigurace W4R4, velikost zprávy 256 bytů.



Obrázek 4.11: Graf závislosti zrychlení optimalizace *hromadné vkládání a vybírání* na velikosti zprávy při různých módech čtení. Další parametry testu: písaři a čtenáři na jednom procesoru, velikost fronty 8192, BATCH_SIZE 20.



Obrázek 4.12: Graf závislosti zrychlení optimalizace *hromadné vkládání a vybírání* na velikosti zprávy při různých módech čtení. Další parametry testu: písaři a čtenáři rozděleni na dva procesory, velikost fronty 8192, BATCH_SIZE 20.



fronty patrné v grafech zrychlení 4.11 a 4.12, které probíhají při rozdílném rozestavení písařů a čtenářů na jeden a dva procesory. V žádné z testovaných situací nezpůsobila tato optimalizace zpomalení běhu aplikace a zrychlení se pohybovalo nejčastěji v rozmezí 20 % až 30 %. Jak již bylo vysvětleno v kapitole s návrhy optimalizací, tak hromadné vkládání a vybírání je ale limitováno pouze na použití v konfiguraci s jedním písařem nebo jedním čtenářem.

4.4 Shrnutí

Z implementovaných optimalizací se většina v měření potvrdila jako skutečný výkonnostní přínos pro *CQ*. U dvou ze všech optimalizací, mazání řádků cache paměti (CLF) a kopírování bez ovlivnění cache paměti (NTM), se ale ukázalo, že zapříčinily naopak zpomalení způsobené různými vlivy, kterým nebyl při návrhu optimalizací přisuzován tak velký podíl na změně propustnosti cyklické fronty. Jedna z optimalizací, přednačítání řádků

(PRE), víceméně v testovaných případech neovlivnila propustnost v žádném z měření.

V tabulce 4.1 závěrečného srovnání optimalizací je zajímavý výběr z ukázkových chování *CQ* při různých kombinacích optimalizací v závislosti na rozdílných parametrech. Nejvhodnější kombinace optimalizací pro běh aplikace lze rozdělit na dvě skupiny. V obou se vyskytuje zarovnání sdílených struktur na začátek řádky cache paměti (CCA) a opožděné obnovení sdílených ukazatelů (LATE), které za všech měřených okolností poskytují skutečné navýšení propustnosti.

Pokud se v některé z kombinací vyskytuje jeden písař nebo jeden čtenář, tak optimalizace na hromadné vkládání a vybírání zpráv (BATCH) vždy pomáhá navýšit propustnost cyklické fronty. Taktéž optimalizace work-stealing (WST) výrazně zvyšuje propustnost ve všech konfiguracích, kde čtenáři nestíhají vybírat zprávy z fronty a pracují v režimu čtení *single*.

Tabulka 4.1: Ukázka nejlepších zapnutých kombinací optimalizací zjištěných během opakovaného spouštění testovací aplikace při zkoušení různém nastavení parametrů.

konfigurace vláken	*rozmístění vláken*	*režim čtení*	*velikost fronty (zpráv)*	*velikost zprávy (bytů)*	*zapnuté optimalizace*	*zrychlení*
W1 R7	1 procesor	single	8192	16	BATCH, CCA, LATE	143 %
W1 R15	1 procesor	single	8192	4	BATCH, CCA, LATE	56 %
W6 R2	1 procesor	single	8192	16	CCA, LATE, WST	138 %
W6 R2	1 procesor	single	8192	2015	CCA, LATE, WST	162 %
W1 R1	1 procesor	single	8192	16	BATCH, CCA, LATE	91 %
W7 R1	1 procesor	single	32768	32	BATCH, CCA, LATE	49 %
W1 R7	1 procesor	broadcast	8192	32	BATCH, CCA, LATE	70 %
W4 R4	1 procesor	broadcast	4096	32	CCA, LATE	81 %
W6 R6	2 procesory	single	8192	32	BATCH, CCA, LATE	439 %
W1 R1	2 procesory	single	131072	16	BATCH, CCA, LATE	622 %
W6 R2	2 procesory	single	8192	8192	CCA, LATE, WST	160 %
W7 R1	2 procesory	single	32768	32	BATCH, CCA, LATE	102 %
W1 R7	2 procesory	broadcast	8192	32	BATCH, CCA, LATE	54 %

Závěr

Tato diplomová práce se snaží poskytnout ucelený pohled na problematiku paralelizace přístupů do cyklické fronty a s ní spojené různé synchronizační techniky pro koordinaci vláken.

V rámci práce jsou navrhnuty a implementovány optimalizace pro paralelní vkládání a zpracování zpráv za účelem vyšší propustnosti fronty. Všechny optimalizace jsou podrobeny důkladnému testování, ze kterého vycházejí některé jako pozitivní přínos pro *CQ* a některé naopak jako negativní přínos. Optimalizace s pozitivním nárůstem propustnosti budou definitivně začleněny do zdrojového kódu *CQ* a například optimalizace pro opožděné obnovení ukazatelů bude vložena do podmíněných překladů pro kompilátor, aby bylo možné tuto optimalizaci případně vynechat a nekontrolovat její vypnutí až za běhu aplikace.

V neposlední řadě je tato práce koncipována jako vysvětlující zdroj budoucím studentům předmětu Vícejádrové systémy (MI-MCS) na FIT ČVUT pro pochopení celé funkčnosti optimalizované cyklické fronty *CQ*, která vznikla právě během výuky tohoto předmětu a bude se využívat i v následujících letech na semestrálních projektech. Naplánovány jsou typové úlohy směřující na použití výpočetního modelu pipeline, jako jsou například paralelní zálohování dat či paralelní generování prvočísel (Eratosthenovo síto).

Literatura

[1] HERLIHY, M.; SHAVIT, N.: *The Art of Multiprocessor Programming, Revised Reprint*. Burlington: Elsevier Science, první vydání, 2012, ISBN 978-0-12397-795-3.

[2] MATTSON, T.; SANDERS, B.; MASSINGILL, B.: *Patterns for Parallel Programming*. Addison-Wesley Professional, první vydání, 2004, ISBN 978-0-32122-811-6.

[3] LAMPORT, L.: Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 1977: s. 125–143, ISSN 0098-5589. Dostupné z: `http://ieeexplore.ieee.org/document/1702415`

[4] Free Software Foundation, Inc.: *GCC online documentation [online]*. 2014, [cit. 2017-03-09]. Dostupné z: `https://gcc.gnu.org/onlinedocs`

[5] VERMA, M.: *Scalable and Performance – Critical Data Structures for Multicores*. Universidade de Lisboa, Universidade de Lisboa, 2013.

[6] SORIN, D. J.; HILL, M. D.; WOOD, D. A.: *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, první vydání, 2011, ISBN 978-1-60845-564-5.

[7] LAMPORT, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on*

Computers, 1979: s. 690–691, ISSN 0018-9340. Dostupné z: http://ieeexplore.ieee.org/document/599898

[8] Intel Corporation: *Intel® 64 Architecture Memory Ordering White Paper*. [cit. 2017-03-09]. Dostupné z: http://www.cs.cmu.edu/~410-f10/doc/Intel_Reordering_318147.pdf

[9] ZICCARDI, M.: Modulo and Division vs Bitwise Operations. [online], [cit. 2017-03-09]. Dostupné z: http://mziccard.me/2015/05/08/modulo-and-division-vs-bitwise-operations

[10] IEEE & The Open Group: *IEEE Standard for Information Technology – Portable Operating System Interface (POSIX) Base Specifications [online]*. 2008, [cit. 2017-03-11]. Dostupné z: http://pubs.opengroup.org/onlinepubs/9699919799

[11] Boost: *Boost C++ Libraries [online]*. 2017, [cit. 2017-04-11]. Dostupné z: http://www.boost.org

[12] YANG, C.; MELLOR-CRUMMEY, J.: A Wait-free Queue As Fast As Fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA: ACM, 2016, ISBN 978-1-4503-4092-2. Dostupné z: http://doi.acm.org/10.1145/2851141.2851168

[13] FRECHILLA, F.: Yet another implementation of a lock-free circular array queue. [online], [cit. 2017-03-09]. Dostupné z: https://www.codeproject.com/articles/153898/yet-another-implementation-of-a-lock-free-circular

Příloha **A**

Seznam použitých zkratek

BATCH hromadné vkládání a vybírání zpráv

CAS compare_and_swap

CCA zarovnání sdílených struktur

CLF mazání nepotřebných řádků cache paměti

CQ circular queue (konkrétní implementace cyklické fronty)

ČVUT České vysoké učení technické v Praze

FIT Fakulta informačních technologií

LATE odkládání čtení sdílených ukazatelů

NTM kopírování s obcházením cache paměti

POSIX Portable Operating System Interface

PRE přednačítání ukazatelů

R reader, čtenář

W writer, písař

WST work-stealing

Příloha B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu XeLaTeX
- `testing`
  - `machine_information.txt` ..... konfigurace testovacího serveru
  - `testing_optimalizace.sh` ..... skript pro testování optimalizací
  - `testing_optimalizace.txt` ..... hodnoty z testování optimalizací
  - `testing_parametry.sh` ..... skript pro testování parametrů
  - `testing_parametry.txt` ..... hodnoty z testování parametrů
  - `testing_srovnani.sh` ..... skript pro srovnání variant *CQ*
  - `testing_srovnani.txt` ..... hodnoty ze srovnání variant *CQ*
  - `testing_script.sh` ..... skript pro opakované spouštění aplikace
- `text` ..... text práce
  - `DP_Kucera_Josef_2017.pdf` ..... text práce ve formátu PDF