ДОДАТОК А
Список публікацій здобувача і відомості про апробацію результатів дисертації

А.1 Список публікацій здобувача

Наукові публікації, в яких опубліковані основні наукові результати дисертації

1. Гольдінер Д. І. Застосування мови програмування GO для моделювання процесів масового обслуговування. *Сучасний стан наукових досліджень та технологій в промисловості*. 2024. № 2(28). С. 65–75. DOI: 10.30837/2522-9818.2024.2.065 [Входить до міжнародної наукометричної бази Google Scholar.]

2. Гольдінер Д. І. Розробка архітектури програмного забезпечення для моделювання систем масового обслуговування під імплементацію мовою програмування GO. *Вісник Національного технічного університету «ХПІ». Серія: Системний аналіз, управління та інформаційні технології*. 2024. № 1 (11). С. 85–90. DOI: 10.20998/2079-0023.2024.01.14 [Входить до міжнародної наукометричної бази Google Scholar.]

3. Гольдінер Д. І., Матвієнко О. І. Зменшення ймовірності відмови в системах масового обслуговування з обмеженою чергою із застосуванням пріоритезації за розміром та штучного інтелекту. *Біоніка інтелекту*. 2024. № 1 (100). С. 36–42. DOI: 10.30837/bi.2024.1(100).05 [Входить до міжнародної наукометричної бази Google Scholar.]

Наукові праці, які засвідчують апробацію матеріалів дисертації

4. Goldiner D., Tevyashev A. System Analysis of the Parallel Execution Problem. *Інформаційні системи та технології – ICT-2019* : матеріали 8-ї Міжнародної науково-технічної конференції, Коблеве-Харків, Україна, 9-14 вересня

2019 р. Харків : ХНУРЕ, 2019. С. 210–213. [Входить до міжнародної наукометричної бази Google Scholar.]

5. Goldiner D. Rejection probability reduction in queueing systems with limited queue using size-based prioritization. *Perspectives of Contemporary Science: Theory and Practic*e : Proceedings of VII International Scientific and Practical Conference, Lviv, Ukraine, 19-21 August 2024. Lviv, Ukraine. 2024. P. 256–262.

А.2 Відомості про апробацію результатів дисертації

Таблиця А.1 – Відомості про апробацію результатів дисертації

№	Назва конференції	Місце проведення	Дати проведення	Форма участі
1.	Восьма Міжнародна науково-технічна конференція «Інформаційні системи та технології – ІСТ-2019»	Україна, с. Коблеве – м. Харків Харківський національний університет радіоелектроніки	9-14 вересня 2019 р.	очна
2.	VII International Scientific and Practical Conference «Perspectives of Contemporary Science: Theory and Practice»	Україна, м. Львів	19-21 серпня 2024 р.	очна (дистанційна)

ДОДАТОК Б

Акт впровадження

ЗАТВЕРДЖУЮ

Перший проректор
Харківського національного
університету радіоелектроніки

_____ І.В. Рубан

«16» 10 2024 р.

АКТ

про використання результатів дисертаційної роботи на здобуття наукового ступеня доктора філософії Гольдінера Дениса Ігоровича в освітньому процесі Харківського національного університету радіоелектроніки.

Комісія у складі начальника навчального відділу ХНУРЕ Міхнової А.В., завідувача кафедри прикладної математики Сидорова М.В, доцента кафедри прикладної математики Гибкіної Н.В. виконали перевірку та склали цей акт про те, що результати дисертаційної роботи Гольдінера Дениса Ігоровича впроваджені в освітній процес на кафедрі прикладної математики.

Комісія визначає, що результати дисертаційної роботи використано при викладанні дисципліни «Програмування» для здобувачів першого (бакалаврського) рівня вищої освіти за спеціальністю 124 Системний аналіз, у курсовому проєктуванні з цієї дисципліни та при підготовці кваліфікаційних робіт здобувачами першого (бакалаврського) та другого (магістерського) рівнів вищої освіти.

Начальник
навчального відділу Аліна МІХНОВА

Завідувач кафедри
прикладної математики Максим СИДОРОВ

Доцент кафедри
прикладної математики Надія ГИБКІНА

ДОДАТОК В

Програмна реалізація

В.1 Реалізація пакета взаємодіючого співвиконання

pkg/merec/models.go

```
package merec

import (
    "context"
    "fmt"
)

// CallOption is a proxy extension option that can do additional
actions during the processing.
type CallOption[In, Out any] interface {
    WithOption(next Call[In, Out]) Call[In, Out]
}

// Call is the function to be executed.
type Call[In, Out any] func(context.Context, In) (Out, error)

// Result is the Call execution result. If the error happened, the
value should be empty.
type Result[Out any] struct {
    value Out
    err   error
}

// String implements io.Stringer interface.
func (r Result[Out]) String() string {
    return fmt.Sprintf("value: %v, err: %v", r.value, r.err)
}

// Value returns the execution result value.
func (r Result[Out]) Value() Out {
    return r.value
}

// Err returns the execution error.
func (r Result[Out]) Err() error {
    return r.err
}

// ValueResult creates a new success result with a set value.
func ValueResult[Out any](value Out) Result[Out] {
    return Result[Out]{value: value}
```

```
}

// ErrorResult creates a new failure result with a set error.
func ErrorResult[Out any](err error) Result[Out] {
    return Result[Out]{err: err}
}
```

pkg/merec/options.go

```
package merec

import (
    "context"
    "fmt"
    "time"
)

type timeoutOption[In, Out any] struct {
    timeout time.Duration
}

// NewTimeoutOption is a constructor for the timeoutOption.
func NewTimeoutOption[In, Out any](timeout time.Duration)
CallOption[In, Out] {
    return timeoutOption[In, Out]{timeout: timeout}
}

// WithOption implements the CallOption interface for the
timeoutOption.
func (to timeoutOption[In, Out]) WithOption(next Call[In, Out])
Call[In, Out] {
    return func(ctx context.Context, in In) (Out, error) {
        ctx, ctxCsl := context.WithTimeout(ctx, to.timeout)
        defer ctxCsl()

        return next(ctx, in)
    }
}

type failFastOption[In, Out any] struct {
    mistakesLimit int
}

// NewFailFastOptionOption is a constructor for the
failFastOption.
func NewFailFastOptionOption[In, Out any](mistakesLimit int)
CallOption[In, Out] {
    return failFastOption[In, Out]{mistakesLimit: mistakesLimit}
}

// WithOption implements the CallOption interface for the
failFastOption.
func (failFastOption[In, Out]) WithOption(next Call[In, Out])
```

```
Call[In, Out] {
    return func(ctx context.Context, in In) (Out, error) {
        out, err := next(ctx, in)
        if err != nil {
            return *new(Out), fmt.Errorf("%w: %w", ErrMustStop, err)
        }

        return out, nil
    }
}
```

pkg/merec/errors.go

```
package merec

import "errors"

// The list of supported errors.
var (
    ErrBusinessLogic = errors.New("business logic execution
failed")
    ErrCtxCancel     = errors.New("root context was canceled")
    ErrCtxDeadline   = errors.New("root context's deadline
passed")
    ErrNilContext    = errors.New("context must be initiated")
    ErrNilInChan     = errors.New("input channel must be
initiated")
    ErrNilCallFunc   = errors.New("call function must be
initiated")
    ErrMustStop      = errors.New("the processing must be
interrupted")
)
```

pkg/merec/run_from_chan_pool.go

```
package merec

import (
    "context"
    "errors"
    "fmt"
    "sync"
)

// RunWorkerPool starts the pool of goroutine workers to consume
from the input channel and execute
// the call functions with inputs. Returns the channel to be
listened to, to get the Result values.
// Executions are independent, and it doesn't stop processing
inputs if call fails.
func RunWorkerPool[In, Out any](
    ctx context.Context,
    inCh <-chan In,
```

```
    call Call[In, Out],
    poolSize int,
    options ...CallOption[In, Out],
) (<-chan Result[Out], error) {
    next := call

    for _, o := range options {
       next = o.WithOption(next)
    }

    return runWorkerPool(ctx, inCh, next, poolSize)
}

func runWorkerPool[In, Out any](
    ctx context.Context,
    inCh <-chan In,
    call Call[In, Out],
    poolSize int,
) (<-chan Result[Out], error) {
    if err := validateRunFromChanInputs(ctx, inCh, call); err !=
nil {
       return nil, err
    }

    ctx, ctxCsl := context.WithCancel(ctx)

    resChanPool := SpawnResChanPool[Result[Out]](poolSize, 0)

    worker := func(resCh chan Result[Out]) {
       defer close(resCh)

       for in := range inCh {
          res, err := call(ctx, in)
          if errors.Is(err, ErrMustStop) {
             resCh <- ErrorResult[Out](fmt.Errorf("%w: %w",
ErrBusinessLogic, err))

             ctxCsl()

             return
          }

          if err != nil {
             resCh <- ErrorResult[Out](fmt.Errorf("%w: %w",
ErrBusinessLogic, err))
             continue
          }

          resCh <- ValueResult[Out](res)
       }
    }

    for i := 0; i < poolSize; i++ {
```

```
        go worker(resChanPool[i])
    }

    return MergeChanPool(resChanPool), nil
}

// SpawnResChanPool creates as many channels as it is needed.
// Implements concurrency pattern FanOut.
func SpawnResChanPool[T any](poolSize int, bufSize int) []chan T {
    chanPool := make([]chan T, poolSize)

    for i := range chanPool {
        chanPool[i] = make(chan T, bufSize)
    }

    return chanPool
}

// MergeChanPool combines the output from lit of channels into a
single one.
// Implements concurrency pattern FanIn.
func MergeChanPool[T any](resChanPool []chan T) chan T {
    mergeCh := make(chan T, len(resChanPool))

    var wg sync.WaitGroup

    wg.Add(len(resChanPool))

    for _, rCh := range resChanPool {
        go func(resCh chan T) {
            for out := range resCh {
                mergeCh <- out
            }

            wg.Done()
        }(rCh)
    }

    go func() {
        wg.Wait()
        close(mergeCh)
    }()

    return mergeCh
}

// MergeSignalChanPool combines the output from lit of channels
into a single one.
// Implements concurrency pattern FanIn. In addition, it takes the
value from the chan only
// if there is a corresponding signal.
func MergeSignalChanPool[T any](inChanPool []chan T,
signalChanPool []chan struct{}) (<-chan struct{}, chan T) {
```

```
    done := make(chan struct{})
    mergeCh := make(chan T, len(inChanPool))

    var wg sync.WaitGroup

    wg.Add(len(inChanPool))

    for i, inCh := range inChanPool {
       go withSignal(&wg, inCh, signalChanPool[i], mergeCh)
    }

    go func() {
       wg.Wait()
       close(mergeCh)
       done <- struct{}{}
    }()

    return done, mergeCh
}

func withSignal[T any](
    wg *sync.WaitGroup,
    inCh <-chan T,
    signalCh <-chan struct{},
    mergeCh chan<- T,
) {
    for out := range inCh {
       <-signalCh
       mergeCh <- out
    }

    wg.Done()
}

// TrySend tries to send the value into the channel.
// If channel is blocked we do nothing and return.
func TrySend[T any](ch chan T, v T) {
    select {
    case ch <- v:
    default:
    }
}

// TryReedSignal tires to read the signal from the channel.
// If there are no values in the channel we do nothing and return.
func TryReedSignal(ch <-chan struct{}) bool {
    select {
    case <-ch:
       return true
    default:
       return false
    }
}
```

В.2 Реалізація пакета систем масового обслуговування

pkg/mss/errors.go

```
package mss

import "errors"

// The list of supported errors.
var (
    ErrEngineNotFound       = errors.New("engine not found")
    ErrGlobalQueueOverflow  = errors.New("global queue is full")
    ErrInvalidTask          = errors.New("the task does not satisfy
the system requirements")
)
```

pkg/mss/interfaces.go

```
package mss

import "github.com/DenisGoldiner/cboost/pkg/merec"

type sourcer[In any] interface {
    feed() <-chan In
}

type preprocessor[In, Out any] interface {
    inQueue(inCh <-chan In, postProcessCh chan<-
merec.Result[Out]) <-chan In
}

type postprocessor[Out any] interface {
    ok(Out)
    err(error)
}
```

pkg/mss/postprocessor_logger.go

```
package mss

import (
    "log/slog"
)

const (
    logKeyResult = "result"
    logKeyError  = "error"
)

// LogPostprocessor is an implementation of the postprocessor
interface.
// It only logs if the execution was successful or not.
```

```
type LogPostprocessor[Out any] struct {
    logger *slog.Logger
}

// NewLogPostprocessor is a constructor for the LogPostprocessor.
func NewLogPostprocessor[Out any](logger *slog.Logger)
LogPostprocessor[Out] {
    return LogPostprocessor[Out]{logger: logger}
}

func (p LogPostprocessor[Out]) ok(resVal Out) {
    p.logger.Info("Successfully processed task", logKeyResult,
resVal)
}

func (p LogPostprocessor[Out]) err(resErr error) {
    p.logger.Info("Failed to process task", logKeyError, resErr)
}
```

pkg/mss/preprocessor_limited_queue_reject.go

```
package mss

import (
    "github.com/DenisGoldiner/cboost/pkg/liberr"
    "github.com/DenisGoldiner/cboost/pkg/merec"
)

// LimitedQueueRejectPreprocessor is an implementation for the
preprocessor interface.
// It simulates the classic mass service system with a limited
queue and rejects in case of overfill.
type LimitedQueueRejectPreprocessor[In, Out any] struct{}

// NewLimitedQueueRejectPreprocessor is a constructor for the
LimitedQueueRejectPreprocessor.
func NewLimitedQueueRejectPreprocessor[In, Out any]()
LimitedQueueRejectPreprocessor[In, Out] {
    return LimitedQueueRejectPreprocessor[In, Out]{}
}

// LimitedQueueRejectEngine simulates limited queue and rejects
new task if the waiting line is full.
func (LimitedQueueRejectPreprocessor[In, Out]) inQueue(
    inCh <-chan In,
    postProcessCh chan<- merec.Result[Out],
) <-chan In {
    globalQueueCh := make(chan In, cap(inCh))

    go func() {
       for in := range inCh {
          select {
          case globalQueueCh <- in:
```

```
            default:
                err := liberr.WrapMsg("failed to plan the task
execution", ErrGlobalQueueOverflow)
                postProcessCh <- merec.ErrorResult[Out](err)
            }
        }

        close(globalQueueCh)
    }()

    return globalQueueCh
}
```

pkg/mss/preprocessor_limited_sized_queue_reject.go

```
package mss

import (
    "fmt"
    "log/slog"
    "time"

    "github.com/DenisGoldiner/cboost/pkg/liberr"
    "github.com/DenisGoldiner/cboost/pkg/merec"
)

const (
    numSizes          = 3
    bufferSize        = 20
    signalsBufferSize = 100
)

// Sizer is an interface to be implemented by tasks that should be
grouped by size.
type Sizer interface {
    Size() int
}

// LimitedSizedQueueRejectPreprocessor is an implementation of the
preprocessor interface.
// It sorts input tasks by size and plans them according to
priorities.
type LimitedSizedQueueRejectPreprocessor[In, Out any] struct {
    logger        *slog.Logger
    incomeIntense int
}

// NewLimitedSizedQueueRejectPreprocessor is a constructor for the
LimitedSizedQueueRejectPreprocessor.
func NewLimitedSizedQueueRejectPreprocessor[In, Out any](
    logger *slog.Logger,
    incomeIntense int,
) LimitedSizedQueueRejectPreprocessor[In, Out] {
```

```
    return LimitedSizedQueueRejectPreprocessor[In, Out]{logger:
logger, incomeIntense:  incomeIntense}
}

func (p LimitedSizedQueueRejectPreprocessor[In, Out]) inQueue(
    inCh <-chan In,
    postProcessCh chan<- merec.Result[Out],
) <-chan In {
    sizedChanPool := p.sortTasks(inCh, postProcessCh)
    signalsChanPool :=
merec.SpawnResChanPool[struct{}](len(sizedChanPool),
signalsBufferSize)
    done, globalQueueCh :=
merec.MergeSignalChanPool(sizedChanPool, signalsChanPool)

    go p.prioritizeTasks(done, sizedChanPool, signalsChanPool)

    return globalQueueCh
}

func (p LimitedSizedQueueRejectPreprocessor[In, Out])
prioritizeTasks(
    done <-chan struct{},
    sizedChanPool []chan In,
    signalsChanPool []chan struct{},
) {
    ticker := time.NewTicker(time.Millisecond)
    defer ticker.Stop()

    for {
       select {
       case <-done:
          return
       case <-ticker.C:
          p.overflowChance(sizedChanPool, signalsChanPool)
          p.sizeBasedChance(signalsChanPool)
       }
    }
}

func (LimitedSizedQueueRejectPreprocessor[In, Out])
overflowChance(
    sizedChans []chan In,
    sizeSignals []chan struct{},
) {
    for i, sch := range sizedChans {
       if len(sch) < cap(sch) {
          continue
       }

       merec.TrySend(sizeSignals[i], struct{}{})
    }
}
```

```
func (LimitedSizedQueueRejectPreprocessor[In, Out])
sizeBasedChance(sizeSignals []chan struct{}) {
    for i := range sizeSignals {
       for j := 0; j < len(sizeSignals)-i; j++ {
          merec.TrySend(sizeSignals[i], struct{}{})
       }
    }
}

func (p LimitedSizedQueueRejectPreprocessor[In, Out]) sortTasks(
    inCh <-chan In,
    postProcessCh chan<- merec.Result[Out],
) []chan In {
    sizedChans := merec.SpawnResChanPool[In](numSizes, bufferSize)

    go func() {
       for in := range inCh {
          p.sortTask(sizedChans, postProcessCh, in)
       }

       for i := range sizedChans {
          close(sizedChans[i])
       }
    }()

    return sizedChans
}

func (LimitedSizedQueueRejectPreprocessor[In, Out]) sortTask(
    sizedChans []chan In,
    postProcessCh chan<- merec.Result[Out],
    in In,
) {
    inSizer, ok := any(in).(Sizer)
    if !ok {
       err := liberr.WrapMsg("the task does not support Size
detection", ErrInvalidTask)
       postProcessCh <- merec.ErrorResult[Out](err)

       return
    }

    size := inSizer.Size()

    if size == 0 || size-1 > len(sizedChans) {
       err := liberr.WrapMsg("the task Size is not supported",
ErrInvalidTask)
       postProcessCh <- merec.ErrorResult[Out](err)

       return
    }
```

```
    select {
    case sizedChans[size-1] <- in:
    default:
       msg := fmt.Sprintf("failed to plan the task Size %d
execution", size)
       err := liberr.WrapMsg(msg, ErrGlobalQueueOverflow)
       postProcessCh <- merec.ErrorResult[Out](err)
    }
}
```

pkg/mss/sourcer_chan.go

```
package mss

// ChanSourcer is a sourcer interface implementation that uses en
external channel as a source for input tasks.
type ChanSourcer[In any] struct {
    inputs chan In
}

// NewChanSourcer is a constructor for the ChanSourcer.
func NewChanSourcer[In any](inputs chan In) ChanSourcer[In] {
    return ChanSourcer[In]{inputs: inputs}
}

func (s ChanSourcer[In]) feed() <-chan In {
    return s.inputs
}
```

pkg/mss/sourcer_slice.go

```
package mss

// SliceSourcer is an implementation for the sourcer interface.
// It uses the external slice as a source for input tasks.
type SliceSourcer[In any] struct {
    inputs []In
}

// NewSliceSourcer is a constructor for the SliceSourcer.
func NewSliceSourcer[In any](inputs []In) SliceSourcer[In] {
    return SliceSourcer[In]{inputs: inputs}
}

func (s SliceSourcer[In]) feed() <-chan In {
    inputIntense := len(s.inputs)
    inCh := make(chan In, inputIntense)

    go func() {
       for _, in := range s.inputs {
          inCh <- in
       }
```

```
        defer close(inCh)
    }()

    return inCh
}
```

pkg/mss/system.go

```
package mss

import (
    "context"
    "log/slog"

    "github.com/DenisGoldiner/cboost/pkg/merec"
)

// System is a simulation of the mass service system.
type System[In, Out any] struct {
    source   sourcer[In]
    pre      preprocessor[In, Out]
    post     postprocessor[Out]
    poolSize int
    logger   *slog.Logger
}

// NewSystem initiates the mass service System with
func NewSystem[In, Out any](
    logger *slog.Logger,
    sourcer sourcer[In],
    preprocessor preprocessor[In, Out],
    postprocessor postprocessor[Out],
    poolSize int,
) System[In, Out] {
    return System[In, Out]{
        logger:   logger,
        source:   sourcer,
        pre:      preprocessor,
        post:     postprocessor,
        poolSize: poolSize,
    }
}

// Run starts the System simulation. The processing happens
asynchronously.
// The function returns the outputs chan at the beginning.
// If the client does not pop results from the outCh, it will
block the processing at some point.
func (s System[In, Out]) Run(
    ctx context.Context,
    call merec.Call[In, Out],
) <-chan merec.Result[Out] {
    inCh := s.source.feed()
```

```
    outCh := s.run(ctx, call, inCh)

    return outCh
}

// run is the heart of the task processing engine.
func (s System[In, Out]) run(
    ctx context.Context,
    ef merec.Call[In, Out],
    inCh <-chan In,
) <-chan merec.Result[Out] {
    // TODO: think about the size of the buffer for the
postProcessCh
    postProcessCh := make(chan merec.Result[Out], cap(inCh))
    outCh := s.postprocess(postProcessCh)

    go func() {
       globalQueueCh := s.pre.inQueue(inCh, postProcessCh)
       workerResCh, _ := merec.RunWorkerPool(ctx, globalQueueCh,
ef, s.poolSize)

       for res := range workerResCh {
          postProcessCh <- res
       }

       close(postProcessCh)
    }()

    return outCh
}

func (s System[In, Out]) postprocess(
    postProcessCh <-chan merec.Result[Out],
) <-chan merec.Result[Out] {
    outCh := make(chan merec.Result[Out], cap(postProcessCh))

    go func() {
       for res := range postProcessCh {
          if res.Err() != nil {
             s.post.err(res.Err())
             outCh <- res

             continue
          }

          s.post.ok(res.Value())
          outCh <- res
       }

       close(outCh)
    }()
```

```
    return outCh
}
```

pkg/liberr/errors.go

```
package liberr

import (
    "errors"
    "fmt"
)

const (
    wrapFormat    = "%w; %w"
    wrapMsgFormat = "%s; %w"
)

// WrapMsg wraps errors with the message. The pattern for errs is
FIFO.
// Every next error should be more specific than previous one.
func WrapMsg(msg string, errs ...error) error {
    if len(errs) == 0 {
       return errors.New(msg)
    }

    var resErr error

    for _, err := range errs {
       if err != nil {
          resErr = wrap(resErr, err)
       }
    }

    return fmt.Errorf(wrapMsgFormat, msg, resErr)
}

// WrapErr wraps errors. The pattern for errs is FIFO.
// Every next error should be more specific than previous one.
func WrapErr(errs ...error) error {
    if len(errs) == 0 {
       return nil
    }

    var resErr error

    for _, err := range errs {
       if err != nil {
          resErr = wrap(resErr, err)
       }
    }

    return resErr
}
```

```
func wrap(err1, err2 error) error {
    if err1 == nil && err2 == nil {
        return nil
    }

    if err1 == nil {
        return err2
    }

    if err2 == nil {
        return err1
    }

    return fmt.Errorf(wrapFormat, err1, err2)
}
```