ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ
УНИВЕРЗИТЕТА У БЕОГРАДУ

ДИПЛОМСКИ РАД

СОФТВЕРСКО ОКРУЖЕЊЕ ЗА СИМУЛАЦИЈУ МУЛТИПРОЦЕСОРСКИХ СИСТЕМА СА ДИСТРИБУИРАНОМ ДЕЉЕНОМ МЕМОРИЈОМ

Ментор:
Проф. Др Вељко Милутиновић

Кандидат:
Зоран Димитријевић 24/93

Београд, јул 1999.

САДРЖАЈ:

1. УВОД 3

2. СИСТЕМИ СА ДИСТРИБУИРАНОМ ДЕЉЕНОМ МЕМОРИЈОМ 4

2.1. Основна структура DSM система 4
2.2. Грануларност одржавања конзистенције 5
2.3. Алгоритми за приступ подацима у DSM системима 6
2.4. Протоколи за одржавање конзистенције дељене меморије 6
2.5. Синхронизација у DSM системима 8

3. ПРИМЕНА СИМУЛАЦИЈЕ У ПРОЈЕКТОВАЊУ СИСТЕМА СА ДЕЉЕНОМ МЕМОРИЈОМ 9

3.1. Методи анализе система са дељеном меморијом 9
3.2. Начини реализације симулатора 10

4. DSM LIMES 11

4.1. Структура меморијског симулатора SMP система 11
4.2. Структура меморијског симулатора DSM система 12

5. ДЕТАЉИ РЕАЛИЗАЦИЈЕ 13

5.1. Измене у језгру Limes-а 13
5.2. Алокација дистрибуиране дељене меморије 13
5.3. Симулација приступа дистрибуираној дељеној меморији 14
5.4. Одржавање конзистенције синхронизационих променљива 15
5.5. Симулација интерконекционе мреже 16

6. ЗАКЉУЧАК 17

7. ЗАХВАЛНИЦЕ 18

8. РЕФЕРЕНЦЕ 19

ДОДАТАК 1: ИЗМЕНЕ ЈЕЗГРА LIMES СИМУЛАЦИОНОГ ОКРУЖЕЊА 21

Реализација дистрибуиране дељене меморије 21
Реализација алокације DSM меморије 24

ДОДАТАК 2: ИЗВОРНИ КОД МЕМОРИЈСКОГ СИМУЛАТОРА СИСТЕМА СА ДИСТРИБУИРАНОМ ДЕЉЕНОМ МЕМОРИЈОМ 25

1. Увод

Потребе савременог света за све бржим рачунарским системима и све бржом комуникацијом су огромне. Паралелни рачунарски системи који су у прошлости коришћени и развијани у малом броју, углавном војних и академских установа, су престали да буду скупи и недоступни, и понудили своју велику рачунарску снагу свакоме коме је потребна.

Паралелни рачунар је скуп процесорских елемената који заједно раде и међусобно комуницирају да би велики проблем решили брзо [Culler97]. Паралелна архитектура се може посматрати као проширење конвенционалне архитектуре елементима за комуникацију. Према начину комуникације између процесорских елемената паралелне рачунаре можемо поделити на: (1) системе са прослеђивањем порука (енгл. Message Passing Systems) и (2) системе са дељеном меморијом (енгл. Shared Memory Systems).

Паралелни рачунарски системи са дељеном меморијом се могу поделити на: (1) SMP (engl. Symmetric Shared-Memory Multyprocessors) системе који се састоје од више процесора који физички и логички деле адресни простор и (2) DSM (енгл. Distributed Shared-Memory) системе који се састоје од више процесора који физички не деле меморију, али логички деле адресни простор.

Циљ овог рада је проширење софтверског окружења Limes [Magdic97] да би се осим симулације SMP система омогућила и симулација DSM система.

Овај рад је организован у седам секција. У другој секцији је дат кратак преглед система са дистрибуираном дељеном меморијом. У трећој секцији су описани начини симулације система са дељеном меморијом. У четвртој секцији је приказана структура меморијског симулатора у SMP и DSM системима. У петој секцији су описани детаљи реализације симулатора система са дистрибуираном дељеном меморијом. У шестој секцији је дат закључак, и у седмој референце. У прилогу 1 је дат део изворног кода језгра Limes симулационог окружења који је измењен да би била омогућена симулација DSM система. У прилогу 2 је дат комплетан изворни код меморијског симулатора система са дистрибуираном дељеном меморијом и секвенцијалним моделом за одржавање конзистенције.

2. Системи са дистрибуираном дељеном меморијом

Системи са дистрибуираном дељеном меморијом представљају комбинацију најбољих особина два различита приступа: (1) SMP (енгл. Symmetric Shared-Memory Multiprocessor) и (2) DCS (енгл. Distributed Computer Systems).

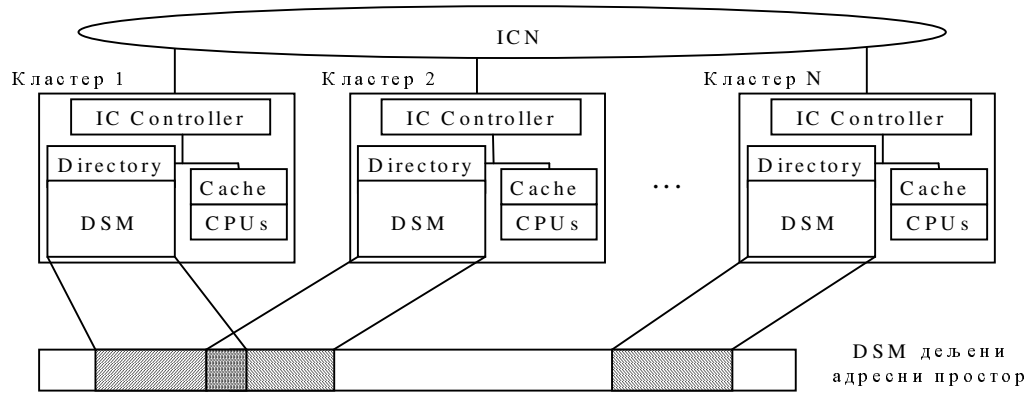
SMP системи имају једну главну добру особину, а то је једноставан програмерски модел, заснован на дељеном адресном простору. Лоша особина ових система је нескалабилност.

Главна добра особина DCS система је скалабилност, а главна лоша особина је програмерски модел заснован на прослеђивању порука (енгл. Message-Passing) и више одвојених адресних простора који је компликован за програмирање.

Системи са дистрибуираном дељеном меморијом комбинују добре особине оба приступа: једноставан програмерски модел заснован на јединственом дељеном адресном простору и скалабилност. Скалабилност је последица чињенице да се делови дељене меморије (логички јединствен адресни простор) физички налазе на различитим чворовима система. Комуникација на најнижем нивоу код система са дистрибуираном дељеном меморијом је остварена путем порука, тако да се ови системи могу посматрати и као проширење DCS система логички јединственим адресним простором. Преглед DSM система може се наћи у [Protic96a].

2.1. Основна структура DSM система

Основна структура DSM система је приказана на слици 2.1. Основа система је системска интерконекциона мрежа (енгл. Interconnection Network - ICN) било ког типа. Неки системи су засновани на минималној интерконекционој мрежи типа магистрале, прстена или локалне мреже - углавном ethernet типа (енгл. Bus, Ring, LAN). Други системи су засновани на максималној интерконекционој мрежи типа grid, mesh или crossbar, или некој хибридној мрежи, која је пројектована поштујући компромис цена-перформансе.



Слика 2.1: Структура и организација DSM система [Protic96a]
Легенда: ICN - интерконекциона мрежа; IC Controller - контролер за приступ интерконекционој мрежи; Directory - директоријум за одржавање конзистенције меморије; DSM - физичка меморија која представља део дистрибуиране дељене меморије у систему; Cache - кеш меморија; CPUs - један или више процесора; DSM дељени адресни простор - приказ логичког адресног простора на коме су мапирани делови физичке меморије.

Чворови DSM система на слици 2.1 су приказани као кластери. Кластер може да буде монопроцесорски или вишепроцесорски (најчешће SMP систем, са додатним интерконекционим контролером). Систем може да буде униформан, са идентичним кластерима, или хибридног типа, са кластерима заснованим на различитим архитектурама.

Основни елементи кластера су: (1) један или више процесора и њихова кеш меморија, (2) део меморије са директоријумом за одржавање конзистенције меморије (енгл. Memory Consistency Maintenance) и (3) интерфејс за повезивање на интерконекциону мрежу.

2.2. Грануларност одржавања конзистенције

Грануларност одржавања конзистенције кеш меморије је типично одређена интерном организацијом микропроцесора и може бити различита у различитим кластерима који чине DSM систем. Величина блока типично варира између 4 и 8 или 16 речи. Протокол за одржавање конзистенције кеш меморије је по правилу реализован у хардверу.

Грануларност одржавања конзистенције дистрибуиране дељене меморије је одређена приликом пројектовања система и може имати различите величине: (1) реч, (2) блок, (3) малу страницу (енгл. page), (4) велику страницу, (5) објекат, (6) сегмент.

Системи који имају реализован део за одржавање конзистенције у хардверу типично користе мању грануларност (реч, блок или мања страница). Ово је могуће јер су хардверска решења ефикасна приликом преноса малих блокова између чворова система. Мала грануларност је добра особина јер побољшава искоришћење меморије, смањује data trashing[1] и false sharing[2].

Софтверски приступи одржавању конзистенције дељене меморије типично користе већу грануларност. Ово је последица чињенице да софтверски системи не могу ефикасно да преносе мале блокове између чворова система. Добра особина је да софтверски системи могу да користе софистициране алгоритме за оптимизацију приступа удаљеним подацима и умање негативне последице коришћења велике грануларности. Софтверски системи могу врло лако користити и семантичке структуре података као што су објекти и сегменти што је веома корисно за неке апликације.

Хибридни системи најчешће користе средње величине јединице за одржавање конзистенције.

[1] Data trashing је енглески термин који означава појаву да се систем јако успори или чак потпуно преоптерети приликом извршавања системских послова као што је одржавање конзистенције.

[2] False sharing означава појаву да се у истом блоку налазе подаци који нису заиста дељени, али пошто се конзистенција меморије одржава на нивоу целог блока, долази до непотребног оптерећења интерконекционе мреже. False sharing се може избећи смештањем података који нису међусобно дељени у различите блокове, али се овим смањује искоришћење меморије.

2.3. Алгоритми за приступ подацима у DSM системима

Постоје три основна алгоритма који се користе у DSM системима: (1) SRSW (енгл. Single Reader Single Writer), (2) MRSW (енгл. Multiple Reader Single Writer) и (3) MRMW (енгл. Multiple Reader Multiple Writer).

SRSW алгоритам подразумева да је само једном чвору дозвољено да чита податке из одређеног блока конзистенције током одређеног временског периода и само једном чвору је дозвољено да уписује податке у тај блок током тог временског периода. Овај алгоритам одговара односима потрошач-произвођач (енгл. consumer-producer). Системи који имају више чворова морају да омогуће механизме за мигрирање блокова. Овај алгоритам се никада не користи као једини алгоритам за приступ, али у појединим системима који имају хардверску подршку за њега, користи се за оне податке који се понашају као SRSW. За оне податке који не испољавају SRSW зависност се користе други алгоритми.

MRSW алгоритам подразумева да неколико чворова може да чита податке из једног блока током одређеног временског периода, али само једном чвору је дозвољено да уписује податке у тај исти блок током тог временског периода. Механизам за мигрирање блокова је потребан за случај када више чворова треба да уписује податке у исти блок.

MRMW алгоритам подразумева да сви или већина чворова може да чита и уписује податке у неки блок без ограничења. У системима у којима интерконекциона мрежа серијализује све уписе у систему, конзистенција података ће бити очувана. У системима у којима то није случај додатне акције морају да се изврше да би се очувала конзистенција.

2.4. Протоколи за одржавање конзистенције дељене меморије

Протоколи за одржавање конзистенције могу бити стриктни (стриктна или секвенцијална конзистенција) или релаксирани (процесорска, слаба, конзистенција при отпуштању, конзистенција лењог отпуштања, улазна, scope конзистенција).

Стриктни модел за одржавање конзистенције меморије подразумева да сви чворови буду тренутно обавештени о свим променама дељених података. Овај модел конзистенције има више теоретски значај осим уколико су кашњења у интерконекционој мрежи занемарљива.

Секвенцијална конзистенција меморије подразумева да сви чворови постану свесни о променама дељених података након кашњења које уноси интерконекциона мрежа. Овај модел одржавања конзистенције је типично подржан у системима са интерконекционом мрежом типа магистрала, прстен или LAN. Сви чворови виде сваки појединачни ток података и глобални ток података у истом редоследу.

Процесорска конзистенција меморије подразумева да сви чворови виде сваки појединачни ток података у истом редоследу, али не и глобални ток података. Овај модел конзистенције типично подржавају системи са интерконекционом мрежом типа магистрала, прстен или LAN са приступним баферима.

Слаба конзистенција (енгл. Weak) подразумева да су специјалне примитиве за синхронизацију укључене у програмски код. Синхронизационе примитиве су: (1) **acquire**, на уласку у критичну секцију и (2) **release**, на изласку из критичне секције. Конзистенција меморије се одржава само када се наиђе на синхронизационе примитиве приликом извршавања програма, и програму није допуштено да настави са извршавањем док се не оствари конзистенција меморије.

Конзистенција при отпуштању (енгл. Release) подразумева да се конзистенција меморије мора остварити само приликом наиласка на синхронизациону примитиву release. Тачке отпуштања (наилазак на примитиву release) морају да поштују правила процесорске конзистенције.

Конзистенција лењог отпуштања (енгл. Lazy Release) подразумева да се конзистенција меморије мора остварити тек приликом наиласка на следећу примитиву acquire. Овим се добија убрзање извршавања апликација, које се мора платити са већим баферима за податке на чворовима система.

Улазна конзистенција (енгл. Entry) подразумева да је свака дељена променљива (или група) заштићена једном синхронизационом променљивом. Увођењем компликованијег програмерског окружења добијено је значајно смањено оптерећење интерконекционе мреже.

AURC (Automatic Update Release Consistency) протокол је реализација конзистенције при отпуштању у којој свака локација у дељеној меморији има свог чвора домаћина (енгл. home node). Протокол користи хардверски акцелератор за аутоматско прослеђивање уписа.

Scope протокол [Iftode96] је покушај да се добре особине улазне конзистенције остваре без увођења новог програмерског окружења. Овај протокол је унапређење AURC протокола.

Детаљнији опис модела и протокола за одржавање конзистенције дељене меморије може се наћи у [Adve96], [Bennett90a], [Bennett90b], [Eggers88], [Gupta92], [Hill98], [Iftode96], [Milutinovic00], [Protic96b], [Protic97], [Tartalja92], [Tartalja97].

2.5. Синхронизација у DSM системима

Конзистенција синхронизационих променљива се мора одржавати независно од одржавања конзистенције обичних дељених променљива. Модел конзистенције синхронизационих променљива мора бити секвенцијална или бар процесорска конзистенција. Приступ синхронизационим променљивама се остварује преко примитива LOCK (acquire) и UNLOCK (release).

У SMP системима постоји јединствена дељена магистрала. Приликом приступања синхронизационим променљивама процесор мора прво да добије приступ магистрали чиме се остварује серијализован приступ дељеним променљивама. Каже се да магистрала серијализује приступ синхронизационим променљивама.

У DSM системима не постоји јединствена магистрала, већ комплексна интерконекциона мрежа. Серијализација приступа синхронизационим променљивама се остварује увођењем менаџера закључавања (енгл. Lock Manager) или на неки други начин. За сваку дељену променљиву постоји менаџер који је одговоран за одржавање конзистенције те дељене променљиве. Менаџер може бити на пример чвор који је власник физичке меморијске локације у којој је смештена синхронизациона променљива. Приликом сваког приступа синхронизационој променљивој захтев се упућује унапред познатом менаџеру који је јединствен, чиме се остварује серијализација приступа.

3. Примена симулације у пројектовању система са дељеном меморијом

3.1. Методи анализе система са дељеном меморијом

Приликом пројектовања нових система и проверавања нових идеја у области архитектуре рачунара користе се аналитички, симулациони и имплементациони поступци.

Аналитички метод подразумева изградњу математичког модела који описује реалан систем. Недостатак овог метода су многобројне апроксимације које су неопходне да би проблем било могуће нумерички решити. Углавном није могуће извести опште закључке користећи искључиво аналитичко моделовање, али оно представља добру основу за процену нових идеја из следећих разлога: (1) аналитичко моделовање помаже да се боље разуме систем који се моделује, (2) релативно је једноставо за реализацију и не захтева скупу опрему, и (3) захтева мање времена него остали методи.

Симулација је други корак приликом анализе система и она омогућава моделовање детаља које није могуће моделовати у математичком моделу. Симулација се врши са мање апроксимација, и може симулирати систем до најситнијих детаља. Симулација омогућава покретање реалних апликација на симулираном систему, што није могуће коришћењем аналитичког метода.

Имплементација подразумева изградњу прототипа реалног система, на коме се могу непосредно извршити сва потребна мерења. Ово је најскупљи и временски најзахтевнији метод. Циљ аналитичких и симулационих метода је да се што боље предвиди понашање реалног система и да се у њега угради само добре идеје и тиме спречи мењање идеја тек након имплементације прототипа. Недостатак имплеметационог метода је у томе што прототип показује само понашање система који је идентичан прототипу. Понашање система са различитим ресурсима, на пример бројем процесора, захтева модификацију прототипа.

Симулација представља најједноставнији и најјефтинији начин да се провере идеје везане за архитектуру мултипроцесорских система са дељеном меморијом. Једино уколико се развија искључиво софтверски DSM систем, изградња прототипа без провере идеја на симулатору је могућа, јер нема потребе да се развија специјални хардвер. Међутим, уколико се развија хардверски или хибридни DSM систем, пре изградње прототипа потребно је извршити детаљну анализу система на симулатору.

3.2. Начини реализације симулатора

Симулатори система који користе реалне програме као оптерећење могу користити: (1) симулацију при извршавању (енгл. execution-driven simulation), (2) интерпретацију кода (енгл. code interpretation) и (3) симулацију покретану траговима извршавања (енгл. trace-driven simulation).

Симулација при извршавању подразумева да су у машински код апликације која се извршава убачени позиви симулатору. Поступак убацивања позива симулатору у код се назива аугментација кода (енгл. augmentation). Примери симулатора који користе овај метод симулације могу се наћи у [Magdic97] и [Herrod96].

Интерпретација кода подразумева да се машински код апликације која се извршава интерпретира у симулатору. Овај метод се користи приликом симулације на процесору који није заснован на истој архитектури која се симулира. Предност овог метода је у томе што није потребно поновно превођење изворног кода апликације. Овај метод је углавном спорији од симулације при извршавању, али се разлика у брзини смањује са повећањем детаљности симулације због чувања контекста потребног приликом сваког позива симулатору у симулацији при извршавању.

Симулација покретана траговима извршавања је најбржи метод симулације меморијског система јер се не извршава реална апликација већ се симулатор покреће већ генерисаним траговима извршавања. Траг (енгл. trace) је запис у коме се налази информација о једном меморијском приступу. Овај начин симулације није прикладан за мултипроцесорске системе, а посебно не за симулацију различитих DSM система са релаксираним моделима за одржавање конзистенције меморије због недетерминизма приликом паралелног извршавања. Више о овој врсти симулатора и о подршци за Limes окружење може се наћи у [Ikodinovic99].

4. DSM Limes

Limes је софтверско окружење за симулацију мултипроцесорских система које је развијено на Електротехничком факултету Универзитета у Београду [Magdic97]. Главна предност Limes окружења у односу на сличне алате (TangoLite [Herrod96] развијен на Стенфорд Универзитету) је у томе што се извршава на оперативном систему Linux на лако доступним и највише коришћеним intel компатибилним процесорима.

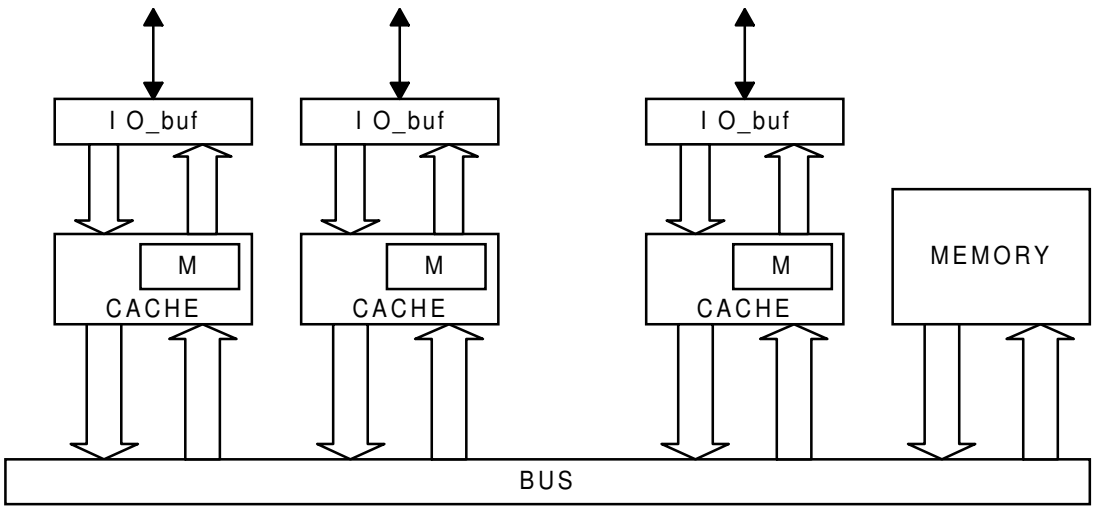
Limes је заснован на симулацији при извршавању (енгл. execution-driven). Оригинални Limes је имао основну подршку за симулацију помоћу трагова (енгл. trace-driven simulation). Софистицирана подршка за симулацију помоћу трагова додата је у [Ikodinovic99].

Детаљно упутство за употребу Limes пакета може се наћи у [Magdic96].

Limes користи ANL макрое за контролу извршавања нити процеса који се користе и у SPLASH и SPLASH2 апликацијама ([Singh92] и [Woo95]).

4.1. Структура меморијског симулатора SMP система

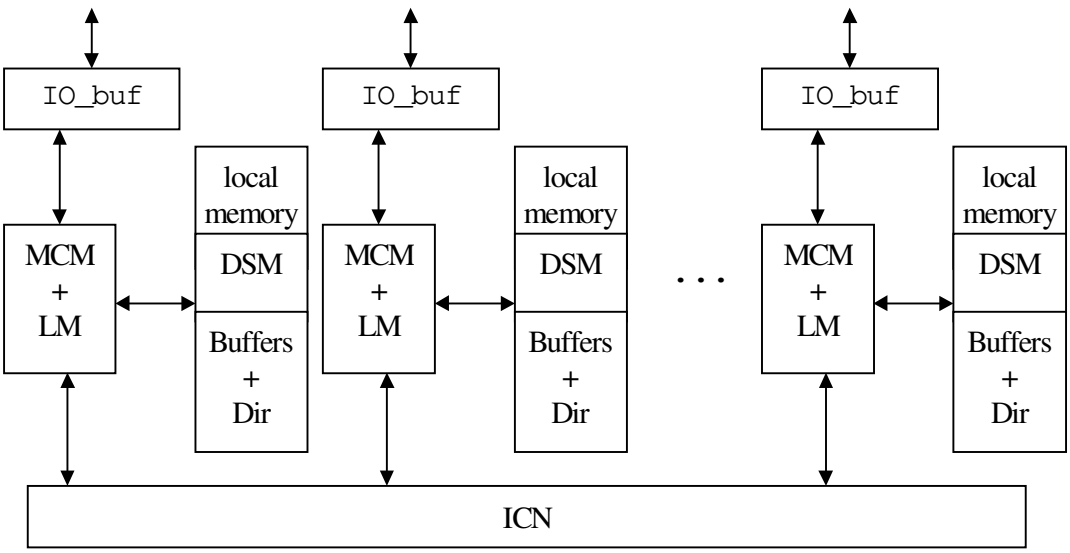
Типична структура меморијског симулатора SMP система приказана је на слици 4.1. Меморијски симулатор комуницира са језгром Limes-а преко класе IO_buf која повезује одговарајући процесор са његовим кеш контролером. Кеш контролер је задужен само за одржавање конзистенције синхронизационих варијабли (lock-ова), и за симулацију приступа дељеној меморији, док је одржавање конзистенције дељене меморије препуштено језгру Limes-а. Ово је задовољавајуће решење за симулацију SMP система, али не одговара детаљној симулацији DSM система са комплексним моделима за одржавање конзистенције меморије. Таква симулација би имала превише конзистентну меморију за DSM систем.



Слика 4.1: Структура меморијског симулатора SMP система
Легенда: BUS - магистрала; Cache - кеш контролер; M - кеш меморија; Memory - заједничка дељена меморија којој се приступа преко магистрале; IO_buf - модул за комуникацију са Limes језгром.

4.2. Структура меморијског симулатора DSM система

Структура меморијског симулатора DSM система приказана је на слици 4.2. Меморијски симулатор комуницира са језгром Limes-а преко класе IO_buf која повезује одговарајући процесор са његовим модулом за одржавање конзистенције. Модул за одржавање конзистенције је задужен за одржавање конзистенције дељене меморије, а његов део, менаџер lock-а је задужен за одржавање конзистенције синхронизационих варијабли. Разлика између SMP и DSM система је првенствено у томе што приступ дељеној меморији код DSM система није униформан - приступ дељеним подацима који се физички налазе у локалној меморији је далеко бржи од приступа дељеним подацима који се физички налазе у меморији неког другог чвора. Због тога се DSM системи називају NUMA (енгл. Non Uniform Memory Access) системи. Да би се побољшале перформансе система користе се релаксирани модели конзистенције меморије и баферисање нелокалних дељених варијабли.

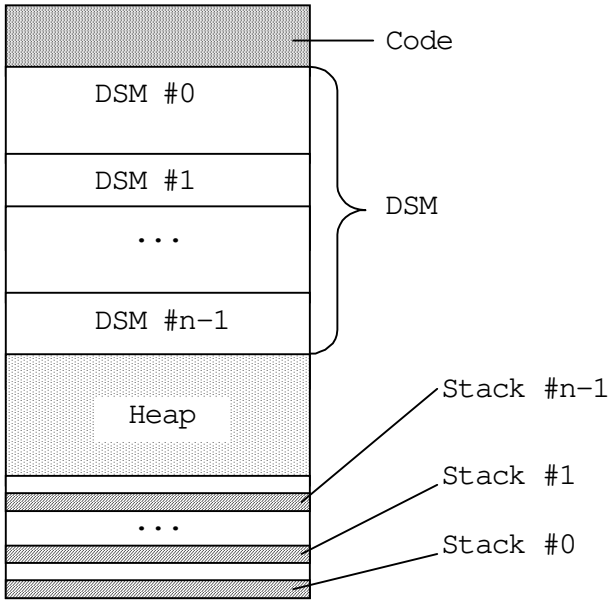


Слика 4.2: Структура меморијског симулатора DSM система
Легенда: ICN - интерконекциона мрежа; MCM+LM - модул за одржавање конзистенције меморије и менаџер lock-а; local memory - локална меморија у којој се налазе локације које нису уопште дељене; DSM - део глобалне дистрибуиране дељене меморије; Buffers+Dir - део меморије који служи за убрзавање приступа локацијама дистрибуиране дељене меморије које се физички налазе у меморији неког нелокалног чвора; IO_buf - модул за комуникацију са Limes језгром.

5. Детаљи реализације

5.1. Измене у језгру Limes-a

Језгру Limes-a је додата дистрибуирана дељена меморија која се иницијализује приликом иницијализације система у kernel.c (погледати прилог 1). Адресни простор симулатора је приказан на слици 5.1.



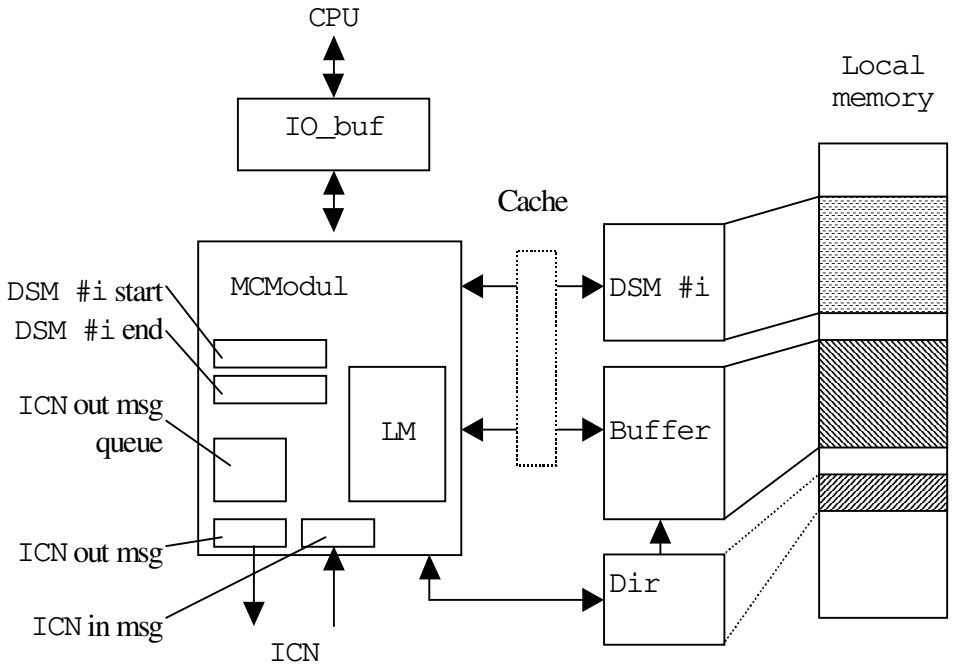
Слика 5.1: Адресни простор DSM Limes симулатора
Легенда: Code - део меморије који садржи код сегмент; DSM - дистрибуирана дељена меморија; DSM #i - сваком чвору i у DSM систему је додељен део јединствене дељене меморије; Heap - простор у коме језго алоцира меморију за потребе симулатора и у коме сваки чвор може да алоцира недељене податке; Stack #i - локални стек за чвор i.

5.2. Алокација дистрибуиране дељене меморије

Сваки чвор може да алоцира простор у дистрибуираној дељеној меморији помоћу макро позива G_MALLOC. Уколико се приликом позива не специфицира чвор на коме треба да се алоцира простор, алоцира се простор у делу DSM меморије која се физички налази у локалној меморији чвора који је позвао макро. Детаљи реализације алокације меморије се могу наћи у прилогу 1.

5.3. Симулација приступа дистрибуираној дељеној меморији

Структура симулатора је приказана на слици 4.2. Детаљнији приказ једног чвора DSM система је приказан на слици 5.3.



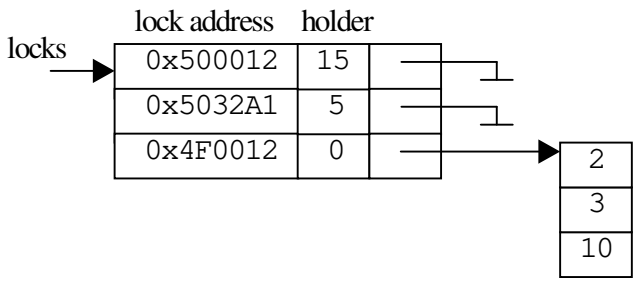
Слика 5.3: Чвор DSM система
Легенда: CPU - процесор; IO_buf - У/И бафер за комуникацију; DSM #i - део дељене меморије који се налази у локалној меморији чвора i; Buffer - део локалне меморије у којој се привремено смештају странице нелокалне DSM меморије приликом приступа; Dir - директоријум за одржавање конзистенције меморије; MCModul - модул задужен за приступ дељеној меморији и одржавање конзистенције; LM - менаџер лока; DSM#i start - регистар у коме се налази показивач на почетак DSM меморије која се налази у меморији локалној за дати чвор; DSM#i end - показивач на крај локалне DSM меморије; ICN out msq queue - ред чекања за поруке које чвор шаље преко интерконекционе мреже; ICN out msg - локација у коју се смешта порука за слање у првом следећем циклусу интерконекционе мреже; ICN in msg - локација у коју се смешта улазна порука која се прима преко интерконекционе мреже.

Модул MCModul је задужен за симулацију приступа дистрибуираној дељеној меморији. Овај модул прима захтеве од језгра Limes окружења, и то: (1) READ REQUEST, (2) WRITE REQUEST, (3) LOCK REQUEST и (3) UNLOCK REQUEST, односно захтеви за читање, писање, узимање и ослобађање лока, респективно. На основу садржаја регистара local_dsm_start и local_dsm_end модул закључује да ли се ради о локацији која се физички налази у локалној меморији чвора. Уколико се налази, онда се меморијска трансакција одради без изласка са захтевом на интерконекциону мрежу. Уколико се локација не налази у локалној меморији прво се одреди који чвор је чвор домаћин за тражену локацију (енгл. home node). Ово се одређује позивом функције find_dsm_node(bitsADDR). Симулатор тренутно не омогућава мигрирање страница, тако да је функција за одређивање чвора домаћина једноставна, јер сваки чвор увек зна ко је домаћин свакој локацији у дељеној меморији. Након одређивања чвора домаћина модул шаље захтев преко интерконекционе мреже и чека одговор. Када модул прими одговор, процесор добија одговор на тражену меморијску трансакцију.

Уколико се симулира неки релаксирани модел за одржавање конзистенције, може се користити баферисање нелокалних страница дистрибуиране дељене меморије. У зависности од протокола за одржавање конзистенције, једна страница се у једном тренутку не налази само на једном месту. Страница може бити баферисана на више чворова DSM система. Оваквим протоколом се убрзава приступ нелокалним подацима у дељеној меморији, јер није увек потребно да се приступа интерконекционој мрежи, али се повећава оптерећење на интерконекционој мрежи, јер се не пребацују само неопходни подаци, већ читави блокови. Да би се остварила конзистенција дељене меморије користе се сложени алгоритми, јер у једном тренутку различити чворови виде различит садржај дељене меморије.

5.4. Одржавање конзистенције синхронизационих променљива

Синхронизација се одржава апстракцијом лока менаџером лока. Детаљна реализација се може видети у прилогу 2. Свакој синхронизационој варијабли је додељен њен менаџер, односно чвор који је домаћин за дату локацију у дељеној меморији. Када процесор захтева лок, уколико је менаџер лока управо тај чвор, менаџер провери да ли је лок слободан или не, и уколико јесте лок додели процесору који га је захтевао. Уколико је лок заузет, менаџер ставља процесор који га је захтевао на крај реда чекања на посматраном року. Уколико је менаџер лока неки други чвор, шаље се захтев преко интерконекционе мреже и чека на доделу захтеваног лока. Приликом захтева за ослобађањем лока примењује се слична процедура. Уколико ред чекања на року није празан, менаџер приликом ослобађања лок додељује процесору који је први у реду за чекање. На слици 5.4 је приказана реализација менаџера лока помоћу динамичке листе.



Слика 5.4: Менаџер лока
Објашњење: На слици ја приказана ситуација у којој су три лока заузета, на локацијама 0x500012, 0x5032A1 и 0x4F0012, од стране процесора 15, 5 и 0, респективно. Ред чекања за прва два лока је празан, а на трећем чекају процесори 2, 3 и 10. Процесор 2 ће први добити лок.

5.5. Симулација интерконекционе мреже

Интерконекциона мрежа је задужена да проследи поруке између чворова. У зависности од жељене детаљности симулације, могуће је симулирати понашање интерконекционе мреже до најситнијих детаља. Чвор уписује поруку за слање у улазни порт, и чита поруке из излазног порта интерконекционе мреже.

Типови порука који се прослеђују кроз интерконекциону мрежу су дефинисани на следећи начин:

```
enum ICN_Message_Type {
    READ_REQUEST,
    READ_RESPONSE,
    WRITE_REQUEST,
    WRITE_ACK,
    LOCK_REQUEST,
    LOCK_GRANTED,
    UNLOCK_REQUEST,
    UNLOCK_ACK
    };
```

Детаљи реализације се могу наћи у прилогу 2.

6. Закључак

Симулатор DSM система се може користити за проверу нових идеја везаних за архитектуру и организацију рачунарских система са дистрибуираном дељеном меморијом. Симулационо окружење Limes има јаку подршку за симулацију SMP система, тако да симулација DSM система са чворовима који су комплетни SMP системи не представља проблем.

Limes симулационо окружење са подршком за DSM системе и разне моделе одржавања конзистенције дељене меморије може се користити у оцењивању ефикасности паралелних алгоритама и њихове примене на одређеним архитектурама. Више о оцењивању ефикасности алгоритама у системима са дељеном меморијом може се наћи у [Agarwal88], [Conte90], [Kessler89], и [Marinov97].

Симулатор DSM система који је написан је само полазна основа за писање симулатора реалних система. Потребно је написати симулаторе за неколико релаксираних модела за одржавање конзистенције дељене меморије и омогућити да чвор DSM система буде комплетан SMP систем. Такође је потребно написати детаљне симулаторе најчешће коришћених типова интерконекционе мреже.

Комплетан изворни код симулатора може се наћи на:
`http://galeb.etf.bg.ac.yu/~zorand/dsmlimes`

7. Захвалнице

Првенствено би желео да се захвалим **Проф. Др. Вељку Милутиновићу** на свему што сам научио током рада у групи коју он води и на разумевању за моја честа кашњења приликом израде разних извештаја. Посебно му се захваљујем за организовање многобројних предавања врхунских светских стручњака који су ми омогућили да стекнем далеко бољу слику тренутног стања у разним областима рачунарских наука.

Велику захвалност дугујем **Мр. Александру Миленковићу** и колеги **Игору Икодиновићу** на драгоценим саветима и посвећеном времену током израде овог рада. Посебно се захваљујем колеги и пријатељу **Давору Магдићу** на многобројним дискусијама током дуготрајне сарадње.

8. Референце

[Adve96] Adve, S., Gharachorloo, K., "Shared Memory Consistency Models: A Tutorial," *IEEE Computer,* Vol. 29, No. 12, December 1996, pp. 66-76.

[Agarwal88] Agarwal, A., Gupta, A., "Memory-reference characteristics of multiprocessor applications under MACH," *Proceedings of the ACM SIGMETRICS Conference*, May 1988, pp. 215-225.

[Bennett90a] Bennett, J. K., Carter, J. B., Zwaenepoel, W., "Adaptive Software Cache Management for Distributed Shared Memory Architectures," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 125-134.

[Bennett90b] Bennett, J. K., Carter, J. B., Zwaenepoel, W., "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," CPPPP, 1990, pp. 168-176.

[Conte90] Conte, T. M., Hwu, W. W., "Benchmark characterization for experimental system evaluation," *Proceedings of the 23rd Annual Hawaii International Conference on System Sciences*, 1990, Vol. 1, pp. 6-18.

[Culler97] Culler, D., Singh, P. J., Gupta, A., "Parallel Computer Architecture," Morgan Kaufmann Puplishers, 1997.

[Eggers88] Eggers, S. J., Katz, R. H., "A characterization of sharing in parallel programs and its application to coherence protocol evaluation," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, May 1988, pp. 373-382.

[Gupta92] Gupta, A., Weber, W. D., "Cache invalidation patterns in shared-memory multiprocessors," *IEEE Transactions on Computers*, Vol. 41, No. 7, July 1992, pp. 794-810, & Correction in Vol. 41, No. 12, pp. 1631-1632.

[Herrod93] Herrod, S. A., "TangoLite: Introduction and User's Guide," Technical Report, Stanford University, Palo Alto, California, USA, November 1993.

[Hill98] Hill, D. M., "Multiprocessors Should Support Simple Memory-Consistency Models," *IEEE Computer,* August 1998, pp. 28-34.

[Iftode96] Iftode, L., Singh, J. P., Li, K., "Scope Consistency: A Bridge Between Release Consistency and Entry Consistency," *Proceedings of the 8th Annual Symposium on Parallel Algorithms and Architectures*, June 1996.

[Ikodinovic99] Ikodinovic, I.., "Trace-driven simulator multiprocesorskih sistema", Diplomski rad, Univerzitet u Beogradu, Beograd, Jugoslavija, 1999.

[Kessler89] Kessler, R. E., Livny, M., "An Analysis of Distributed Shared Memory Algorithms," *Proceedings of the 9th International Conference on Distributed Computing Systems,* June 1989, pp. 498-505.

[Magdic96] Magdic, D., "Limes User Guide," Technical Report, University of Belgrade, Belgrade, Yugoslavia, 1996.

[Magdic97] Magdic, D., "Limes: A Multiprocessor Simulation Environment," *IEEE TCCA Newsletter*, March 1997, pp. 68-71.

[Marinov97] Marinov, D., Magdic, D., Milenkovic, A., Protic, J., Tartalja, I., Milutinovic, V. "Characterization of Parallel Workload in VSM Systems", *Proceedings of the 31st Annual Hawaii International Conference on System Sciences,* Hawaii, USA, January 1998.

[Milutinovic97] Milutinovic, V., Milenkovic, A., "Cache Injection Control Architecture," *Proceedings of the 5th MASCOTS*, Haifa, Israel, January 1997.

[Milutinovic00] Milutinovic, V., "Microprocessor and Multimicroprocessor Systems," Copyright by Wiley, USA, 2000.

[Protic96a] Protic, J., Tomasevic, M., Milutinovic, V., "Distributed Shared Memory: Concepts and Systems," *IEEE Parallel and Distributed Technology,* Vol. 5, No. 1, Summer 1996.

[Protic96b] Protic, J., Tartalja, I., Tomasevic, M., "Memory Consistency Models for Shared Memory Multiprocessors and DSM Systems," *Proceedings of the 8th Mediterranean Electrotechnical Conference melecon '96,* Bari, Italy, May 1996.

[Protic97] Protic, J., Milutinovic, V., "Entry Consistency versus Lazy Release Consistency in DSM Systems: Analytical Comparison and a New Hybrid Solution," *Proceedings of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems FTDCS'97,* October 1997, Tunis, Tunisia.

[Singh92] Singh, J. P., Weber, W. D., Gupta, A., "SPLASH: Stanford parallel applications for shared-memory," *Computer Architecture News,* Vol. 20, No. 1, March 1992, pp. 5-44.

[Tartalja92] Tartalja, I., Milutinović, V., "An approach to dynamic software cache consistency maintenance based on conditional invalidation," *Proceedings of the 25th Annual Hawaii International Conference on System Sciences*, IEEE Computer Society Press, Los Alamitos, California, January 1992, pp. 457-466.

[Tartalja97] Tartalja, I., "Dinamičko softversko održavanje konzistencije keš memorija zasnovano na uslovnom samo-poništavanju privatnih kopija zajedničkih segmenata podataka," Doktorska disertacija, Univerzitet u Beogradu, Beograd, Jugoslavija, Februar 1997.

[Woo95] Woo S. C., Ohara M., Torrie E., Singh J. P., Gupta A., "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995, pp. 24-36.

Додатак 1: Измене језгра Limes симулационог окружења

Реализација дистрибуиране дељене меморије

preferences.h

```
...
/*
 * SMP or DSM simulation
 * Change globals.make for DSM or SMP simulation
 */

/*
 * Maximum number of processors in the system.
 */
#define MAX_CPU_COUNT 32

/*
 * DSM preferences
 * DSM_SIZE is size of the simulated DSM memory
 */
#define DSM_SIZE 32*1024*1024
...
```

kernel.h

```
...
/*
 * DSM memory
 */
#define DSM_START (bitsADDR)&_etext
#define DATA_HEAP_START (bitsADDR)&_end
extern bitsADDR dsmStart[MAX_CPU_COUNT], dsmEnd[MAX_CPU_COUNT];

/*
 * Macro determinining whether an adress is shared - in the sense that it
 * could be both read and written by all. Locations other than those in the
 * data segment could be shared as well (like constants in the code
 * segment), but they are read only and need not be simulated to the level
 * of contents.
 * Actually, the second line should read "START+MAXSIZE", but since MAXSIZE
 * is 512MB, DATA_SEGMENT_START is much less than DATA_SEGMENT_MAXSIZE.
 */
#ifndef DSM_LIMES
#define ISSHARED(ADDR)                                                  \
    ((bitsADDR)(ADDR) >  DATA_SEGMENT_START &&                          \
     (bitsADDR)(ADDR) < DATA_SEGMENT_MAXSIZE)
#else
#define ISSHARED(ADDR)                                                  \
    ((bitsADDR)(ADDR) >  DSM_START &&                                   \
     (bitsADDR)(ADDR) < dsmEnd[MAX_CPU_COUNT-1])
#endif /* DSM_LIMES */
```

```
/*
 * system-memory allocation policy: kernel and memory simulator heap should
 * be separated from the application's; this is what mmap() is for, called
 * by the sys_malloc() that kernel and memsim should use instead of malloc
 * if mmap() doesn't work on your system, uncomment this and the ordinary
 * malloc() will be used; but there will be no much difference anyway.
 * DSM_LIMES extensions tend to be more allocation intensive and do not use
 * mmap... DSM_SIZE is all you get for shared memory, although application
 * can use malloc for non-shared data.
 */
#ifndef DSM_LIMES
#define USE_MMAP_FOR_SYS_MALLOC
#endif /* DSM_LIMES */
...
```

kernel.c

```
...
static void do_system_init(int *argc, char *argv[]) {
    int i;
    /* set up defaults */
    sys_output = stdout;        /* by default, output goes to stdout */
    sys_debug_events = 0;
    sys_gather_statistics = 0;
 /* parse command line options: "--" option means the rest is for Limes */
    for(i = 1; i < *argc; ++i)
        if (!strcmp(argv[i],"--"))  {
            int appargc = i++;
            startup_argv = &argv[i]; /* remember the rest of the array */
            while(i < *argc) {
                if (argv[i][0] != '-') break;
                switch (argv[i][1]) {
                    case 'e':   /* force output to stderr */
                        sys_output = stderr;    break;
                    case 't':   /* produce trace file */
                        produce_trace_file = 1; break;
                    case 's':   /* statistics on */
                        sys_gather_statistics = 1;  break;
                    case 'i':   /* INI file other than limes.ini */
                        INIfile = argv[i+++1];      break;
                    case 'd':   /* define INI item, form -dITEM=VALUE */
                        break;
                    case 'b':   /* lemme sleep */
                        beep_at_end = 0;    break;
                    default :
                      fprintf(stderr, "Incorrect simulator option. \n");
                      fprintf(stderr, "Syntax: <application> <app args> "\
                            "-- [-e] [-t] [-s] [-i file] [-dKEY=VALUE]\n");
                      fprintf(stderr, "\t-e : force output to stderr "\
                                        "instead of stdout\n");
                      fprintf(stderr, "\t-t : produce trace file\n");
                      fprintf(stderr, "\t-s : dump threads' statistics\n");
                      fprintf(stderr, "\t-i : specifies INI file other "\
                                       "than limes.ini\n");
                      fprintf(stderr, "\t-d : specifies value for an "\
                                       "INI item (overrides .ini value)\n");
                      exit(1);
                    }
                ++i;
                }
               *argc = appargc; argv[appargc] = NULL;
```

```
               /* the app won't see "--" options */
            break;
            }
PRINTF("\n************************************************\n");
    PRINTF("Simulation start: ");
    for(i=0; argv[i]; ++i) PRINTF("%s ",argv[i]);
    if (startup_argv) {
        PRINTF("-- ");
        for(i=0; startup_argv[i]; ++i) PRINTF("%s ",startup_argv[i]);
        }
    PRINTF("\n");
#ifdef DSM_LIMES
    do_dsm_init();
    dsm_malloc_init();
#endif
    sys_init_threads_and_memsim();
    simulation_start = time(NULL);
    PRINTF(
      "MAX_THREAD_COUNT=%d, BUSY_LOCK_TIME=%d, %sBUSY_LOCK_OPTIMIZATION\n",
      MAX_THREAD_COUNT, BUSY_LOCK_SLEEP_TIME, BUSY_LOCK_OPTIMIZATION ? "" :
      "NO ");
    PRINTF("------------------------------------------------------------\n");
    FFLUSH;
    sys_signal_setup();
    }

/*
 * Init the dsm memory and misc
 */
static void do_dsm_init() {
      int i;
      bitsADDR p;

      dsmStart[0]=(bitsADDR)malloc(DSM_SIZE);
    if(!dsmStart[0]) {
        fprintf(stderr,"Could not allocate DSM_MEMORY: "\
                       "try to decrease DSM_SIZE\n");
        exit(1);
        }
    dsmEnd[MAX_CPU_COUNT-1]=dsmStart[0]+DSM_SIZE;
    p=DSM_SIZE/MAX_CPU_COUNT;
      for(i=1;i<MAX_CPU_COUNT;i++) {
            dsmEnd[i-1]=dsmStart[i]=dsmStart[i-1]+p;
            }
/*    DEBUG (
            printf("DSM_START=%d\n",DSM_START);
            printf("DATA_HEAP_START=%d\n",DATA_HEAP_START);
            for(i=0;i<MAX_CPU_COUNT;i++)
                  printf("dsm[%d]=%d - %d\n",i,dsmStart[i],dsmEnd[i]);
            )
*/
    }
```

Реализација алокације DSM меморије

dsm.c

```
#include "sys_stdlib.h"
ALOCKDEC(dsmMallocLock,MAX_CPU_COUNT);
bitsADDR dsmMallocP[MAX_CPU_COUNT];

void dsm_malloc_init() {
      int i;
      ALOCKINIT(dsmMallocLock,MAX_CPU_COUNT);
      for(i=0;i<MAX_CPU_COUNT;i++) dsmMallocP[i]=dsmStart[i];
      }

void *dsm_malloc(size_t allocSize, int n) {
      void *p;

      if(allocSize<=0) return NULL;

      if (n==-1) {
            AUG_OFF
            n=sys_thread_ID;
            AUG_ON
            }
      ALOCK(dsmMallocLock,n);
      p=(void *) dsmMallocP[n];
    if (dsmMallocP[n]+allocSize < dsmEnd[n]) {
      dsmMallocP[n]+=allocSize;
      }
    else {
      p=NULL;
      }
      AULOCK(dsmMallocLock,n);
      return p;
    }

void dsm_free(void *p) {
      /* Not implemented */
    }
```

Додатак 2: Изворни код меморијског симулатора система са дистрибуираном дељеном меморијом

dsm.make

```
CS     = memsim.cc topology.cc io_buf.cc icn.cc mcm.cc lock.cc

CC    = $(GCC) -O2 -g -I$(INCLUDEDIR)
CPP    = $(GCPP) -I$(INCLUDEDIR) -Wall -O2 $(DETACHED)

OS    = $(CS:.cc=.o)

$(SIMLIB): $(OS)
    $(AR) ru $(SIMLIB) $(OS)

$(OS): %.o : %.cc
    $(CPP) -c $< -o $@

dep:
    $(CPP) -MM $(CS) > .depend

ifeq (.depend, $(wildcard .depend))
include .depend
endif

clean:
    rm -rf $(OS) $(SIMLIB)
```

dsm.h

```
/*
 *    DSM LIMES - dsm.h
 *    Author: Zoran Dimitrijevic
 *    zorand@galeb.etf.bg.ac.yu
 */

extern "C" {
#include "memsim_interface.h"
}

// and this: redefinition of new and delete so that all the data needed by
// your simulator is allocated on the system heap, and not the application
// heap
// This is not needed for DSM_LIMES because sys_malloc is same as malloc

static inline void * operator new(size_t size) { return sys_malloc(size);}
static inline void * operator new[](size_t size){ return sys_malloc(size);}
static inline void operator delete(void *p)      { sys_free(p); }
static inline void operator delete[](void *p)    { sys_free(p); }

// dsm memory simulator specifics

#include "icn_interface.h"

#define TOPOLOGY_CLASS          Base_Topology
#define TOPOLOGY_FRIENDS        friend class Base_Topology;
#define TOPOLOGY_HDR            "topology.h"

#define ICN_HDR                 "icn.h"
#define ICN_CLASS               ICN_Module

#define IO_BUF_HDR              "io_buf.h"
#define IO_BUF_CLASS            IO_Buf

#define MCM_HDR                 "mcm.h"
#define MCM_CLASS               MCModul

#endif _DSM_H_
```

icn_interface.h

```
#ifndef _ICN_INTERFACE_H
#define _ICN_INTERFACE_H
/*
 *    DSM LIMES - ICN interface
 *    Author: Zoran Dimitrijevic
 *    zorand@galeb.etf.bg.ac.yu
 */
enum ICN_Message_Type {
    READ_REQUEST,
    READ_RESPONSE,
    WRITE_REQUEST,
    WRITE_ACK,
    LOCK_REQUEST,
    LOCK_GRANTED,
    UNLOCK_REQUEST,
    UNLOCK_ACK
    };

typedef struct {
    enum ICN_Message_Type type;
    bitsMAX data;
    bitsADDR addr;
    int size;
    } ICN_Message;
#endif  // _ICN_INTERFACE_H
```

icn.cc

```
#define PRAGMA_IMPLEMENTATION_ICN_MODULE
#include "icn.h"
```

icn.h

```
#ifndef _ICN_H_
#define _ICN_H_
/*
 *    DSM LIMES - Inter Connection Network class
 *    Author: Zoran Dimitrijevic
 *    zorand@galeb.etf.bg.ac.yu
 */
#include "dsm.h"
#include TOPOLOGY_HDR

// InterConnectionNetwork IO Buffers
struct ICN_IO_Buf {
    struct In {
        int STROBE;
        int dest;
        ICN_Message msg;
        } in;                // to the ICN
    struct Out {
        int STROBE;
        int src;
        ICN_Message msg;
        } out;               // to the MCM
    };
```

```
// Simple InterConnection Network
class ICN_Module {
    const DEFAULT_ICN_DELAY = 10;   // CPU clock - ICN clock rate ratio
    int icn_delay;
    int icn_delay_cnt;
    struct Stats{
        int msgcnt;
        } stats;
  public:
    struct ICN_IO_Buf io_buf[MAX_CPU_COUNT];

    ICN_Module();
    void dump_stats(Thread_Time);
    void cycle();
    void dump_state();
    bool isidle() { return 1; }
      TOPOLOGY_FRIENDS
    };

#ifdef PRAGMA_IMPLEMENTATION_ICN_MODULE
////////////////////////////////////
#include <iostream.h>

ICN_Module::ICN_Module() {
    stats.msgcnt=0;
    icn_delay=DEFAULT_ICN_DELAY;
    icn_delay_cnt=icn_delay;
    for(int i=0;i<MAX_CPU_COUNT;i++) {
        io_buf[i].in.STROBE=0;
        io_buf[i].out.STROBE=0;
        }
    }

void ICN_Module::dump_stats(Thread_Time total_time) {
    if (total_time == 0) return;
    cout<<"ICN stats: MSG_CNT=" << stats.msgcnt << endl;
    }

// This method is called by Topology every topology cycle
void ICN_Module::cycle() {
    if(icn_delay_cnt--) return;
    icn_delay_cnt=icn_delay;
    for(int i=0;i<MAX_CPU_COUNT;i++) {
        if(io_buf[i].in.STROBE)
            if(io_buf[io_buf[i].in.dest].out.STROBE==0) {
                io_buf[io_buf[i].in.dest].out.STROBE=1;
                io_buf[io_buf[i].in.dest].out.msg=io_buf[i].in.msg;
                io_buf[io_buf[i].in.dest].out.src=i;
                io_buf[i].in.STROBE=0;
                stats.msgcnt++;
                }
        }
    }

void ICN_Module::dump_state() {
    }
#endif //PRAGMA_IMPLEMENTATION_ICN
#endif //_ICN_H_
```

io_buf.cc

```
#define PRAGMA_IMPLEMENTATION_IO_BUF
#include "io_buf.h"
```

io_buf.h

```
#ifndef _IO_BUF_H_
#define _IO_BUF_H_

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Module: io_buf.h
 * Author: Davor Magdic, magdic00390d@buef31.etf.bg.ac.yu
 *
 * Changed for DSM limes
 * Author: Zoran Dimitrijevic, zorand@galeb.etf.bg.ac.yu
 *
 * This module isolates the other modules from the Limes kernel.
 * It reads the requests from the kernel, and leaves them on its
 * output ports. It reads the responses from its input ports and
 * transmits them to the kernel.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

#include "dsm.h"
#include TOPOLOGY_HDR

class IO_Buf {
    int index;                  // index of the belonging PE (processing
element)
    bool request_pending;       // 1 (true) if yes, 0 (false) if not
    // output signals
    struct Out {
        // to the MCM
        struct Request {
            struct Memory_Request *request_ptr;     // request description
            int STROBE;         // set by IO_Buf and cleared by Topology
            } request;
        } out;
    struct In {
        // input signals: from the MCM
        struct Response {
            enum PE_response signal;  // filled by topology when called
            bitsMAX data;
            } response;
        } in;
  public:
    IO_Buf();
    void assign_index(int ind)  { index = ind; }
    void start_cycle();
    void end_cycle();
    bool isidle() {return !request_pending;}
    TOPOLOGY_FRIENDS
    };
```

```
#ifdef PRAGMA_IMPLEMENTATION_IO_BUF
//--------------------------------

#include <iostream.h>

IO_Buf::IO_Buf() {
    request_pending = false;
    index = 0;
    out.request.STROBE = 0;
    }

// each IO_Buf module has two main methods: start_cycle() and end_cycle().
// The Topology decides when to call each: at the beginning of the
// simulated cycle, start_cycle() is called for each IO_Buf instance if
// there is an indication that one or more requests exist at this cycle.
// Each IO_Buf checks it, reads the request from the kernel and leaves it
// on its output port, waiting for the cache to collect it. At the end of
// each simulated cycle, end_cycle() is called, which checks if there is
// any response from the underlying cache controller. The response is
// forwarded to the kernel.

void IO_Buf::start_cycle() {
    if (REQUEST_EXISTS(index)) {
        if (request_pending) {
            cerr << "\aIO_Buf[" << index << "] overrun error!\n"; exit(1);}
        REQUEST_FILL(index, out.request.request_ptr);

        // misalignment not allowed! GCC never makes one anyway.
        bitsADDR addr = out.request.request_ptr->virtual_addr;
        int size = out.request.request_ptr->size;
        int mask = ~(DATA_WIDTH-1);
        if ( (addr & mask) != ( (addr+size-1) & mask ) )  {
            cerr << "\aIO_Buf[" << index << "] alignment error!\n";
            exit(1);
            }
        out.request.STROBE  = 1;
        request_pending = true;
        }
    }

void IO_Buf::end_cycle() {
    if (request_pending) {
        if (!Topology::io_buf_input(index))  // if there is no response
            REQUEST_RESPOND(index, PROCESSOR_STALLED);
        else {
            // if there was a read from the CPU, it had waited till now
            fastcpy(&sys_threads[index]->memory_request.data,
                &in.response.data,
                sys_threads[index]->memory_request.size);
            REQUEST_RESPOND(index, in.response.signal);
            request_pending = false;
            }
        }
    else    // if there was no request, there will be no response
        REQUEST_RESPOND(index, PROCESSOR_NORESPONSE);
    }

#endif // PRAGMA_IMPLEMENTATION_IO_BUFF_
#endif //_IO_BUF_H_
```

lock.cc

```
#define PRAGMA_IMPLEMENTATION_LOCK_MANAGER
#include "lock.h"
```

lock.h

```
#ifndef _LOCK_H_
#define _LOCK_H_
/*
 *    DSM LIMES - Lock Manager class
 *    Author: Zoran Dimitrijevic
 *    zorand@galeb.etf.bg.ac.yu
 */
#include "dsm.h"
#include TOPOLOGY_HDR

#include <stdio.h>

class Locks{
  public:
    bitsADDR lockaddr;
    int holder;
    class LockQueue *lq;
    class Locks *next;
    };

class LockQueue{
  public:
    int node;
    class LockQueue *next;
    };

// LockManager class is abstraction of locks for DSM systems
// Locks allways comply to sequentialy MCM
// in fact, this realization for simulation purposes never looks what
// is really in memory location; it is using only addresses of locks
class LockManager {
    Locks *locks;
  public:
      LockManager();
      void addnode(LockQueue * &lq, int node);
      bool lock(bitsADDR, int);
      int unlock(bitsADDR lockaddr);        // returns node to get lock
next or -1 if none
    void printlocks();
    };
```

```
#ifdef PRAGMA_IMPLEMENTATION_LOCK_MANAGER
//---------------------------------

#include <iostream.h>

LockManager::LockManager() {
    locks=NULL;
    }

void LockManager::addnode(LockQueue * &lq, int node) {
    LockQueue *prev=lq, *cur=lq;
    if(lq==NULL) {
        lq=new LockQueue;
        lq->next=NULL;
        lq->node=node;
        }
    else {
        while (cur) {
            prev=cur;
            cur=cur->next;
            }
        prev->next=new LockQueue;
        prev->next->next=NULL;
        prev->next->node=node;
        }
    }

// returns true if lock is granted or false if busy
bool LockManager::lock(bitsADDR lockaddr, int node) {
    Locks *cur=locks, *prev=locks;
    while (cur) {
        if (cur->lockaddr == lockaddr) {
            addnode(cur->lq, node);
            return false;
            }
        prev=cur;
        cur=cur->next;
        }
    if (prev) {
        prev->next=new Locks;
        prev->next->lockaddr=lockaddr;
        prev->next->holder=node;
        prev->next->next=NULL;
        prev->next->lq=NULL;
        return true;
        }
    else {
        locks=new Locks;
        locks->lockaddr=lockaddr;
        locks->holder=node;
        locks->next=NULL;
        locks->lq=NULL;
        return true;
        }
    }
```

```
// returns -1 if nobody is waiting for lock or
// number of the node to be granted the lock next
int LockManager::unlock(bitsADDR lockaddr){
    Locks *cur=locks, *prev=locks;
    LockQueue *lqtmp;
    int node;
    while (cur) {
        if(cur->lockaddr==lockaddr){
            if(cur->lq) {
                node=cur->lq->node;
                lqtmp=cur->lq;
                cur->lq=cur->lq->next;
                delete lqtmp;
                cur->holder=node;
                return node;
                }
            else {
                if(cur==locks){
                    locks=locks->next;
                    delete cur;
                    return -1;
                    }
                else {
                    prev->next=cur->next;
                    delete cur;
                    return -1;
                    }
                }
            }
        prev=cur;
        cur=cur->next;
        }
    printf("Unlock error! - unlock attempt of not locked lock\n");
    return -2;
    }

// print locks - for debuging
void LockManager::printlocks() {
    Locks *cur=locks;
    LockQueue *lqcur;
    printf("locks:\n");
    while(cur) {
        printf("%d:[",cur->lockaddr);
        lqcur=cur->lq;
        while(lqcur){
            printf("%d,",lqcur->node);
            lqcur=lqcur->next;
            }
        printf("]\n");
        cur=cur->next;
        }
    }

#endif // PRAGMA_IMPLEMENTATION_LOCK_MANAGER
#endif //_LOCK_H_
```

mcm.cc

```
#define PRAGMA_IMPLEMENTATION_MCM
#include "mcm.h"
```

mcm.h

```
#ifndef _MCM_H_
#define _MCM_H_
/*
 *    DSM LIMES - Memory Consistency Model Modul class
 *    Author: Zoran Dimitrijevic
 *    zorand@galeb.etf.bg.ac.yu
 */
#include "dsm.h"
#include TOPOLOGY_HDR
#include ICN_HDR
#include "lock.h"

// Queue of ICN messages to be sent
class ICN_Queue {
    struct Queue_Member {
        ICN_Message msg;
        int dest;
        } q[MAX_CPU_COUNT];
    int size;
    int ptr;
  public:
      ICN_Queue() { size=0; ptr=0; }
      bool isempty() { return size?false:true; }
      bool put(int dest, ICN_Message msg) {
          if(size<MAX_CPU_COUNT){
            q[(ptr+size)%MAX_CPU_COUNT].dest=dest;
            q[(ptr+size)%MAX_CPU_COUNT].msg=msg;
            size++;
            if(size>MAX_CPU_COUNT) {
                printf("ICN_Queue buffer overflow !\n");
                exit(-1);
                }
            return true;
        } else return false;
    }
    bool peek(int &dest, ICN_Message &msg) {
        if(!size) return false;
        dest=q[ptr].dest;
        msg=q[ptr].msg;
        return true;
    }
    void get() {
        if(size>0) {
            size--;
            ptr=(ptr+1)%MAX_CPU_COUNT;
        }
    }
};
```

```
// MCM controler module declaration
class MCModul {
    int index;

    // port definitions
    struct In {                          // Input port:
        struct Memory_Request request;   // from the CPU
        } in;
    struct Out {                         // Output port
        struct Response {                // to the CPU

    // signal to the IO_Buf: PROCESSOR_SATISFIED, PROCESSOR_STALLED etc.
            enum PE_response signal;
            bitsMAX data;
            bool STROBE;     // set to high when 'signal' is formed
            } response;
        } out;

    struct IcnPorts {
        ICN_Message out_msg;       // to the ICN
        int send_to_node;
        ICN_Message in_msg;        // from the ICN
        int node;
    } icn;

    // output queue for ICN messages
    class ICN_Queue icnqueue;

    // Lock Manager
    class LockManager lock_mng;

    // dsm registers
    bitsADDR local_dsm_start;
    bitsADDR local_dsm_end;

    // returns addr home node number
    int find_dsm_node(bitsADDR addr);

    // states this finite state machine can be found in
    enum State {
        INITIAL,                 // INITIAL -- not doing anything
        READ_WAIT_RESPONSE,      // wait for non local read response
        WRITE_WAIT_ACK,          // wait for non local write ack
        LOCK_WAIT,               // wait for local lock grant
        LOCK_WAIT_GNT,           // wait for non local lock grant
        UNLOCK_WAIT_ACK          // wait for non local unlock ack
        } state;

    // some internal vars
    int wait_state_counter;
    int get_lock_from_myself;
    int next_lock_holder;

    struct Stat {                // statistics:
        Stat()  { }
        void dump(int index);
        } stat;
```

```
  public:

    MCModul();
    void assign_index(int ind) {
        index = ind;
        local_dsm_start = index?dsmStart[index]:DSM_START;
        local_dsm_end = dsmEnd[index];
        }
    void Introduce();
    void cycle();
    bool isidle() {return (state == INITIAL);}
    void dump_state();
    void dump_stats() {stat.dump(index);}
    TOPOLOGY_FRIENDS
    };

//-----------------------------------------------------
#ifdef PRAGMA_IMPLEMENTATION_MCM

#include <iostream.h>
#include <stdio.h>

MCModul::MCModul() {
    index = 0;           // default, but will be changed
    state = INITIAL;     // PCC will be in the INITIAL state
    // clear input and output ports
    out.response.signal = PROCESSOR_NORESPONSE;
    out.response.STROBE = 0;
    };

int MCModul::find_dsm_node(bitsADDR addr) {
    if ( (addr>=DSM_START) && (addr<dsmEnd[0]) ) return 0;
    for(int i=1;i<MAX_CPU_COUNT;i++)
        if ( (addr>=dsmStart[i]) && (addr<dsmEnd[i]) ) return i;
    return -1;   // not shared in DSM address space
    }

void MCModul::Introduce() {
    COUT << "Protocol: DSM [" << index << "] sequential MCM controller. ";
    COUT << "\n";
    }

void MCModul::Stat::dump(int index) {
//    COUT << "DSM"<<index<<" " << "\n";
    }

// the main method: Topology calls it in each simulated cycle. If it was
// not in initial state, it will resume from the point where it left off in
// the previous cycle.
void MCModul::cycle() {
    ICN_Message tmpmsg;
    int tmpdest;
    bool in_msg_exists;

    // get msg if exist
    if(in_msg_exists=Topology::icn_receive_msg(index,icn.node,icn.in_msg))
        switch(icn.in_msg.type) {
            case READ_REQUEST:
                icn.out_msg.type=READ_RESPONSE;
                icn.out_msg.size=icn.in_msg.size;
                fastcpy(&icn.out_msg.data,
```

```
                    (void *)icn.in_msg.data,
                    icn.in_msg.size);
            icnqueue.put(icn.node,icn.out_msg);
            break;

         case WRITE_REQUEST:
            fastcpy((void *)icn.in_msg.addr,
                    &icn.in_msg.data,
                    icn.in_msg.size);
            icn.out_msg.type=WRITE_ACK;
            icnqueue.put(icn.node,icn.out_msg);
            break;

        case LOCK_REQUEST:
            if(lock_mng.lock(icn.in_msg.data,icn.node)){
                icn.out_msg.type=LOCK_GRANTED;
                icnqueue.put(icn.node,icn.out_msg);
                }
            break;

        case UNLOCK_REQUEST:
            next_lock_holder=lock_mng.unlock(icn.in_msg.data);
            icn.out_msg.type=UNLOCK_ACK;
            icnqueue.put(icn.node,icn.out_msg);
            if(next_lock_holder==index)
                get_lock_from_myself=1;
            else if(next_lock_holder!=-1){
                icn.out_msg.type=LOCK_GRANTED;
                icnqueue.put(next_lock_holder,icn.out_msg);
                }
            break;
        case READ_RESPONSE:
        case WRITE_ACK:
        case LOCK_GRANTED:
        case UNLOCK_ACK:
            break;
        default:
            break;
    }

// try to send one msg if buffer is not empty
if(!icnqueue.isempty()) {
    icnqueue.peek(tmpdest,tmpmsg);
    if(Topology::icn_send_msg(index,tmpdest,tmpmsg)) icnqueue.get();
    }

switch (state) {
    case INITIAL                : goto L_STATE_INITIAL;
    case READ_WAIT_RESPONSE     : goto L_STATE_READ_WAIT_RESPONSE;
    case WRITE_WAIT_ACK         : goto L_STATE_WRITE_WAIT_ACK;
    case LOCK_WAIT              : goto L_STATE_LOCK_WAIT;
    case LOCK_WAIT_GNT          : goto L_STATE_LOCK_WAIT_GNT;
    case UNLOCK_WAIT_ACK        : goto L_STATE_UNLOCK_WAIT_ACK;
    }
```

```
L_STATE_INITIAL:
    if (! Topology::mcmodul_read_request(index))
        return;                      // return if there was none
    switch (in.request.type) {       // select where to go
        case REQUEST_READ   : goto L_READ;
        case REQUEST_LOCK   : goto L_LOCK;
        case REQUEST_UNLOCK : goto L_UNLOCK;
        case REQUEST_WRITE  : goto L_WRITE;
        default : break;
        }
    return;

// ------------------------ READ --------------------------
L_READ:
    if (in.request.isshared) {
        if ( (in.request.virtual_addr>=local_dsm_start)
             && (in.request.virtual_addr<local_dsm_end) ) {

        // local memory access:
            fastcpy(&out.response.data,
                    (void *)in.request.virtual_addr,
                    in.request.size);
            out.response.signal=PROCESSOR_SATISFIED;
            out.response.STROBE=1;
            state=INITIAL;
            return;
            }
        else {

        // remote memory access
            icn.out_msg.type = READ_REQUEST;
            icn.out_msg.data = in.request.virtual_addr;
            icn.out_msg.size = in.request.size;
            icn.send_to_node = find_dsm_node(in.request.virtual_addr);
            icnqueue.put(icn.send_to_node,icn.out_msg);
            state = READ_WAIT_RESPONSE;
            return;
            }
        }
    else {
        // this is read from local memory - kernel maintains consistency
        out.response.signal=PROCESSOR_SATISFIED;
        out.response.STROBE=1;
        state=INITIAL;
        return;
        }

L_STATE_READ_WAIT_RESPONSE:
    if(in_msg_exists && (icn.in_msg.type==READ_RESPONSE)) {
        fastcpy(&out.response.data,
                &icn.in_msg.data,
                icn.in_msg.size);
        out.response.signal=PROCESSOR_SATISFIED;
        out.response.STROBE=1;
        state=INITIAL;
        }
    return;
```

```
// ------------------------ WRITE --------------------------
L_WRITE:
    if (in.request.isshared) {
        if ( (in.request.virtual_addr>=local_dsm_start)
             && (in.request.virtual_addr<local_dsm_end) ) {

            // local memory access:
            fastcpy((void *)in.request.virtual_addr,
                    &in.request.data,
                    in.request.size);
            out.response.signal=PROCESSOR_SATISFIED;
            out.response.STROBE=1;
            state=INITIAL;
            return;
            }
        else {

            // remote memory access
            icn.out_msg.type = WRITE_REQUEST;
            icn.out_msg.data = in.request.data;
            icn.out_msg.addr = in.request.virtual_addr;
            icn.out_msg.size = in.request.size;
            icn.send_to_node = find_dsm_node(in.request.virtual_addr);
            icnqueue.put(icn.send_to_node,icn.out_msg);
            state = WRITE_WAIT_ACK;
            return;
            }
        }
    else {
        // this is write to local memory - kernel maintains consistency
        out.response.signal=PROCESSOR_SATISFIED;
        out.response.STROBE=1;
        state=INITIAL;
        }
    return;

L_STATE_WRITE_WAIT_ACK:
    if(in_msg_exists && (icn.in_msg.type==WRITE_ACK)) {
        out.response.signal=PROCESSOR_SATISFIED;
        out.response.STROBE=1;
        state=INITIAL;
        }
    return;

// ------------------------ LOCK --------------------------
L_LOCK:
    if ( (in.request.virtual_addr>=local_dsm_start)
         && (in.request.virtual_addr<local_dsm_end) ) {
    // I am lock manager for this lock
        if (lock_mng.lock(in.request.virtual_addr,index)) {
            out.response.signal=PROCESSOR_SATISFIED;
            out.response.STROBE=1;
            state=INITIAL;
            return;
            }
        else {
            get_lock_from_myself = 0;
            state=LOCK_WAIT;
            return;
            }
        }
```

```
    else {
    // Must ask lock from lock manager
        icn.out_msg.type = LOCK_REQUEST;
        icn.out_msg.data = in.request.virtual_addr;
        icn.send_to_node = find_dsm_node(in.request.virtual_addr);
        icnqueue.put(icn.send_to_node,icn.out_msg);
        state = LOCK_WAIT_GNT;
        return;
        }

L_STATE_LOCK_WAIT:
    if(get_lock_from_myself){
        out.response.signal=PROCESSOR_SATISFIED;
        out.response.STROBE=1;
        state=INITIAL;
        return;
        }
    return;

L_STATE_LOCK_WAIT_GNT:
    if(in_msg_exists && (icn.in_msg.type==LOCK_GRANTED)) {
        out.response.signal=PROCESSOR_SATISFIED;
        out.response.STROBE=1;
        state=INITIAL;
        return;
        }
    return;

// ------------------------ UNLOCK --------------------------
L_UNLOCK:
    if ( (in.request.virtual_addr>=local_dsm_start)
         && (in.request.virtual_addr<local_dsm_end) ) {
    // I am lock manager for this lock
        next_lock_holder=lock_mng.unlock(in.request.virtual_addr);
        if(next_lock_holder!=-1){
            icn.out_msg.type = LOCK_GRANTED;
            icnqueue.put(next_lock_holder,icn.out_msg);
            }
        out.response.signal=PROCESSOR_SATISFIED;
        out.response.STROBE=1;
        state=INITIAL;
        return;
        }
    else {
    // Must ask unlock from lock manager
        icn.out_msg.type = UNLOCK_REQUEST;
        icn.out_msg.data = in.request.virtual_addr;
        icn.send_to_node = find_dsm_node(in.request.virtual_addr);
        icnqueue.put(icn.send_to_node,icn.out_msg);
        state = UNLOCK_WAIT_ACK;
        return;
        }
L_STATE_UNLOCK_WAIT_ACK:
    if(in_msg_exists && (icn.in_msg.type==UNLOCK_ACK)) {
        out.response.signal=PROCESSOR_SATISFIED;
        out.response.STROBE=1;
        state=INITIAL;
        }
    return;
// ------------------------------------------------------------
    }
```

```
// debugging...
void MCModul::dump_state() {
    COUT << "MCM" << index << ": State: ";
    COUT << "\t\t[";
    if (state != INITIAL) {
        switch (in.request.type) {
            case REQUEST_READ  : COUT << "READ " << in.request.addr; break;
            case REQUEST_WRITE : COUT << "WRIT " << in.request.addr; break;
            case REQUEST_LOCK  : COUT << "LOCK " << in.request.addr; break;
            case REQUEST_UNLOCK: COUT << "UNLK " << in.request.addr; break;
            default : break;
            }
        }
    COUT << "]\n";
    }
#endif PRAGMA_IMPLEMENTATION_MCM
#endif _MCM_H_
```

topology.h

```
#ifndef _TOPOLOGY_H_
#define _TOPOLOGY_H_
/*
 *    DSM LIMES - Topology class
 *    Author: Zoran Dimitrijevic
 *    zorand@galeb.etf.bg.ac.yu
 */
#include "dsm.h"

// declares the types of modules in the system
class MCM_CLASS;
class ICN_CLASS;
class IO_BUF_CLASS;

class Base_Topology {
  protected:        // each module is instantiated through a pointer to it
    static IO_BUF_CLASS             * io_buf [MAX_CPU_COUNT];
    static MCM_CLASS                * mcmodul [MAX_CPU_COUNT];
    static ICN_CLASS                * icn;
  public:
    // services for individual types of modules
    static bool io_buf_input(int index);
    static bool mcmodul_read_request(int index);

    // icn servicies
    static bool icn_send_msg(int src, int dest, ICN_Message msg);
    static bool icn_receive_msg(int dest, int &src, ICN_Message &msg);

    // system services. These exist for each simulator.
    static void system_init();
    static void system_cleanup(Thread_Time last_time);
    static bool system_isidle();
    static void system_dump_state();
    static void system_collect_requests();
    static void system_cycle();
    };
typedef TOPOLOGY_CLASS Topology;
#endif // _TOPOLOGY_H_
```

topology.cc

```
/*
 *    DSM LIMES - Topology methods
 *    Author: Zoran Dimitrijevic
 *    zorand@galeb.etf.bg.ac.yu
 *
 */

#include <stdio.h>
#include "topology.h"

#include MCM_HDR        // include header files with definitions
#include ICN_HDR        // of the classes that represent
#include IO_BUF_HDR     // the hardware modules

// these are the pointers for module instances
IO_BUF_CLASS            * Base_Topology::io_buf[MAX_CPU_COUNT];
ICN_CLASS               * Base_Topology::icn;
MCM_CLASS               * Base_Topology::mcmodul[MAX_CPU_COUNT];

// these routines are called from the memsim

void Base_Topology::system_init() {
    icn = new ICN_CLASS;
    for(int i=0; i<MAX_CPU_COUNT; i++)  {
        io_buf[i] = new IO_Buf;
        io_buf[i]->assign_index(i);
        mcmodul[i] = new MCM_CLASS;
        mcmodul[i]->assign_index(i);
        }
    }

void Base_Topology::system_cleanup(Thread_Time last_time) {
    PRINTF("TOTAL simulation time: %d cycles.\n", last_time);
    for(int i=0; i<MAX_CPU_COUNT; ++i)
        mcmodul[i]->dump_stats();     // should be doing DELETE in
addition...
    icn->dump_stats(last_time);
    }

// returns true if all the modules in the system are idle
bool Base_Topology::system_isidle() {
    if (!icn->isidle()) return false;
    for (int i=0; i<CPU_COUNT; ++i) {
        if (!io_buf[i]->isidle()) return false;
        if (!mcmodul[i]->isidle()) return false;
        }
    return true;
    }

void Base_Topology::system_dump_state() {
    for(int i=0; i<CPU_COUNT; ++i)
        mcmodul[i]->dump_state();
    icn->dump_state();
    }
```

```
// Called at times when a set of request *could* exist
void Base_Topology::system_collect_requests() {
    for(int i=0; i<CPU_COUNT; ++i)
        io_buf[i]->start_cycle();
    }

// The main function: polls all the modules to do their share in this
cycle.
void Base_Topology::system_cycle() {
    for(int i=0; i<CPU_COUNT; ++i)
        mcmodul[i]->cycle();
    icn->cycle();
    for(i=0; i<CPU_COUNT; ++i)
        io_buf[i]->end_cycle();
    }

// Topology for your own simulator will need all the methods listed
// above; only implementations of these method will differ.

// These methods are called from various modules; their names and
// implementations depend on the structure of your system.

// ******************* write in IO_buf's input port ******************

// each IO_Buf calles this method (passing its index) upon the end
// of each simulated cycle. This method checks if the cache controller
// placed immediately below the io_buf module in question has set its
// STROBE bit high. If so, the controller has an information to communicate
// to the io_buf module. The topology will read this information from the
// controller's output port, write it into the io_buf module's input port,
// and clear the STROBE bit.

bool Base_Topology::io_buf_input(int index) {
    if (mcmodul[index]->out.response.STROBE) {
        io_buf[index]->in.response.signal =
            mcmodul[index]->out.response.signal;
        io_buf[index]->in.response.data =
            mcmodul[index]->out.response.data;
        mcmodul[index]->out.response.STROBE = 0;  // clear
        return true;    // tell the io_buf that a response appeared
        }
    else
        return false;
    }

// ******************** MCM modules services *********************

// The similar procedure is followed for reading io_buf's output
// (request description) and writing it into MCM controller's
// input port. The STROBE bit is also used. However, the controller
// module will not call this method at every cycle, but only when
// it's idle. If it is already processing previous request (such as
// read that was a miss), it will not call this method and the STROBE
// bit will not be cleared. Only when the controller gets idle again
// (returns to its INITIAL state) it will call this method. By inspecting
// its STROBE bit, the IO_Buf can know that the controller has read the
// request.

bool Base_Topology::mcmodul_read_request(int index)   {
    if (io_buf[index]->out.request.STROBE) {
        mcmodul[index]->in.request =
```

```
            * io_buf[index]->out.request.request_ptr;
        // this is where virtual address translates to physical.
        // In this case, just physcial = virtual.
        mcmodul[index]->in.request.addr =
            (mcmodul[index]->in.request.virtual_addr);
        io_buf[index]->out.request.STROBE = 0;          // clear
        return true;    // yes, a request exists
        }
    else
        return false;   // no request this time
    }

// ********* Inter Connection Network-related Topology functions *********

bool Base_Topology::icn_send_msg(int src, int dest, ICN_Message msg) {
    if(!icn->io_buf[src].in.STROBE) {
        icn->io_buf[src].in.STROBE=1;
        icn->io_buf[src].in.dest=dest;
        icn->io_buf[src].in.msg=msg;
        return true;
        }
    else return false;
    }

bool Base_Topology::icn_receive_msg(int dest, int &src, ICN_Message &msg) {
    if(icn->io_buf[dest].out.STROBE) {
        src=icn->io_buf[dest].out.src;
        msg=icn->io_buf[dest].out.msg;
        icn->io_buf[dest].out.STROBE=0;
        return true;
        }
    else return false;
    }
```

memsim.cc

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Module: memsim.cc
 * Author: Davor Magdic, magdic00390d@buef31.etf.bg.ac.yu
 *
 * Changed for DSM LIMES
 * Author: Zoran Dimitrijevic, zorand@galeb.etf.bg.ac.yu
 *
 * The main (and generic) simulation class, followed by the C++ to C
 * interface to the simulation kernel.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

#include <iostream.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include "dsm.h"
#include TOPOLOGY_HDR

#ifdef DETACHED_MEMSIM
static int debug = 1;
#else
static int debug = 0;
#endif

class Memsim {
    Thread_Time last_simulation_time;
  public:
    void init() { Topology::system_init(); }
    void end()  { Topology::system_cleanup(last_simulation_time); }
    void simulate(Thread_Time time_before, Thread_Time time_now);
    void dump_state()
        { Topology::system_dump_state(); }
    void memsim_print_requests(Thread_Time time);
    void memsim_print_responses(Thread_Time time);
    } memsim;

/*
 * The main routine. Called by the kernel every time a new request or a set
 * of requests appear, and in successive cycles where all the processors
 * are stalled.
 */
void Memsim::simulate(Thread_Time time_before, Thread_Time time_now) {
    for(Thread_Time time=time_before; time<=time_now; ++time) {
        int i;

        // optimization: if the whole system is in initial time,
        // shift current time to present
        if (time < time_now && Topology::system_isidle()) time = time_now;

        // if time now is time_now, wherein there are some requests, for
        // that is the reason we have been called in the first place,
        // collect them. The only situation where before != now is when
        // in before-1 everyone was satisfied, and we had to finish a
        // write or something. From before to now-1 there are no requests.
        if (time == time_now) {
            if (debug)
                for(i=0; i<CPU_COUNT; ++i)
```

```
                    // don't print anything if there are no requests
                    if (REQUEST_EXISTS(i)) {
                        memsim_print_requests(time);
                        break;
                        }
            Topology::system_collect_requests();
            }
        if (debug) dump_state();
        Topology::system_cycle();
        if (debug) { memsim_print_responses(time); FFLUSH; }
        }
    last_simulation_time = time_now;
    }

/*
 * These two are used for debugging.
 */
void Memsim::memsim_print_requests(Thread_Time time) {
    struct Memory_Request *req;
    char line[100]; int i;
    int pos =0;
    for(i=0; i<78; ++i) line[i] = ' ';
    sprintf(line, "==>  [%3d]:", time); pos += 15;
    #define BASE (bitsADDR) DATA_SEGMENT_START
    for(i=0; i<CPU_COUNT; ++i) {
        REQUEST_FILL(i, req);
        if (REQUEST_EXISTS(i))
            sprintf(line+pos, "P%d:", i);
        pos +=4;

        if (REQUEST_EXISTS(i)) {
            REQUEST_FILL(i, req);

            switch (req->type) {
                case REQUEST_READ  : sprintf(line+pos, "R %u",
                    req->virtual_addr - (req->isshared ? BASE : 0));
                    break;
                case REQUEST_WRITE : sprintf(line+pos, "W %u",
                    req->virtual_addr - (req->isshared ? BASE : 0));
                    break;
                case REQUEST_LOCK  : sprintf(line+pos, "L %u",
                    req->virtual_addr - (req->isshared ? BASE : 0));
                    break;
                case REQUEST_UNLOCK: sprintf(line+pos, "U %u",
                    req->virtual_addr - (req->isshared ? BASE : 0));
                    break;
                default : break;
                }
            switch (req->size) {
                case 1 : line[pos+1] = 'b'; break;
                case 2 : line[pos+1] = 'w'; break;
                default: break;
                }
            }
        else
        pos += 12;
        }
    for(i=0; i<78; ++i) if (!line[i]) line[i] = ' ';
    line[78] = 0;
    COUT << line << '\n';
    }
```

```
void Memsim::memsim_print_responses(Thread_Time time) {
    struct Memory_Request *req;
    char line[100];
    int pos =0;
    const DELC = '-';
    for(int i=0; i<78; ++i) line[i] = DELC;
    sprintf(line, "<==  [%3d]: ", time); pos += 15;
    for(i=0; i<CPU_COUNT; ++i) {
        REQUEST_FILL(i, req);
        switch (req->satisfied) {
            case PROCESSOR_SATISFIED:
                sprintf(line+pos, "P%d: OK ", i);
                break;
            case PROCESSOR_LOCK_BUSY:
                sprintf(line+pos, "P%d: LOCK_BUSY ",i); break;
            case PROCESSOR_STALLED:
                sprintf(line+pos, "P%d: STALLED ",i); break;
            default : break;
            }
        pos += 16;
        }
    for(i=0; i<78; ++i) if (!line[i]) line[i] = DELC;
    line[78] = 0;
    COUT << line << "\n";
    }

/*********************************************************************/
// C++ to C interface. Guess it's common for all the memsim's.

enum Memsim_Memory_Strategy memsim_init() {
    memsim.init();
    return Memsim_Maintains_Memory_Consistency;
    }

void memsim_cleanup() {
    memsim.end();
    }

void memory_simulate(Thread_Time time_before, Thread_Time time_now) {
    memsim.simulate(time_before, time_now);
    }

void memsim_dump_state() {
    memsim.dump_state();
    }

Thread_Time initial_thread_time(Thread_Time parent_time, int thread_number)
{
    return parent_time;
    }

void memsim_fork(int parent_ID, int child_ID) {
    }

void memsim_debug_toggle(int on) {
    debug = on;
    }
```

dir.cc

```
#define PRAGMA_IMPLEMENTATION_DIR
#include "dir.h"
```

dir.h

```
#ifndef _DIR_H_
#define _DIR_H_

/*
 *    DSM LIMES - Directory class
 *    Author: Zoran Dimitrijevic
 *    zorand@galeb.etf.bg.ac.yu
 */

#include "dsm.h"
#include TOPOLOGY_HDR
#include <values.h>                // just for MAXINT definition

class Dir {
    const DEFAULT_DIRSIZE=256;    // default Directory size in pages for
one node
    const DEFAULT_PAGESIZE=4096;// default pagesize in bytes
    int dirsize;
    int pagesize;
    bitsADDR pagemask;

  public:

  // line - should be modified for advanced MCM
    struct Line {
        bool valid;
        int access_cnt;
        bitsADDR page;
        bitsADDR bufaddr;
        bool modified;
        } *line;
    Dir();
    Dir(int size);
    bool ispresent(bitsADDR);
    int getlinenum(bitsADDR);
    int put(bitsADDR);
    bitsADDR get(bitsADDR adr, int n=-1);
    int find_page_for_replacement();
    void del(int);
    };

#ifdef PRAGMA_IMPLEMENTATION_DIR
//--------------------------------

#include <iostream.h>

Dir::Dir() {
    pagesize=DEFAULT_PAGESIZE;
    pagemask=~((bitsADDR)pagesize-1);
    dirsize=DEFAULT_DIRSIZE;
    line = new Line[dirsize];
    for (int i=0;i<dirsize;i++) {
        line[i].valid=0;
```

```
        line[i].bufaddr=0;
        }
    }

Dir::Dir(int size) {
    pagesize=DEFAULT_PAGESIZE;
    pagemask=~((bitsADDR)pagesize-1);
    dirsize=size;
    line = new Line[dirsize];
    for (int i=0;i<dirsize;i++) {
        line[i].valid=0;
        line[i].bufaddr=0;
        }
    }

// returns true if adr is in buffer on the node (page is valid in dir)
bool Dir::ispresent(bitsADDR adr) {
    for(int i=0;i<dirsize;i++)
        if(line[i].valid && (line[i].page==(adr&pagemask)) ) return 1;
    return 0;
    }

// returns -1 if page with adr is not in dir, and linenum if it is
int Dir::getlinenum(bitsADDR adr) {
    for(int i=0;i<dirsize;i++)
        if(line[i].valid && (line[i].page==(adr&pagemask)) ) return i;
    return -1;
    }

// put is caaled if page is not in dir to create new line
// returns -1 if dir is full
int Dir::put(bitsADDR adr) {
    for(int i=0;i<dirsize;i++)
        if(!line[i].valid) {
            line[i].page=adr&pagemask;
            line[i].valid=1;
            line[i].access_cnt=0;
            line[i].bufaddr=(bitsADDR) malloc(pagesize);
            return i;
            }
    return -1;
    }

// returns address in local buffer memory for adr
// if n is not specified get finds the page if valid in dir or returns 0 if
not
bitsADDR Dir::get(bitsADDR adr, int n=-1) {
    int i;
    bitsADDR bufaddr;
    if(n==-1){
        for(i=0;i<dirsize;i++)
            if(line[i].valid && (line[i].page==(adr&pagemask)) ) n=i;
        if(n==-1) return 0;
        }
    bufaddr=line[n].bufaddr+adr&(~pagemask);
    return bufaddr;
    }
```

```
// returns linenum for replacement
int Dir::find_page_for_replacement() {
    int i;
    int minaccess=MAXINT;
    int page=-1;

    for (i=0;i<MAXINT;i++) if(line[i].access_cnt<MAXINT) page=i;
    return page;
    }

// deletes linenum n from dir
// n is returned value from Dir::find_page_for_replacement()
void Dir::del(int n) {
    line[n].valid=0;
    line[n].access_cnt=0;
    line[n].modified=0;
    free((void *)line[n].bufaddr);
    }

#endif // PRAGMA_IMPLEMENTATION_DIR_
#endif //_DIR_H_
```