Pavlo Bazilinskyy

Multi-core Insense

University of St Andrews

June 2013

ABSTRACT

This project set out to investigate the benefits of using private heaps for memory management and static thread placement for optimising performance and cache usage. For this study, Insense, which is a component-based programming language developed in the University of St Andrews that abstracts over complications of memory management, concurrency control and synchronisation, was used (Dearle et al. 2008). Two memory management schemes are under investigation: use of a single shared heap and use of multiple private heaps. Further, three thread placement schemes are designed and implemented: 1) even distribution among cores; 2) placing all components on a single core; 3) locating Insense components based on frequency of inter-component communication.

Furthermore, several elements of this investigation are worth emphasizing. With regard to allocation and deallocation of memory taking place in component instances running on different cores, the efficiency of using a private heap for each component resulted in speedup by a factor of 16. Then, utilising private heaps reduces a number of L1 cache misses by ~30%. Distributing components over cores according to communication pattern, for the most part performed similar to allowing the OS to perform thread placement dynamically according to load balance. In cases where no exchange of data between components takes place, static placement outperformed because there is no computation which may make load balancing dynamic placement of threads under control of the OS difficult. In this case, the static placement scheme was faster than dynamic balancing by a factor of 2.4.

DECLARATION

I declare that the material submitted for assessment is my own work except here credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 19,991 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

ACKNOWLEDGEMENTS

Words cannot describe my deep appreciation to my supervisor Jonathan Lewis, who was always there to listen to my ideas and who was patient enough to often describe the same concept to me for the 2nd, 3rd and sometimes even 4th time. He was always available whenever I needed him, and his positive attitude towards this project encouraged me to not give up, especially during the first half of my work, when I would very often have entered a vicious circle, in which I could not have achieved desired results for days or even weeks.

I would also like to express my gratitude to Professor Alan Dearle, who is one of the creators of Insense language. Me, Jonathan, and him spent countless hours discussing theoretical aspects of the project, which gave me numerous sleepless nights. He had faith in me and it helped me go forward and motivated me greatly. Also, thank you enormously for showing me power of taking notes with my iPad.

Special thanks, to my parents, my grandmother, and my whole family who if it were not for them, I would not be here submitting this project. Even though we have been separated by thousands of miles for now four years, I always had continuous support from their side, which proved to be more than enough to help me make it to the end, without regrets.

CONTENTS

1 INTRODUCTION 1

1.1 Dissertation structure 1

2 OBJECTIVES 2

3 CONTEXT SURVEY 3

3.1 Multi-core Systems 3

3.1.1 Memory Cache in Multi-core Systems 4

3.2 Overview of Insense Language 5

3.2.1 Components and Channels in Insense 7

3.2.2 Data Types 10

3.3 Memory Management in C Language 11

3.4 Utilising Private Heaps for Memory Management 12

3.5 Thread Affinity 14

4 SOFTWARE ENGINEERING PROCESS 14

4.1 Project Plan 15

5 REQUIREMENTS SPECIFICATION 16

5.1 Functional Requirements 16

5.2 Non-functional Requirements 17

6 ETHICS 17

7 DESIGN 17

7.1 Design of Multi-core Insense 18

7.1.1 Components 18

7.1.2 Channels 18

7.1.3 Program Entry Point 19

7.1.4 Memory Management & Garbage Collection 20

7.2 Data Placement Schemes 21

7.2.1 Shared Heap 21

7.2.2 Private Heaps 22

7.2.3 Data Placement and Potential Cache Misses 23

7.3 Thread Placement Schemes 27

7.3.1 Dynamic Placement 28

7.3.2 Static Placement 28

8 IMPLEMENTATION 32

8.1 Base Unix Implementation 33

8.1.1 Components 34

8.1.2 Arrays 37

8.1.3 Channels 38

8.1.4 Program Entry Point 43

8.1.5 Memory Management & Garbage Collection 44

8.1.6 Thread-safe List 46

8.1.7 Build System 47

8.2 Data Placement Schemes 49

8.2.1 Shared Heap 52

8.2.2 Private Heaps 53

8.2.2.1 Creation of a New Private Heap 54

8.2.2.2 Allocation and Deallocation of Memory within the Private Heaps55

8.3 Thread Placement Schemes 59

8.3.1 Dynamic Placement 59

8.3.2 Static Placement 60

8.3.2.1 Even Distribution of Threads Among Cores 62

8.3.2.2 Static Placement of Threads to a Single Core 63

8.3.2.3 Static Placement of Threads Based on Communication 63

9 EXPERIMENTS 63

9.1 Effects of Memory Management Schemes on Allocation and Deallocation of Memory 64

9.1.1 Design of Experiments 64

9.1.2 Program Used 65

9.1.3 Experimental Results 65

9.1.4 Discussion of Results 66

9.2 Effects of Memory Management Schemes on Cache Usage 68

9.2.1 Design of Experiments 68

9.2.2 Program Used 69

9.2.3 Experimental Results 69

9.2.4 Discussion of Results 70

9.3 Effects of Thread Placement and Communication Between Components on Performance 71

9.3.1 Design of Experiments 71

9.3.2 Programs Used 72
9.3.3 Experimental Results...73
9.3.4 Discussion of Results 77
10 CONCLUSIONS AND FURTHER WORK...78
11 BIBLIOGRAPHY..80

APPENDICES

APPENDIX 1: List of Figures...84
APPENDIX 2: Gantt Chart for the Project...89
APPENDIX 3: Instruction on Executing Insense Programs 90
APPENDIX 4: Creation of a New Component 90
APPENDIX 5: Deallocation of Memory for Private Heaps..92
APPENDIX 6: Test_alloc Program Used to Evaluate Effects of Using Multiple Heaps for (De-)allocation of Memory..93
APPENDIX 7: Test_cache Program Used to Evaluate Effects of Memory Management Schemes on Cache Usage 93
APPENDIX 8: Programs Used to Evaluate Effects of Thread Placement and Communication Between Components on Performance..94
Appendix 8.1: “Sender - Receiver” Scheme 94
Appendix 8.2: “Client - Server” Scheme 95
Appendix 8.3: “Dispatch - Worker” Scheme...97
Appendix 8.4: “No communication” Scheme 98
APPENDIX 9: Real, User, and System Time of Running Experiments 99
Appendix 9.1: Experiment with (De-)allocation of Memory, 1 Component 99
Appendix 9.2: Experiment with (De-)allocation of Memory, 4 components...100
Appendix 9.3: Results from “Sender - Receiver” example with 2 senders and 2 receivers..100
Appendix 9.4: Results from “Client - Server” example 101
Appendix 9.5: Results from “Dispatch - Worker” example 102
APPENDIX 10: Ethics approval form...104
APPENDIX 11: Output of Cachegrind for the test_cache program......................................109
Appendix 11.1: Shared Heap - Dynamic..109
Appendix 11.2: Private Heaps - Dynamic 109
Appendix 11.3: Shared Heap - 1 core...109
Appendix 11.4: Private Heaps - 1 core 110

Appendix 11.5: Shared Heap - Round-Robin 110

Appendix 11.6: Private Heaps - Round-Robin 110

1 INTRODUCTION

Insense is a component-based programming language developed in the University of St Andrews that abstracts over complications of memory management, concurrency control and synchronisation, and separates application software from the hardware and the operating system (OS). Insense applications are constructed as collections of software components that communicate with each other by using typed, directional, synchronous channels. A component may contain updatable locations that may only be accessed by code defined in the component. Each component is a unit of concurrent computation. The language is mainly used in wireless sensor networks (WSN). However, researchers behind Insense have been speculating about porting it to multi-core systems. (Dearle et al. 2008)

Many multi-core operating systems provide mechanisms for load-balancing thread placement on cores. Dynamic memory is commonly allocated on a single large heap shared by all threads and processes running on the OS. However, such approaches may not always be the best ones for a given programming language on multi-core systems.

This project focuses on investigating benefits of utilisation of private (small) heaps for memory management and static thread placement for optimising efficiency and performance of applications running on multi-core systems. The base implementation of the project involves porting the language to the Unix-based Scientific Linux operating system. Three main objectives were set for this project.

1.1 Dissertation structure

The remainder of this dissertation is structured as follows: Chapter 2 outlines objectives that were set for this project. Chapter 3 surveys the context applicable to this project and includes a brief overview of relevant multi-core systems and the Insense programming language. Chapters 4 and 5 give a brief description of software engineering principles applied in the project and list functional and non-functional requirements that were set. Chapters 6 deals with ethical concerns. Chapter 7 summarises design process that was undertaken in the project and lists design decisions that were taken. Then, Chapter 8 first describes how Insense was adjusted to run on Unix-based systems that operate with multiple cores, then it gives a description of the main work performed on implementing schemes for memory management and thread placement described in Chapter 7. Chapter 9 introduces experiments that were performed to test implemented memory management and thread placement schemes and describes achieved results. Lastly, Chapter 10 sums up what was accomplished and lists suggestions for future work.

2 OBJECTIVES

The main objectives of this project are to investigate efficient memory management and thread placement schemes for Insense on multi-core systems. To this end, a new Insense implementation for multi-core, Unix systems will be designed, implemented, and tested. This is an arduous undertaking in its own right because it involves creating a new version of the Insense language runtime (written in C) and modifying the compiler as well (written in Java). Some time must also be spent to understand the language, current runtime, and the InceOS and Contiki OS for which there is limited documentation.

Experimentation will then be used to establish which schemes are more efficient. Two memory management schemes are under investigation: use of a single shared heap and utilisation of multiple private heaps. When a single heap is used, it is anticipated that inefficiencies in cache and concurrent access to the shared heap from multiple threads may impact on performance. The research aims to investigate such effects and the effect of using multiple private heaps that are allocated to component instances or cores.

Further, three thread placement schemes are under investigation. One is based on static placement, where Insense components are distributed evenly among cores. Then, the scheme where all components are placed on a single core. The third scheme involves placing Insense components on cores based on certain properties that they have (e.g. frequency of communicating with other components). Results retrieved from experimenting with these three schemes will be compared to default behaviour of Unix-based systems where load balancing is performed via live migration of threads to different cores. Differences in performance of these approaches will be investigated.

Moreover, strong encapsulation of components in Insense applications may be advantageous for multi-core systems. The fact that Insense components are strongly encapsulated implies that they cannot share memory references to the component state. Information[1] can only be exchanged between components using channel abstraction. This feature of Insense can be tested by performing comparative analysis of solutions to a simple problem executed in Insense and a control language, for example, C, where direct access to shared memory from multiple threads can be exploited in the parallel algorithm. Retrieving this experimental data was set as the third, optional, goal of the project.

3 CONTEXT SURVEY

3.1 Multi-core Systems

The first member of the x86 mircroprocessor family was created as early as in 1978 when the Altair 8086 processor was designed (Gove 2011). Since then the world has seen a number of improvements in performance of central processing units (CPUs). The most notable improvement has been gain in speed of processors have come from increasing the clock speed (frequency at which a processor is running). The 8086 processor was functioning at around 5 MHz, today readings of clock speed can go as high as 5.5GHz (Halfacree 2012).

Furthermore, improvements in performance of processors were achieved by exploiting instruction-level parallelism - simultaneously performing multiple operations in a computer program (Hennessy 2007). Processors that use instruction-level parallelism have the ability to issue numerous instructions concurrently. In their pipelines, instructions are pre-fetched, split into sub-components and executed out-of-order (Schauer 2008). The Pentium IV CPU released in 2000 was one of the last and the most powerful single-core processors ("*Intel Introduces*" 2000). The "Prescott" and "Cedar Mill" cores from Pentium IV family featured as as many as 31 stages in their pipelines, the longest in the history of mainstream computing (Schmid 2004). However, there are certain factors that limit efficiency of systems that rely on this approach. Achieving satisfying levels of instruction-level parallelism depends on efficiency of branch prediction performed by hardware or software. It is not trivial, which was proved as early as in 1991 (Wall 1991). Additionally, cache miss penalty to main memory which costs hundreds of CPU cycles and complexity of hardware that needs to be built often reduce benefits that can be achieved from implementing instruction-level parallelism (McKee 2004).

[1] Information is used in the context of "data" in this document, unless stated otherwise.

More recent advancements in development of hardware for performing computations have mostly been emphasising importance on increasing the number of cores that reside on the chip, rather than experimenting with changing the clock rate or improving methods behind instruction-level parallelism. As a result a new type of systems powered by a single processor that incorporates more that one central processing unit (or “cores”) was developed (Rouse 2007). These cores are responsible for reading and executing instructions given to the CPU by programs. Adding additional cores onto the silicon base to improve performance increases the upper bound of amount of work that can be done by the processor by a factor of the total number of cores that the CPU obtains (Gove 2011). The motivation behind switching to multi-core systems resides in the fact that improving serial performance (performance of CPUs with one core) has become increasingly hard.

3.1.1 Memory Cache in Multi-core Systems

Multi-core systems also came with new memory cache approaches. As an example, IBM, does not use separate caches in its multicore server chips at all. On the other hand, Figure 3.1 outlines a more typical cache hierarchy model, used in, for example, Intel’s Itanium and AMD’s Opteron chips for servers and workstations, where each CPU core has its own private Level 1 (L1) and Level 2 (L2) data and instruction caches. The Level 3 (L3) unified cache is normally shared between all cores. Also, all core's L1 and L2 caches reside on the same die as the core and cannot be accessed by other cores. The cores are each connected to the L3 cache via the shared data bus. Allowing any processor to access any processors' cache memory is problematic, since there are no direct physical connections between different caches. If one core requires access to data that resides in another core's cache, the only path through which it can be achieved is the system bus. Assigning separate cache to each core removes the extra work required to design chips so that multiple cores can work with a single, centralized cache. (Geer 2005)

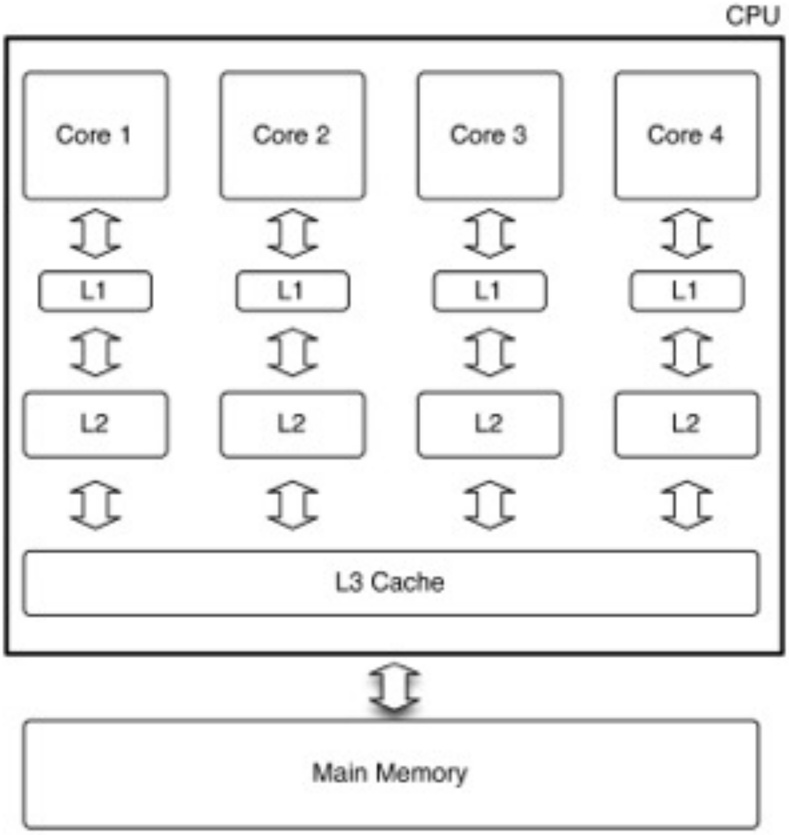


Figure 3.1 Three components connected by two channels.

As mentioned in Chapter 2 above, this project aims to investigate efficient memory management and thread placement schemes for Insense on multi-core systems. Insense is discussed in the following Section.

3.2 Overview of Insense Language

Insense is a high-level programming language developed at the University of St Andrews (Dearle et al. 2008). The initial goal behind creating the language was to decrease the complexity of developing Wireless Sensor Network (WSN) applications by abstracting over such concepts of programming as synchronisation, memory management and event-driven programming. The language was built to be run on two operating systems: Contiki (Lewis, Dearle 2011) and InceOS (Harvey et al. 2012).

This language relies on the idea of having components as basic computational units - building blocks - of applications. They serve as units of concurrent computation (Sharma et al. 2009). In Insense, the complexity is supposed to be borne by the language implementation rather than by the developer. Components may be composed in a fractal manor because components can create instances of other components (Dearle et al. 2008).

Moreover, Insense components are strongly encapsulated: they are stateful and the only medium of inter-component communication is usage of channels. To avoid unintended sharing of variables, an Insense component cannot reference any external data objects or locations. All inter-component communication passes through channels. Further, channels are typed and directional. All elements of the language including components and other channels can be shared through channels. Communication through them is synchronous: operations involving sending data are blocked until the messages are received and receive calls are blocked when there is nothing to be received. Figure 3.2 below shows two components `Sender` and `Receiver` that are connected by two channels; the `Sender` component sends data, the `Receiver` component receives it.

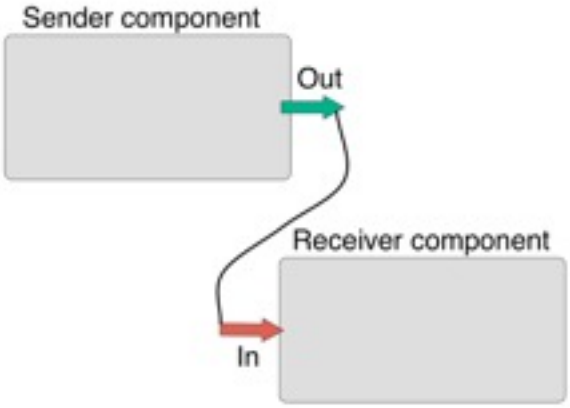


Figure 3.2 Topology with two components connected by a single channel.

More complex topologies can also be described by Insense components and channels. Figure 3.3 below features five components interconnected by five channels. The components presented in the diagram can be divided into two categories: *senders* who send data on channels and *receivers* that receive data from channels. In this case datum is sent by `Sender 1`, by `Sender 2` or by `Sender 3` will be received by either `Receiver 1` or `Receiver 2`, but not by both. If only one of the receivers is ready to receive, then the receiver that is ready will receive the datum, otherwise it could be either receiver which receives the datum. This latter case demonstrates that send and receive have an element of non-determinism.

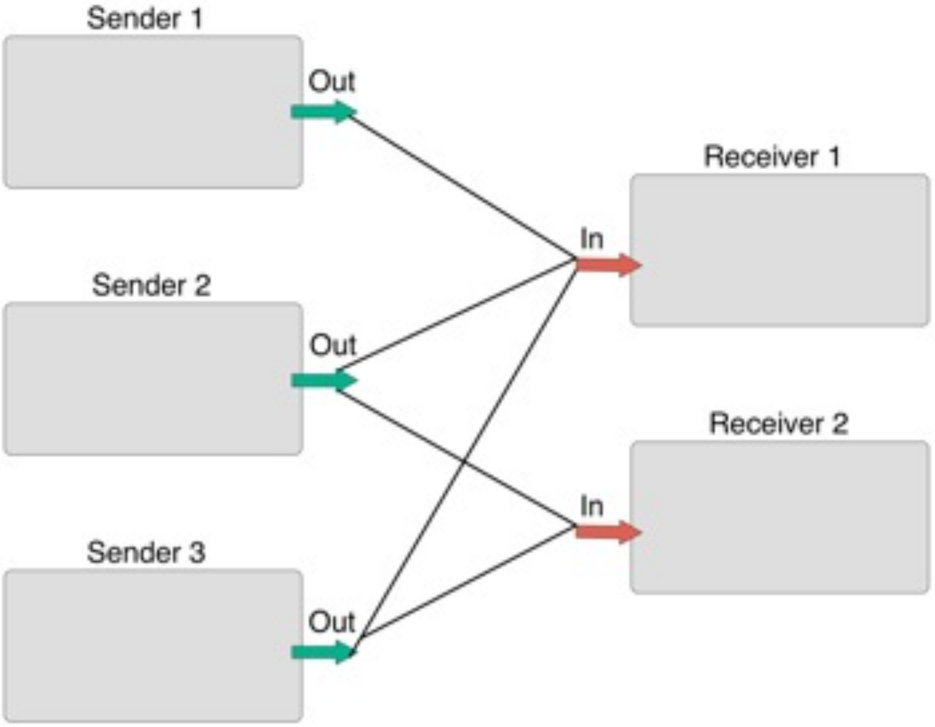


Figure 3.3 Topology with five components connected by nine channels.

Insense was designed for sensory systems that are often used by people that have no background in computer science. It was created to be an easy to use programming language with syntax that is not difficult to learn and use to most (even unskilled) programmers. Handling of memory management that is often problematic in languages like C is not required from developers that use Insense.

3.2.1 Components and Channels in Insense

All basic building blocks of Insense - components - are assigned with a type. The type of the component is described by its interface, which can contain any number of channels via which it may interact with other components. Figure 3.4 illustrates a "Hello World" example program written in Insense. It features one component that prints "Hello World" in its `behaviour` function. All Insense components are active and their activity defined by a syntactic construct identified by the keyword `behaviour`. The behaviour block may be likened to a single-threaded function which loops until explicitly stopped by using the `stop` keyword.

```
type IHello is interface () // Interface for Hello component
component Hello presents IHello { // Hello World component
    count = 1
    constructor() {
    }
    behaviour {
        printString("\nHello World ")
        printInt(count)
        count := count + 1
    }
}

hello = new Hello() // Create and test component
```

Figure 3.4. “Hello World” program written with Insense language. (Lewis 2013)

Insense components are instantiated by using constructors that are defined in a similar manner to Java, C++ or C#. An Insense component declaration must contain at least one constructor. These constructors permit component variables to be initialised. A keyword `presents` follows a definition of a component, it is followed by a number of interfaces that the component presents for interaction with other components.

The `Hello` component in Figure 3.4 presents the `IHello` interface which in this simple example program does not contain any channels. It has a local variable called `count`. Declaration is done using the “`=`” symbol. Variables declared inside of components are local and they cannot be accessed from outside of their declaring component. The scope of variables declared inside of components is until the end of the scope of the component. The value of `count` is updated by using the “`:=`” assignment operator in the `behaviour` function.

As mentioned before, Insense components are strongly encapsulated. Channels have a role of being the only medium for inter-component communication.

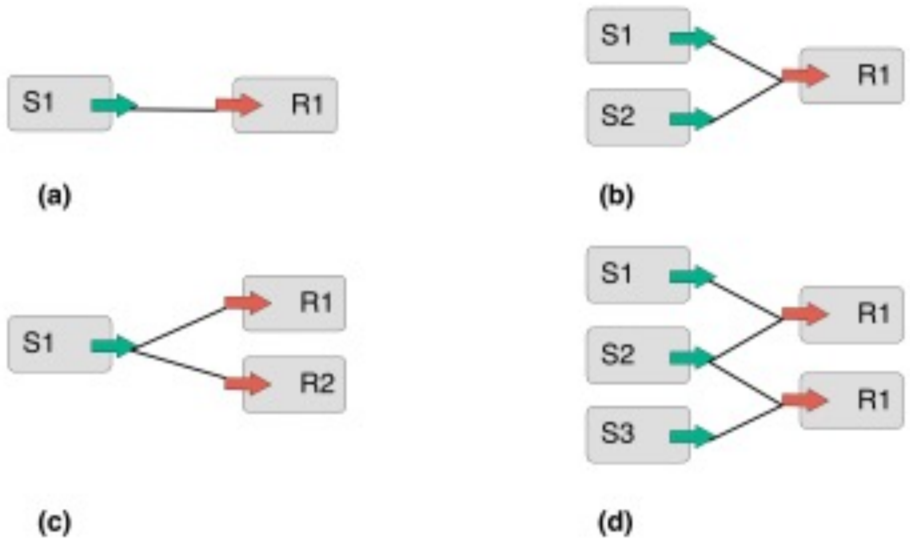


Figure 3.5 Connection topologies supported by channels in Insense. (Sharma et al. 2009)

Figure 3.5 (a) represents a one-to-one connection between a sender component `S1` and a receiver component `R1`. The semantics of sending and receiving data over a one-to-one connection are similar to sending data with use of a traditional pipe: all values sent by `S1` are received by `R1` in the order they were sent. (Sharma et al. 2009)

The connection topology shown in Figure 3.5 (b) outlines a many-to-one connection pattern in which a number of output channels from one or numerous components may be connected to an input channel associated with another component. For this topology, `R1` non-deterministically receives values from either of sending components on a single incoming channel. The receiving component cannot determine the identity of the sending component or the output channel that was used to send the message. Also, the arrival order of messages is determined by scheduling algorithm in use. The pattern is useful as a multiplexer in which `R1` can multiplex data sent from `S1` and `S2` and could forward the data to a fourth component. The multiplexer pattern is utilised to allow multiple components to connect to a single shared output channel.

The topology in Figure 3.5 (c) shows a one-to-many connection pattern between a sender component `S1` and two receiver components `R1` and `R2`. Each value sent by `S1` is non-deterministically received by either `R1` or `R2`. Values are received only by one of the connected to the source of data components. Also, for the sender it is irrelevant which component receives sent data. An example scenario for the one-to-many connection pattern used in

Insense is that a sender wishes to request a service from an arbitrary component in a server farm. Each of the three basic patterns of connectivity depicted in Figure 3.5 (a) - (c) may be modified and extended to create further variations. Figure 3.5 (d) shows an example variation combining the patterns from Figure 3.5 (b) and Figure 3.5 (c). (Sharma et al. 2009)

```
type ITempReader is interface ( out bool tRequestChan ; in integer
tValueChan)
component TempReader presents ITempReader { // Temperature reader
    constructor() {
    }
    behaviour {
        send true on tRequestChan
        receive temp from tValueChan
        printString("\nTemp = ")
        printInt(temp)
    }
}

tr = new TempReader()
connect tr.tRequestChan to sensors.tempRequest
connect tr.tValueChan to sensors.tempOut
```

Figure 3.6 Insense application with a temperature sensor and two channels. (Lewis 2013)

The example illustrated in Figure 3.6 has two channels associated with the component. Interfaces that the component presents are listed on the type declaration stage: `out bool tRequestChan ; in integer tValueChan`. Interfaces describe how instances of components can interact with each other. In this example, the `TempReader` component has channels named `tRequestChan` and `tValueChan`. Keywords `in` and `out` specify direction of the channel: channels can be used for sending data (`out`) and receiving it (`in`). Also, a type of data that channel is meant to receive needs to be declared, in the example above the `tRequestChan` channel permits data of type `Boolean` to be sent and the `tValueChan` receives integers.

3.2.2 Data Types

Insense supports most data types that can be found in other programming languages, including: `integer`, `real`, `Boolean`, `byte` and `enum`. In addition, it supports structs, enumerations, and arrays similar to those found in many other programming languages such as Java or C++. (Dearle 2011)

```
newArray = new integer[3][3][2] of 0
```

Figure 3.7 Declaration of a 3-dimensional array in Insense.

In Insense, arrays must be given a predefined size and they have to be initialised on declaration. Figure 3.7 shows an example of array declaration: in this case a multidimensional array (array of arrays of arrays) of integers was created, all elements inside of it are initialised to 0.

```
type Car is struct( string model; integer miles )
car1 = new Car( “mlsb”, 4000 )
```

Figure 3.8 Creation of a structure and an instance of it.

Figure 3.8 shows that structures may be defined by using the keyword `struct`. In order to permit static worst-case size estimation of components and their data, Insense structs cannot contain references to other structs. Therefore, structs can be formed from all other data types, other than other structs. Structures are instantiated by using the keyword `new`.

In Insense functions can be declared either as global (outside of components) or as component local declarations.

```
proc multiply(integer i, j,) : integer {
    return i * j
}
```

Figure 3.9 Function declaration in Insense.

Figure 3.9 presents an example of a global function that performs multiplication of two integers that is visible to all components in a compilation unit. On the other hand, locally defined functions can be accessed only by the component that contains the definition.

3.3 Memory Management in C Language

In C, there are two ways of allocating memory for variables:

1. *Static allocation* is used when one declares static variables. One block of space is allocated for each static variable, its size cannot be changed and it cannot be freed manually. The allocated piece of memory is freed when the application that used it stops.
2. *Automatic allocation* is called when developers create automatic variables (e.g. local variables or function arguments). Space for variables of this type is allocated when a

compound statement that contains the variable is entered; it is freed when the statement is exited.

In C, variables are not dynamically allocated on the heap. However, blocks of memory can be dynamically allocated on a heap and their base addresses are then stored in variables of an appropriate pointer type. This process is known to require more computational time and programmers tend to use it when static and automatic allocation cannot be utilised. The most common method used for dynamic memory allocation is utilisation of a single shared (big) heap structure, where a program can ask for a block of memory to store an object, and request that block to be deallocated at any time during execution of the computer program. A heap is a pool of memory available for the allocation and deallocation of arbitrary-sized blocks of memory (Wilson et al. 1995). Most of the basic design used for implementing memory management with shared heaps was achieved in the 1960's (Knuth 1997).

Data is allocated into contiguous and nonoverlaping ranges of memory. Generally, only entire blocks are allocated or freed, and the allocator is unaware of the type of or values of data stored in a block, it only knows the size requested.

3.4 Utilising Private Heaps for Memory Management

To refine the effective memory hierarchy performance in multithreaded applications, a number of hardware solutions have been proposed. Methods such as tiling and thread scheduling used for cache locality are a few examples of known techniques (Park, Hong & Prasanna 2003, Philbin et al. 1996). This project focuses on moving away from the idea of using one large heap for memory management and going towards utilisation of private heaps instead. It is seen as a way to improve performance and efficiency of memory management in multithreaded applications.

There is no functionality in the C standard library that permits programmers to make use of separate private heaps for individual threads or processes. However, this project seeks to design and implement a version of Insense that permits memory to be allocated on private heaps as opposed to the standard single shared heap offered by the standard library. The motivation for this work is that memory management can be more efficient when multiple private heaps are used. There are a number of reasons why this is a case ("*Managing memory*"[2] 2013):

1. Allocation into a single heap often creates memory blocks that reside in different pages of memory. As an example, let us consider allocation of memory for items of a linked list. If one allocates memory for other data between adding nodes of the list, the blocks linked with nodes of the list may end up on multiple different pages. It means that accessing data from the linked list would potentially involve swapping multiple pages, jumping from one page to another. Utilisation of multiple private heaps implies that one can specify which data is allocated in which heap. It allows reduction of overhead created by swapping between pages of memory since data that is often accessed at the same time (in case of the example before: data stored in the elements of the linked list) can be stored in fewer pages, and it remains close together.
2. Applications dealing with multiple threads can benefit from using private heaps as well. The reasoning goes as follows. Using a single large shared heap as a memory resource, no more than one thread can allocate memory into a heap at the same time. The reason for this restriction is to ensure that memory is safely allocated and deallocated in the shared resource. As such, allocation and deallocation of memory in a single heap shared by multiple threads must be serialised via mutexes or semaphores. This slows down allocation and deallocation due to threads having to wait until they can access the shared resource. Performance could be improved if a separate private heap is created for each thread, reducing both the time that threads have to wait and the overhead needed for serializing access to the heap.
3. Private heaps could also be created for specific data structures (we may take the example dealing with the linked list from point 1 of this list into account). If one has a linked list that has its elements allocated in various pages in memory, freeing the blocks must be done individually, which can take time. If a separate heap is created specifically for that list, deallocating the whole list could be done by a single call that destroys the entire data structure. In this case, freeing memory could be made more efficient and performance could be improved. This scenario is not investigated in this project.
4. Another drawback of using a single shared heap for memory management becomes apparent when a large number of allocations and deallocations of data of the same large

[2] Citations of references with no author are given by the first few words of the title and year.

size take place. With time, memory space may not be able to fit in any more large data due to adding smaller pieces of information into memory between allocation and deallocation of large data. For example, if one has a block of free memory of size 1MB, allocating even 1B into that region makes it impossible to allocate another 1MB into the same block.

3.5 Thread Affinity

Another aspect of multithreaded applications, placement of threads, may be shown to affect performance. Thread affinity allows to bind and un-bind threads to a physical CPU or to a range of CPU cores, rather than allowing threads to run on any core. In this way a thread will run only on a core or cores in question.

Mechanisms that allow thread affinity scheduling prove to be of growing interest since in modern machines amount of time needed to access a memory location cached locally versus one held in main memory is usually significantly lower. (Salehi, Kurose & Towsley 1995, Devarakonda, Mukherjee 1992, Gupta, Tucker & Urushibara 1991). Achieving successful and efficient affinity of threads may take advantage of speedup that can be received through accessing locally cached memory. By using thread affinity and avoiding dynamic placement of them programmers wish to reach such situations where work can be transferred for execution onto separate processors, when feasible and profitable.

Static affinity of threads can be beneficial for optimizing cache usage: a number of times the program must switch processors can be significantly reduced, which lowers a rate of flushing one cache and repopulating another. Additionally, thread affinity takes advantage of the fact that cache used by the thread may store data that was put there during one of the previous runs (Yang 2010). Research in the field of static thread affinity has been ongoing for more than 20 years now (Vaswani, Zahorjan 1991). We intend to investigate to what extent performance of programs and utilisation of cache may be improved by statically setting the processor affinity of threads in Insense.

4 SOFTWARE ENGINEERING PROCESS

This Section touches issues connected with principles of software engineering that were employed throughout the work on the project. Multi-core Insense was developed by following concepts of evolutionary (incremental) development. Figure 4.1 outlines main stages that projects which follow the model go through. Incremental software development is known to be better than a waterfall approach for most business, e-commerce, and personal systems (Sommerville 2011). The evolutionary development model was chosen due to its characteristics that seemed to be promising for a project of this type and scale. Firstly, this method is beneficial for a project that in its objective has creation of a part of a large system ("*Understanding*" 2006). Such risks as missing deadlines, creating unusable products of low quality can be addressed and managed by breaking the project into smaller, more manageable pieces (May, Zimmer 1996).

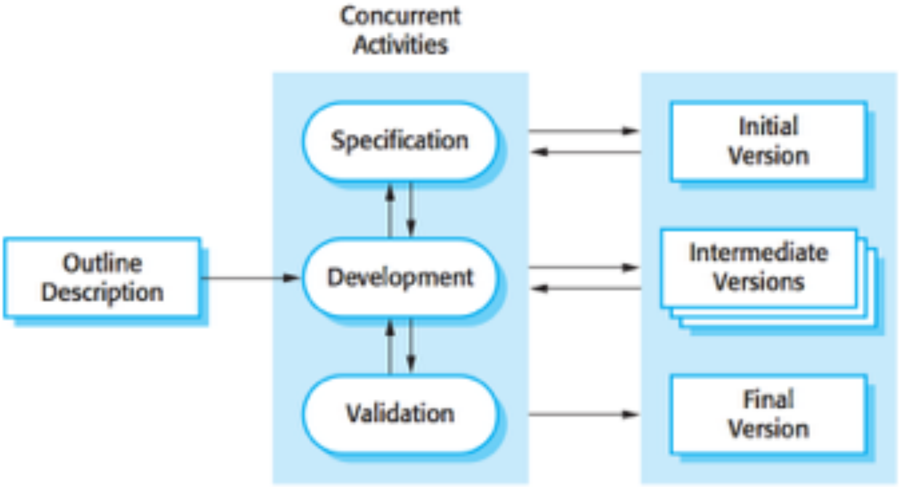


Figure 4.1 The evolutionary development model. (Sommerville 2011)

At first, a *description* of the project was outlined. Research goals, system's objectives and constraints were established on this stage. The main goal was defined as successfully analising benefits of using schemes of memory management and thread placement discussed in the scope of this project. Outline requirements were specified for Multi-core Insense prior to starting the evolutionary development cycle.

4.1 Project Plan

After outlining description of the project, it entered the *implementation* phase where the more detailed design and implementation of the system took place. Work was conducted in cycles. As can be seen from the Figure 4.1 above, each cycle consists of three main concurrent activities: gathering *specifications*, conducting *development*, and *validating* results. Three main stages can be outlined: 1. Insense was ported to Unix. 2. A number of methods utilised for management of memory were first designed and then implemented. 3. Schemes of placement of threads were designed and implemented. The initial, intermediate, and final versions were created. Multiple intermediate versions were achieved by following feedback received through validation of cycles of design/implementation.

The time scale of the project was around five month. More than a half of this time was spent on understanding Insense and software systems behind it and working on the base implementation that allowed designing, implementing and eventually experimenting with memory management and thread placement. Appendix 2 contains a Gantt chart for the project that was used to orgranise work.

5 REQUIREMENTS SPECIFICATION

A number of outline requirements were established on the initial stage of the project.

5.1 Functional Requirements

Functional requirements describe how the system should and should not behave, what kind of services it should provide and what type of output it should produce when it is given certain kinds of input (Sommerville 2011). The following list of initial (outline) requirements was established for the project:

Components:

1. The system must be able to create, assign to POSIX threads, and stop components.
2. When on multiple cores, components must be able to run in parallel.
3. Components must be able to communicate with other components through channels.

Channels:

4. The system must be able to create, destroy, bind together and unbind channels.
5. Channels must be able to serve as a medium for transferring data of such types as integers, strings, arrays etc.
6. Channels must support 1:1 and N:M connection types (described in Section 7.3).

Memory management:

7. Support for allocation and deallocation of memory into a single shared heap and into multiple private heaps must be given.

Thread placement:

8. Support for the following schemes of thread placement: 1) dynamic placement, where OS takes care of setting affinity; 2) static placement, Insense components are distributed evenly among cores; 3) all components are placed on a single core; 4) locating Insense components based on frequency of communicating with other components

Insense Compiler and Runtime:

9. Insense programs must be able to run on a 64-bit architecture.
10. Users must be able to choose among supported data placement scheme.
11. Users must be able to choose among supported thread placement scheme.

5.2 Non-functional Requirements

Non-functional constraints are normally put on the services or functions offered by the system. To name a few, this type of requirements includes constraints on the development process, timing constraints, and constraints imposed by standards. (Sommerville 2011)

The following non-functional requirements were outlined for this project:

1. Keep changes to the Insense compiler to a minimum so as to maximise compatibility with existing versions.
2. The project must be finished and results delivered by June 26th 2013.
3. Updates on achieved progress must be given to a supervisor on weekly basics.

6 ETHICS

This project did not involve conducting experiments that require participation of humans. As such, there were no ethical concerns associated with this project. Appendix 10 contains a scanned copy of the ethics approval form.

7 DESIGN

This Section summaries design process that was undertaken in the project and discusses taken design decisions.

7.1 Design of Multi-core Insense

In order to successfully use Insense on a multi-core Unix-based system a number of essential elements of the language had to be redesigned.

7.1.1 Components

A component is active and the unit of concurrency in Insense. For the Contiki-based implementation, component behaviour was represented using proto-threads, a light-weight form of thread provided by the Contiki OS. Under InceOS, behaviour of components was embodied by utilising an active and pre-emptable component abstraction provided by InceOS. For the Unix implementation, component behaviour could either be represented using Unix Processes or POSIX threads (p-threads).

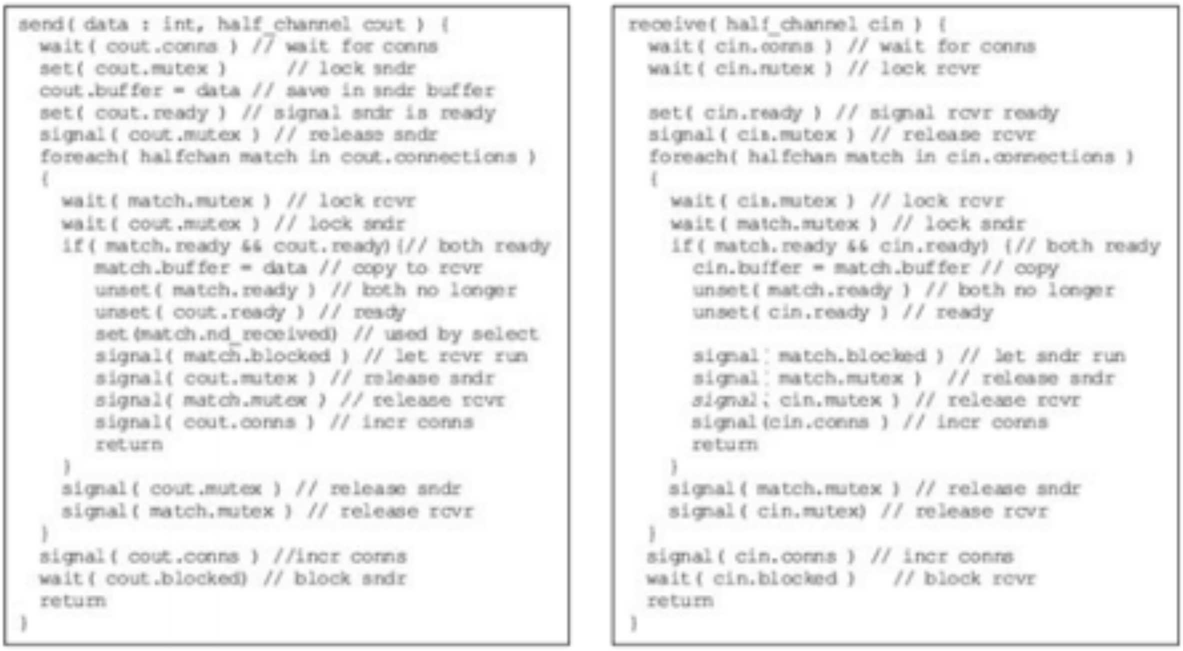
A decision was taken to use POSIX threads because threads are commonly considered to be more light-weight than processes in consuming less resources in the OS (Stallings 2009). Further, component communication over channels is likely to be easier to design and implement when components are represented as threads as these may communicate via a shared virtual memory space without the need for additional Inter-Process Communication (IPC) mechanisms when communicating between different processes.

The following operations on components must be supported by the multi-core Insense runtime:

1. *Creation of components*: a POSIX-type thread is created and the component behaviour will be implemented by the new thread. If private heaps are utilised for memory management, as discussed in Section 7.2.2 below, a new heap assigned to the component is created at this stage. Also, if required, thread affinity can be set, as discussed in Section 7.3.
2. *Stopping components*: the thread implementing the component's behaviour is stopped. It is achieved by changing value of a `Boolean` flag that indicates whether component is running or not to `FALSE`, as discussed in more details in Section 8.1.1.

7.1.2 Channels

A decision was taken to use the algorithms proposed in (Sharma et al. 2009) for this project. The reason behind using these algorithms is that they have been verified by the scientific community and successful experimental results were achieved, as described in the aforementioned paper.



```
send( data : int, half_channel cout ) {
  wait( cout.conns ) // wait for conns
  set( cout.mutex )      // lock sndr
  cout.buffer = data // save in sndr buffer
  set( cout.ready ) // signal sndr is ready
  signal( cout.mutex ) // release sndr
  foreach( halfchan match in cout.connections )
  {
    wait( match.mutex ) // lock rcvr
    wait( cout.mutex ) // lock sndr
    if( match.ready && cout.ready){// both ready
       match.buffer = data // copy to rcvr
       unset( match.ready ) // both no longer
       unset( cout.ready ) // ready
       set(match.nd_received) // used by select
       signal( match.blocked ) // let rcvr run
       signal( cout.mutex ) // release sndr
       signal( match.mutex ) // release rcvr
       signal( cout.conns ) // incr conns
       return
    }
    signal( cout.mutex ) // release sndr
    signal( match.mutex ) // release rcvr
  }
  signal( cout.conns ) //incr conns
  wait( cout.blocked) // block sndr
  return
}
```

```
receive( half_channel cin ) {
  wait( cin.conns ) // wait for conns
  wait( cin.mutex ) // lock rcvr

  set( cin.ready ) // signal rcvr ready
  signal( cis.mutex ) // release rcvr
  foreach( halfchan match in cin.connections )
  {
    wait( cis.mutex ) // lock rcvr
    wait( match.mutex ) // lock sndr
    if( match.ready && cin.ready) {// both ready
      cin.buffer = match.buffer // copy
      unset( match.ready ) // both no longer
      unset( cin.ready ) // ready

      signal( match.blocked ) // let sndr run
      signal( match.mutex )  // release sndr
      signal( cin.mutex ) // release rcvr
      signal(cin.conns ) // incr conns
      return
    }
    signal( match.mutex ) // release sndr
    signal( cin.mutex) // release rcvr
  }
  signal( cin.conns ) // incr conns
  wait( cin.blocked )    // block rcvr
  return
}
```

Figure 7.1 Send and Receive algorithm proposed by (Sharma et al. 2009).

Figure 7.1 above outlines algorithms for `send` and `receive` operations. The design of these functions is almost symmetric, as can be seen in the algorithms. Both operations check whether any components are waiting in the list of connections with the sender looking for a waiting receiver and vice-versa. If no such match is found the sender or receiver blocks on the blocked semaphore until it is re-awakened by at least one component moving to the state that allows sending/accepting data.

In addition to the send and receive operations in Figure 7.1 above, a number of other operations are required to support channel communication in Insense. These include creation of a new channel with given payload type and direction and connection and disconnection of component channels. The channel operations are described in more detail in (Sharma et al. 2009).

7.1.3 Program Entry Point

Under Unix, it was necessary to design and implement a system-level entry point to the entire Insense program. This entry point may be likened to the main function in a C program or the main method in Java. It was decided that such a main function should be defined once in the Insense Runtime library and should be designed to instantiate any necessary data structures and then hand over control to a programmer-defined entry point in the Insense program.

Insense programs commonly contain declarations of any global procedures (for use by all components) followed by component declarations and finally a sequence of code that serves as the user-level entry point to the program, i.e. where user-level program execution begins. In InceOS, this entry point was represented as a schedulable InceOS component and in Contiki, it was represented as a Contiki process. Under Unix, a decision was taken to represent the programmer-defined entry point as a function that is called by the system-level entry point as explained above.

Any dynamically allocated memory required by the main thread is handled by using the `malloc` function. Hence, even when the "private heaps" scheme is in use, memory required by the main component will be allocated dynamically, into the shared heap. The main thread requests allocation of memory for a little amount of data: one descriptive string per component and for what is passed as members of the `argv` array of program arguments. This design decision was made since it may be argued that the amount of dynamically allocated data required by the main thread is minimal and location of it does not affect experiments involving private heaps that are outlined further in this document.

7.1.4 Memory Management & Garbage Collection

Insense uses a special reference-counting garbage collection scheme for dynamically allocated memory. With this method, each allocated block of memory is prepended with a piece of memory that stores information about the allocated memory range.

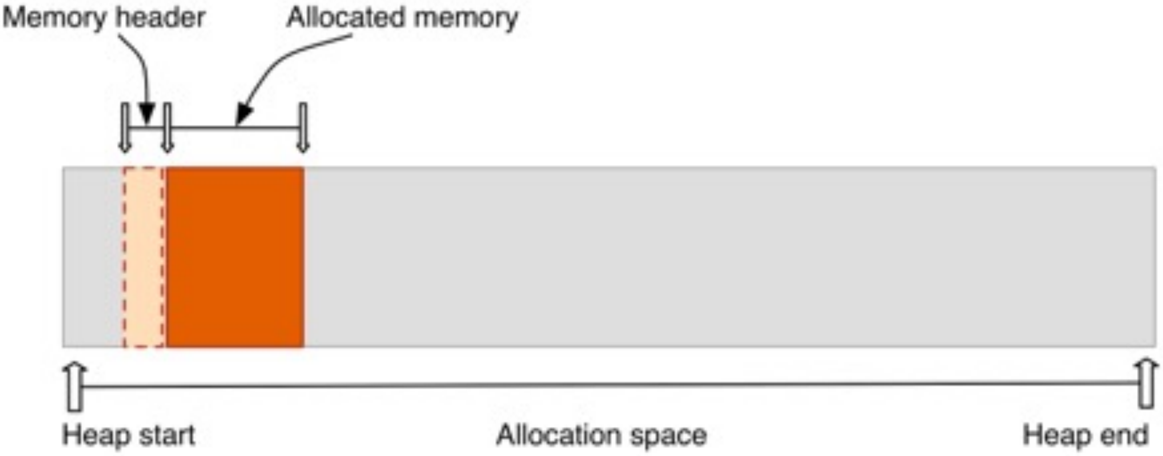


Figure 7.2 Header prepended to allocated memory (in the shared heap).

The number of references to every object dynamically allocated in Insense is accounted for and stored in the memory header. Whenever Insense variables are assigned to dynamically allocated data (such as an array, channel, component, struct, any) the reference count in the memory header is adjusted to reflect the assignment. Whenever a reference count reaches zero, when it is no longer referenced by any variable, the object is considered as ready to be “garbage collected”/destroyed. Moreover, the process of garbage collecting recurses down and it deallocates memory assigned for all objects that depend of the object that needs to be garbage collected (e.g. elements of an array). Figure 7.2 shows a schematic diagram that outlines how a header is attached to a dynamically allocated region of memory.

7.2 Data Placement Schemes

Two data placement schemes were designed. The final design of the system permits memory for dynamically allocated objects to be allocated in one shared large heap and in multiple small private heaps.

7.2.1 Shared Heap

In this scheme, memory is allocated dynamically for Insense components, structures, arrays or channels at runtime in a single shared heap.

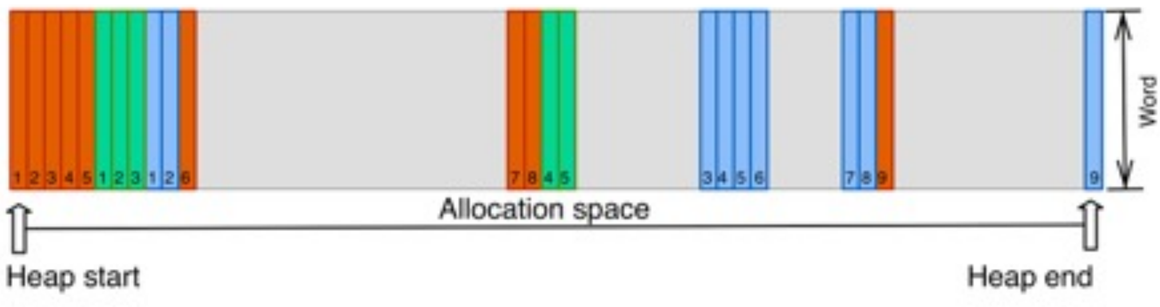


Figure 7.3 Dynamically allocated memory for three components inside of the shared heap.

Data allocated by different threads inside of one Insense program is not separated. Figure 7.3 demonstrates a possible scenario where three Insense components make use of a shared heap, colours represent memory allocated by different components. Allocated memory is numbered, based on order of allocations taking place in each thread. Areas of grey colour between allocated memory indicate free fragments of the allocation space. The design of the "shared heap" allocation mechanism merely requires adoption of the standard library mechanisms available in C, as these allocate on a single shared heap.

As indicated in Figure 7.3 above, memory allocated by particular threads may become fragmented and dispersed over the shared heap depending on the interleaving of allocations and deallocations by different threads. Also, as already mentioned in Section 3.4 above, allocation into the shared heap by multiple threads will be serialised. The issues surrounding fragmentation and concurrent access to a single shared heap from multiple threads are discussed in more detail in Section 3.4 above.

7.2.2 Private Heaps

For Insense on multi-core Unix, it was decided to investigate a different memory management mechanism to that discussed in Section 7.2.1 above, where each component is given a small private heap in which dynamic allocation of memory can take place. Figure 7.4 shows a diagram with three components that allocate memory into private heaps that were assigned to them (represented by different colours). The allocation space now comprises three heaps. This illustration contrasts with a Figure 7.3 in the previous section where allocation is performed into a single shared heap. Similar to the previous figure, numbers indicate ordering of allocation within each thread. Although the exact sequence of the various components' allocations may take place in any order in time, allocated memory is inputted into one private heap and it is contiguous in space.

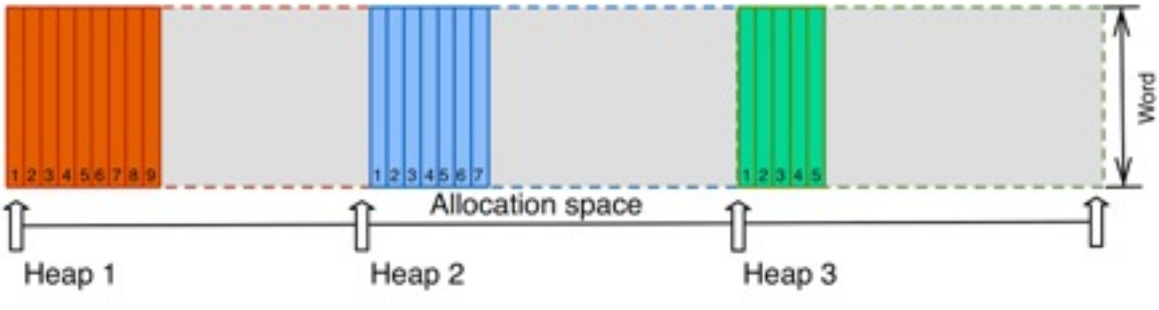


Figure 7.4 Dynamically allocated memory for three components inside of private heaps.

The most significant advantage of this scheme is that multiple threads can allocate in their private heaps concurrently, whereas threads that allocate into one large heap have serialised access (with mutex locks). Another advantage of this scheme is that when memory required by one thread is allocated contiguously into the same heap, the likelihood of allocated data residing in one page of allocation space is higher, compared to the "shared heap" scheme. Hence, performance of the program may be improved by reducing overhead created by swapping multiple pages.

7.2.3 Data Placement and Potential Cache Misses

Another problem that systems which work with dynamic allocation of memory face is cache misses. When a program accesses a data item for the first time, the data item will be hauled into a fast cache close to the processor (prior to access) from the next-level cache or main memory, which costs time. Cache misses occur whenever a program tries to access an uncached data item (Gove 2011, Hennessy 2007). A data item may be uncached because it has *a)* never been accessed before (compulsory miss) or *b)* it has been accessed before, but had to be evicted from the cache when another data item was accessed by the program (conflict miss). Cache misses impact on performance because they may result in the processor having to wait (stall) while the data item is cached. Much research has been done in the area of reduction of the number of cache misses that occur during execution of a program, there is a number of parameters that can be tweaked for receiving gains in performance: size of cache, associativity, block size etc. (Wulf, McKee 1995, Ghosh, Martonosi & Malik 1997, Lam, Rothberg & Wolf 1991)

The level 1 caches used in most computers today are either directly mapped caches or set associative caches (Handy, 1998). In case of a directly-mapped cache, each location in main memory can only be put into one cache line in the cache. When the cache is set associative, instead of mapping every address to a particular cache line, every address maps to a particular set of a certain number of cache lines (e.g. 8), but can be placed in any of the cache lines within that set.

Figure 7.5 shows how the *Offset* and *Index* portions of the memory address specify where the datum may be placed and retrieved in a directly mapped cache. The *Index* portion of the address specifies the cache line and the *Offset* where the datum is within that cache line. A *Tag* that represents a significant part of the memory address is also stored with each cache line and is used to record the remaining higher parts of the memory address of a datum that is held in the cache. When a system attempts to access a data item from memory, the *Tag* stored in the cache line is compared to the *Tag* portion of the address that is being accessed in order to determine if the requested data item is cached or not. In case of a set-associative caches, the *Index* indicates which set of cache lines can be utilised to cache a datum with a given memory address. Within that set, any unused cache line among the cache lines in the set can be used, or one must be evicted prior to caching the new datum if they were previously all in use. As before, the *Tag* is used to store the higher-order bits of the memory address to indicate which address is actually cached in the cache and the *Offset* locates a datum within a specific cache line.

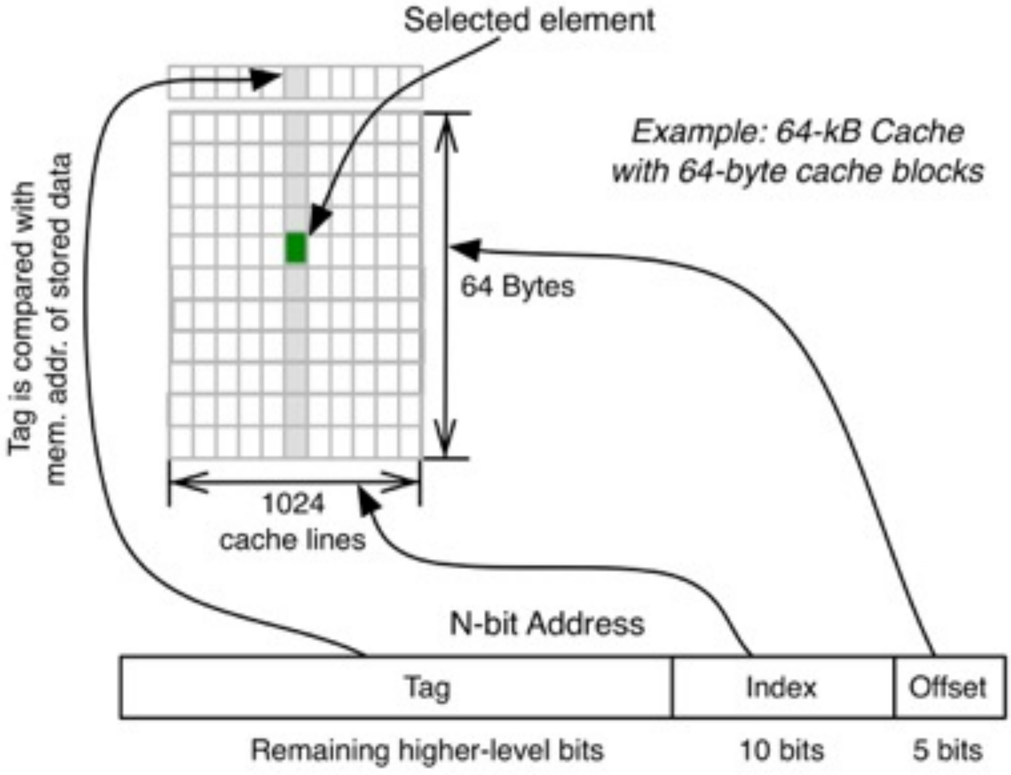


Figure 7.5 Retrieving data from directly mapped cache.

When a core has access to more data allocated in the main memory than can fit inside of the L1 cache, which is often a case, certain blocks of memory are linked to the same location inside of the cache. The main idea of set-associative caches is to reduce the chances of conflict misses because for any memory address there is a number of possible cache lines to use rather than a single line in the direct-mapped cache. However, both the directly mapped and the set associative caches suffer from high rates of cache misses (Hill 1988).

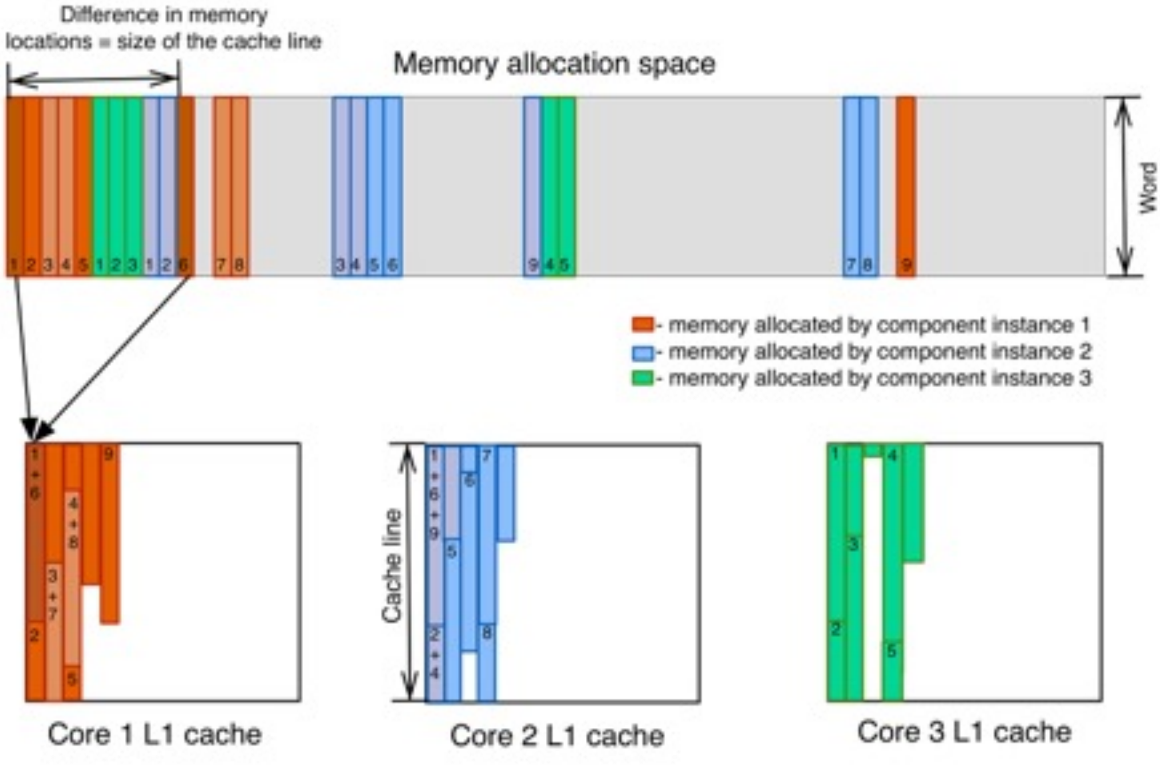
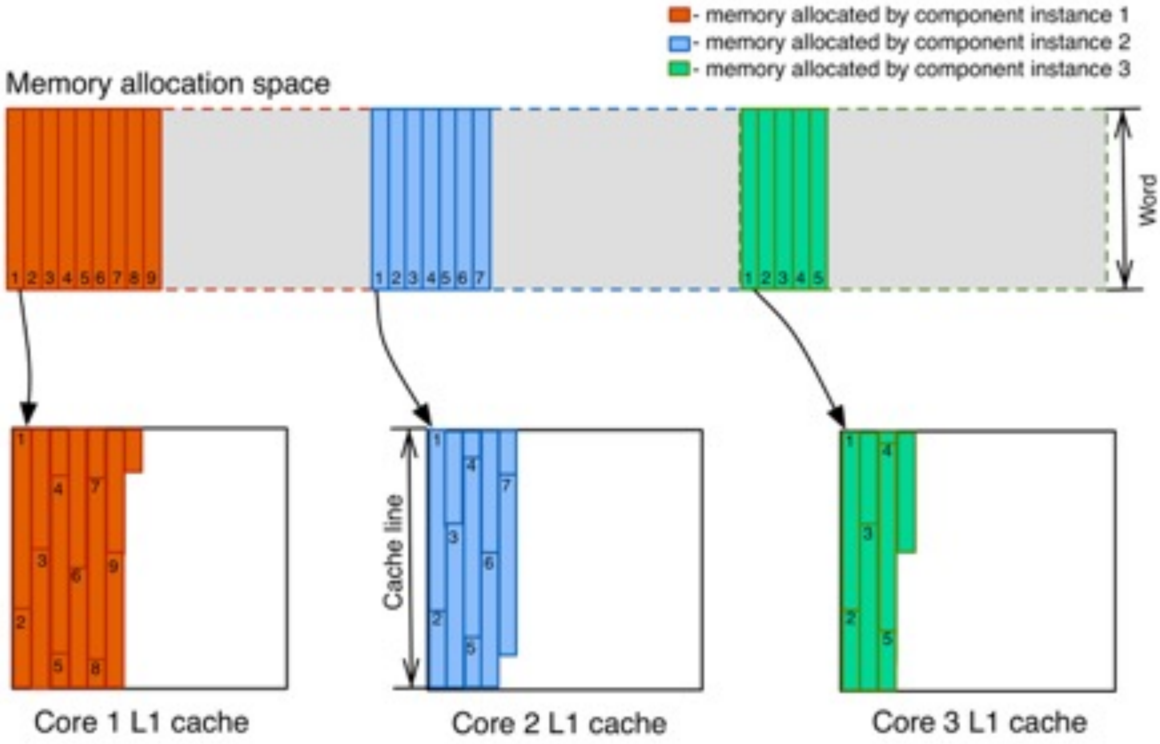

Difference in memory locations = size of the cache line
Memory allocation space
1 2 3 4 5 1 2 3 1 2 6 7 8 3 4 5 6 9 4 5 7 8 9
Word
- memory allocated by component instance 1
- memory allocated by component instance 2
- memory allocated by component instance 3
1 + 6
2
3 + 7
4 + 8
5
9
Cache line
1 + 6 + 9
2 + 4
5
6
7
8
1
2
3
4
5
Core 1 L1 cache
Core 2 L1 cache
Core 3 L1 cache

Figure 7.6 Cache misses with the “shared heap” scheme.

The “shared heap” scheme utilised for memory management in multi-core Insense is expected to be affected by conflict misses due to memory fragmentation to a greater extent than the "private heaps" scheme where fragmentation is anticipated to be less common. An example of the issue of cache misses can be understood when Figure 7.6 is taken in account. This diagram illustrates occurrence of cache misses in a system that uses a single shared heap for memory and has a directly-mapped cache in its processor. Multiple components are executed on different cores. Level 1 caches of three cores are represented on the illustration. The size of each cache line is a number of words, the exact number of words can be left undefined for this discussion. Data allocated by three component instances is fetched from main memory and placed into cache. Here we assume that a difference between memory addresses of blocks 1 and 6 of instance 1 (red) equals to the size of the L1 cache. These blocks are represented with different shades of red in the diagram. Based on the description of the mechanisms behind the directly mapped cache, one notices that block 1 and block 6 both map to the same cache line. As a result, when a system wishes to retrieve data stored in the red block 1, a cache miss will occur if block 6 was accessed previously. Similar logic can be applied to other blocks of memory where cache misses will eventually occur. Additionally, Red blocks 3 and 7, 4 and 8; Blue blocks 1, 3 and 9, 2 and 4 will create similar cache misses. They are also marked with different shades of their representing colour in the figure. A similar situation occurs when a set-associative cache is used. Only a number of cache misses is lower due to the fact that allocated memory maps to a particular set of cache lines. Hence, chances of a particular cache line being overwritten by data from main memory are reduced by a factor of the size of the set.



- memory allocated by component instance 1
- memory allocated by component instance 2
- memory allocated by component instance 3
Memory allocation space
Word
Cache line
Core 1 L1 cache
Core 2 L1 cache
Core 3 L1 cache

Figure 7.7 Cache access with the “private heaps” scheme.

Figure 7.7 attempts to show that a number of cache misses can be reduced if private heaps are utilised for memory management. Similar to the example involving Figure 7.6 described above, cache misses are possible when private heaps are used. However, because memory allocated inside of private heaps is contiguous, a number of cache misses is significantly lower, compared to the “shared heap” scheme. As a result, we would expect to see a performance increase and reduced number of cache misses for the "private heap" scheme compared to the "shared heap" scheme.

7.3 Thread Placement Schemes

Another aspect that may affect performance of applications written in Insense and running on multi-core systems is placement of threads that are responsible for behaviour of Insense components.

7.3.1 Dynamic Placement

The first scheme of choosing affinity that was considered in the scope of this project is dynamic placement of threads implementing Insense component behaviour. With this method, affinity of threads is handled by the operating system.

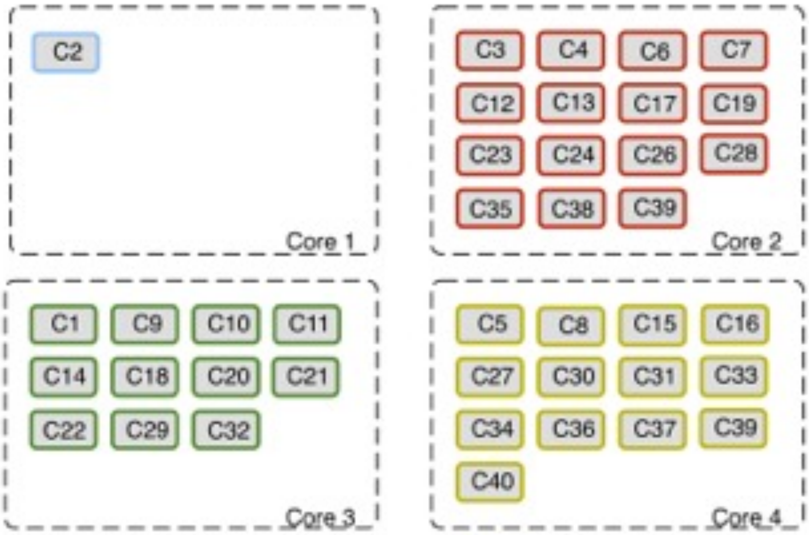


Figure 7.8 40 threads linked to Insense components dynamically placed on 4 cores.

Figure 7.8 illustrates how thread placement is handled on Unix-based machines. In this example the OS decided to place a single compute-intensive component instance `C2` on Core 1 in order to balance the execution load of the cores. Thread affinity is handled by live migration of threads to different cores. Unix-based systems use the hybrid (push/pull) migration algorithm for multiple-processor scheduling: *push* - the systems have a special task that checks the load on each processor every 200 ms; *pull* - a free processor takes a task from another processor that has higher load (Garg 2009). As a result, a computer program has no knowledge of which threads are placed on which cores. We intend to investigate the impact on performance, since efficiency of cache utilisation may suffer.

7.3.2 Static Placement

The alternative approach to dynamic placement of threads is setting affinity statically. The program or the programmer or, possibly, the compiler decides a core that a particular thread is pinned to. Threads can be associated with a particular core during the whole duration of the program's execution. Statically setting the affinity of threads can be beneficial for optimizing cache usage: a number of times the program must switch processors can be significantly reduced. It lowers a rate of flushing one cache and repopulating another. Deciding on the core that is used for setting affinity to a thread is not straightforward and it may be done by involving various approached and algorithms. In the scope of this project three methods of setting static affinity to threads are investigated:

1. *Even distribution among cores*: an algorithm similar to round-robin scheduling algorithm (Silberschatz 2009) is experimented with. A pool of cores available for setting affinity of threads is traversed circularly. Figure 7.9 demonstrates an example. In this diagram 40 Insense component instances are distributed between 4 processors. If one assumed that amount of computation performed by all components is uniform, it would be possible to say that all cores in the CPU perform approximately the same amount of work.

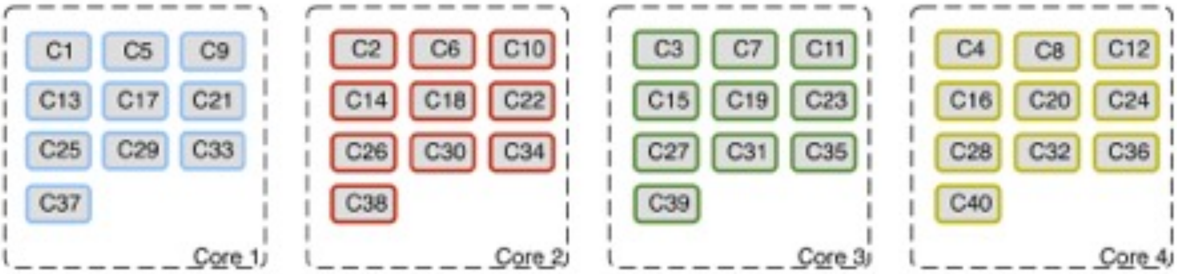


Figure 7.9 40 threads linked to Insense components statically placed with the "Round-Robin" algorithm.

2. *Assignment of threads to a single core*: threads are linked with a particular core. Figure 7.10 outlines a system where all Insense components are assigned to one core (in this case Core 1). Investigation of utilisation of this scheme is a part of this project since in some circumstances it may be beneficial to performance of the program.

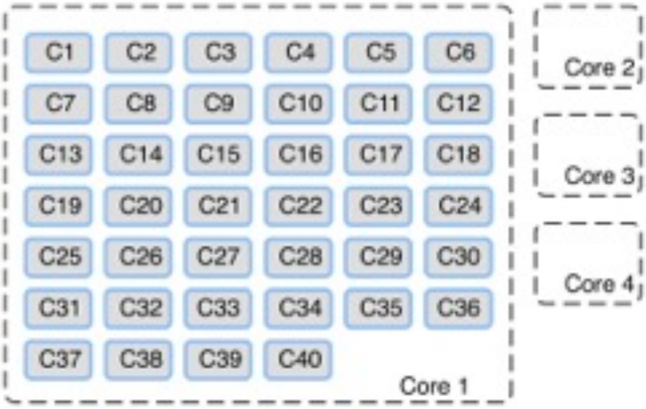


Figure 7.10 40 threads linked to Insense components statically placed on a single core.

This scheme is not expected to be particularly efficient other than for programs with inter-component dependencies that restrict active computation to a single component instance at any particular point in time.

3. *Assignment of threads to cores based on the nature of communication with other threads*: threads implementing component behaviour are linked to cores based on the nature of communication with other Insense components that they undergo. To explain this placement scheme, an outline of three different patterns of component operation and communication between components is given next. Figure 7.11 below demonstrates five patterns:

(a) *Sender - Receiver (1:1)* : in this pattern one or multiple sender component(s) send data (e.g. an `integer`) and receiver component(s) receive data. No calculations take place on either side. This scheme outlines a case where a built system relies heavily on communication between components.

(b) *Dispatch - Worker (1:N)* : Consider a *Dispatch* component which always sends a number and *Worker* component which receives that number and then performs amount of work controlled by the number. The *Dispatch* will not be able to send another number until any connected *Worker* component instances have done their work and execute their `receive` statement again (because the channels are synchronous).

(c) *Client - Server (N:1)* : In this scenario, one or multiple *Client* instances send requests to a single *Server* which receives the request and performs computation of some kind. There is no need to reply to the *Client*.

(d) *Many to Many (N:M)*: this is a more complicated case which can be seen as a combination of patterns (b) and (c) above. Component instances on both sides of the diagram (components 1 - 3 and 4 - 6) can perform computation.

(e) *No communication*: in this simple case no communication between components takes place.

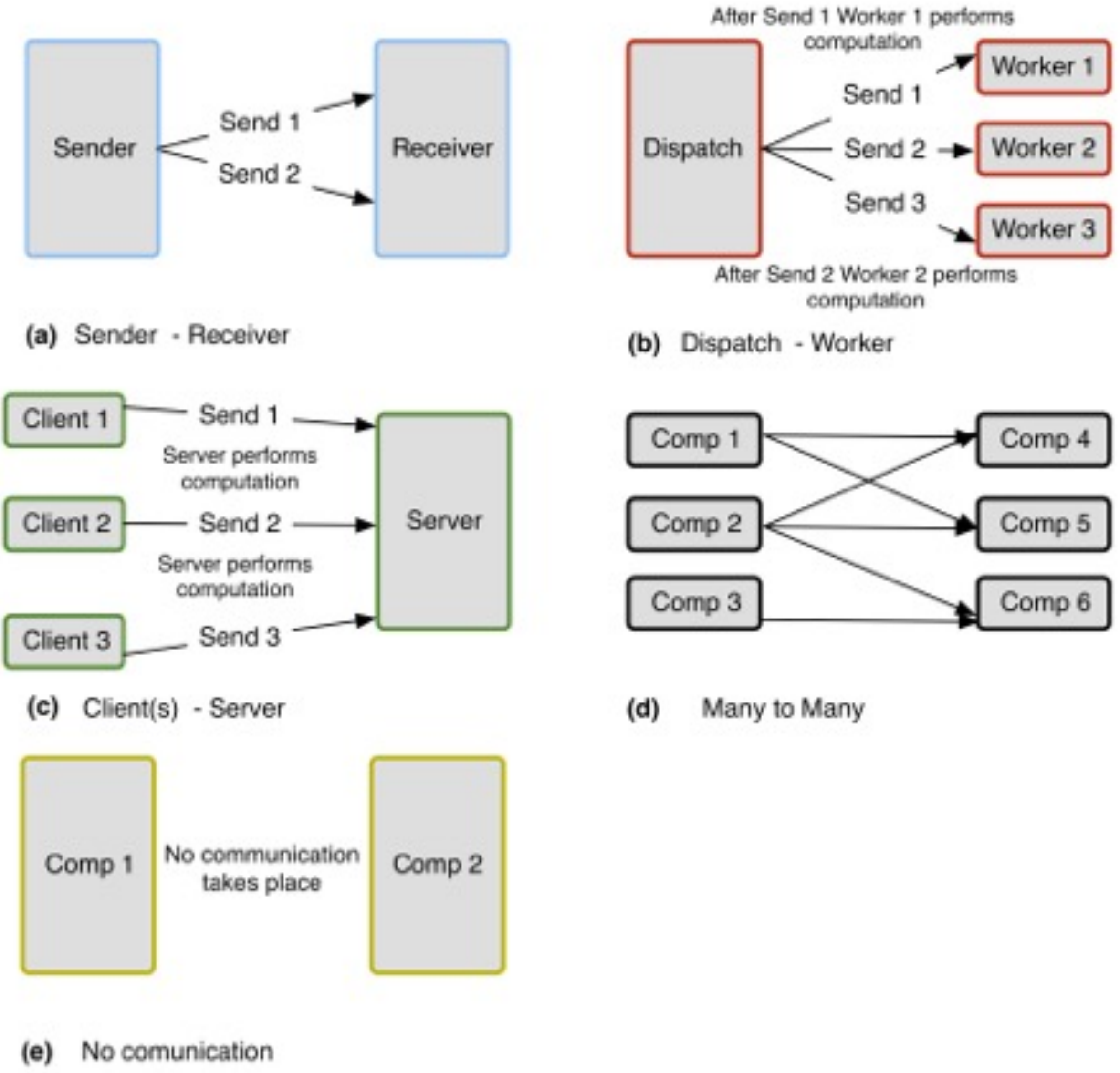
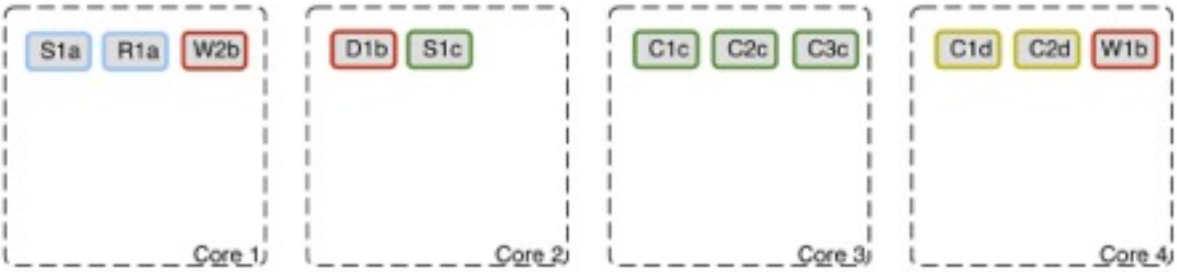


Figure 7.11 Component connectivity patterns.

Figure 7.12 reveals an example with four different groups[3] of components: “a” (marked with blue borders), “b” (red borders), “c” (green borders), and “d” (yellow borders).



[3] The word “group” in this Section refers to Insense components in Figure 7.11 that are marked with the same ending letter and the same border colour. E.g. for the component “S1a”, a letter “a” denotes the group of the component, which in this case is “a”.

Figure 7.12 11 threads linked to Insense components statically placed on cores based on the nature of communication with other threads.

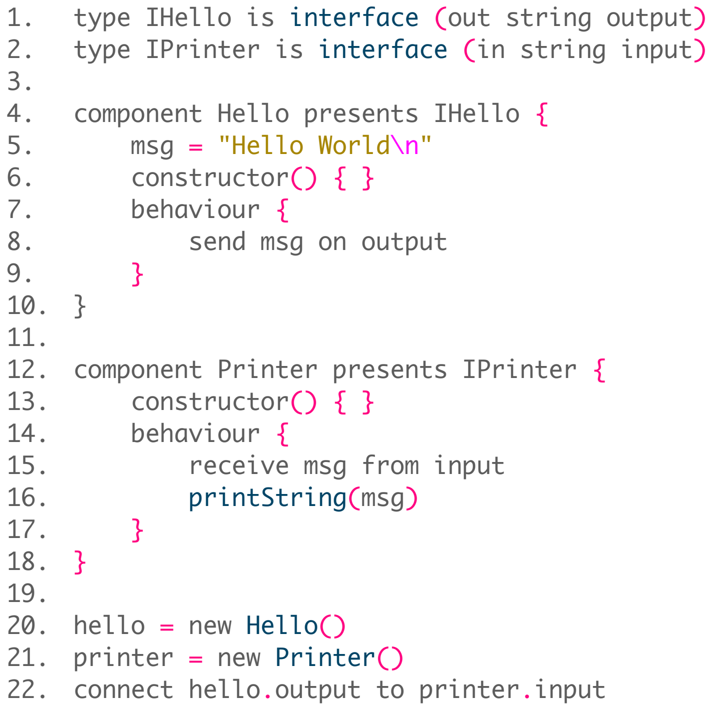
To continue with the example, let us assume that the following statements hold true:

- The group “a” of components may be described by the *Sender - Receiver* example. `R1a` must wait to receive a value before it can do anything else and `S1a` must send a value before being able to do anything else. In this case executing both components on the same core may be beneficial: as neither can make progress while the other component is doing something other than sending or receiving.
- The group “b” of components may be described by the *Dispatch - Worker* example. Component instance `D1b` is a *Dispatch* component and two instances - `W1b` and `W2b` - are instances of a *Worker* component. In this case executing `D1b`, `W1b`, and `W2b` on different cores may be beneficial in terms of maximising concurrency at the expense of more inter-core component communication.
- The group “c” of components may be described by the *Client - Server* example. `C1c`, `C2c`, and `C3c` are multiple *Client* instances and `S1c` is a single *Server* which receives requests sent by clients and performs computation. We argue that with this scenario it may be beneficial to execute the clients (`C1c`, `C2c`, `C3c`) on one core and the server `S1c` on a different core.
- The group “d” consists of two components - `C1d` and `C2d` - that have no communication with any other components. This is an example of the “*No communication*” pattern.

This work aims to investigate the efficiency of dynamic component placement compared to static placement according to interaction pattern. Naturally, using only amount of communication with other threads as the only way parameter for deciding which cores a particular thread should be pinned to may not be efficient enough. One may think of multiple other characteristics: type of computations performed in the component, estimated execution time etc.

8 IMPLEMENTATION

This chapter first describes how Insense was adjusted to run on Unix-based systems that operate with multiple cores. Then, a description of the main work performed on implementing schemes for memory management and thread placement is described.



```
type IHello is interface (out string output)
type IPrinter is interface (in string input)

component Hello presents IHello {
    msg = "Hello World\n"
    constructor() { }
    behaviour {
        send msg on output
    }
}

component Printer presents IPrinter {
    constructor() { }
    behaviour {
        receive msg from input
        printString(msg)
    }
}

hello = new Hello()
printer = new Printer()
connect hello.output to printer.input
```

Figure 8.1 The “Hello” program written in Insense.

Figure 8.1 presents the source code of a single example program written in Insense that will be used throughout the chapter. It will be used at various stages of explanation to showcase different aspects of the implementation of multi-core Insense. Essentially, it is a “Hello world“ example. The program has two components: “Hello” and “Printer”. The “Hello” component contains a string object, it sends the object over a channel to the “Printer” component, where the string is outputted on the screen.

8.1 Base Unix Implementation

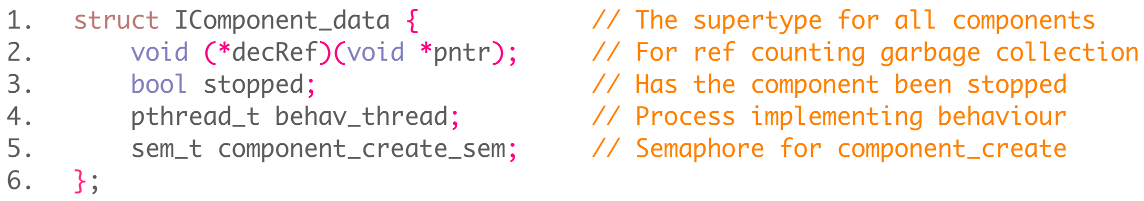
When the project was started, modifying Insense to an extent that allows running a demo application on a Unix-based system was set to be a minor step that needs to be taken in the beginning. While working on porting the language to Unix it was learnt that time commitment and amount of work had been largely underestimated. Every step that had to be taken to modify Insense came with issues that took time and effort that had not been previously expected. Working on this stage took more than two months. Problems that were met on this phase of the project could be explained by the fact that development was based around using two relatively large projects (Insense compiler and Insense runtime environment) that in total are comprised of more than 1,700 files written in three languages: C, Java and Insense itself.

Additionally, the code is not well-documented. While working on providing support for executing Insense applications on multi-core Unix-based machines, a better level of understanding was achieved in how pieces of the whole system behind Insense fit together and what principles of software engineering were used in development of Insense language.

8.1.1 Components

Code related to Insense components may be found in `component.c` and `component.h` files.

The main difference between components in the form they are used in the multi-core implementation of Insense and variation of Insense designed for InceOS and Contiki resides in the fact that components are run on POSIX-type threads.



```
struct IComponent_data {                // The supertype for all components
    void (*decRef)(void *pntr);         // For ref counting garbage collection
    bool stopped;                       // Has the component been stopped
    pthread_t behav_thread;             // Process implementing behaviour
    sem_t component_create_sem;         // Semaphore for component_create
};
```

Figure 8.2 The supertype structure for Insense components.

Every Insense component instance is described by a structure that stores various metadata about a component in question. The original design of the structure used for this task that was developed for the InceOS and Contiki operating systems had to be changed to accommodate for the multithreaded environment. In the current implementation, each Insense component is described by the structure of type `IComponent_data`, source code of which is given in Figure 8.2. The structure contains the following information about a component:

1. `void (*decRef)(void *pntr)` - pointer to a function used for reference counting garbage collecting described in Section 8.1.5.
2. `bool stopped` - Boolean flag indicating whether a component can run (set to `FALSE`) or if it has been stopped (set to `TRUE`).
3. `pthread_t behav_thread` - This is a handle to a POSIX thread implementing the component instance's behaviour.
4. `sem_t component_create_sem` - a semaphore used to avoid synchronisation issues that arise in the component creation stage where more than one component use private heaps for memory management.

To create a component, one needs to use the function void *component_create(behaviour_ft behaviour, int struct_size, int stack_size, int argc, void *argv[]). Appendix 4 contains source code of the function. The initial part of the function creates a structure *this_ptr of type IComponent_data that is used for storing metadata described in the beginning of this section. Figure 8.3 shows that a mutex was added to the code of the component_create() function, it is required to avoid problems with scheduling of threads.

```
pthread_mutex_lock(&thread_lock);
pthread_create(&this_ptr->behav_thread, NULL, startRoutine, argStruct);
pthread_mutex_unlock(&thread_lock);
```

Figure 8.3 A part of component_create() where component is assigned to a POSIX thread

Once the this_ptr structure for the component is initialised, the component's behaviour function is ready to be run with the POSIX thread. This operation is achieved through calling the pthread_create function ("*pthread_create(3)*" 2013). This function allows passing only one parameter along with the function that is to be run inside of the thread. The component's this structure, the pointer to the Insense constructor function and its arguments are placed into a struct and passed to the start_routine function as a single argument (as required by pthread_create). The start_routine is then able to call the Insense constructor. A wrapper function that "wraps" three objects into one structure that can then be passed to the pthread_create had to be designed. Figure 8.4 illustrated the source code.

```
void * startRoutine(void *args_p) {
    pthread_mutex_lock(&thread_lock);// Avoid problems with scheduling
    struct argStructType *args = (struct argStructType *) args_p;
    args->behaviour(args->this_ptr, args->argc, args->argv);
    free(args_p);
    pthread_mutex_unlock(&thread_lock); // Unlock mutex
    return ((void *) 1);
}
```

Figure 8.4 The wrapper function used to pass three parameters to the function run inside of a POSIX thread.

Figure 8.5 shows the function void component_stop(void * this_ptr). It is used for stopping components. This action is possible by calling stop inside of a component to stop this component or stop(sender) to stop a component instanced sender.

```
void component_stop(void * this_ptr) {
    struct IComponent_data *t = (struct IComponent_data*) this_ptr;
    t->stopped = 1;
}
```

Figure 8.5 The function for stopping Insense components.

Let us take a look at now familiar Insense program “Hello” as an example of how components are handled by the Insense runtime. The program was extended and two calls to “`stop`” were added, for the purposes of demonstration.

```
type IHello is interface (out string output)
type IPrinter is interface (in string input)
component Hello presents IHello {
    msg = "Hello World\n"
    constructor() { }
    behaviour {
        send msg on output
        stop
    }
}
component Printer presents IPrinter {
    msg = ""
    constructor() { }
    behaviour {
        receive msg from input
        printString(msg)
        stop
    }
}
hello = new Hello()
printer = new Printer()
connect hello.output to printer.input
```

(a)

```
1.component_create for Hello
2.component_create for Printer

3.component_stop for Hello, after
1 run of behaviour
4.component_stop for Printer,
after 1 run of behaviour
```

(b)

Figure 8.6 (a): “Hello” program that illustrates management of components. **(b)**: timeline of calls of functions dealing with management of Insense components during execution of the “Hello” program.

Figure 8.6 shows the program along with a timeline of calls to functions that create and stop components, which are outlined above. Each use of the `new` keyword (e.g. `hello = new`

Hello()) in Insense code, once compiled and run, results in a call to the component_create function.

As stated above, the stop keyword is called twice in the program, each call is done in the end of the first run through the behaviour function. These calls result in the function component_stop being called, which terminate execution of components.

8.1.2 Arrays

No major changes had to be made to the compiler or runtime in order to get arrays to work efficiently on Unix except for changes made to the following function array_loc in IArray.c inside of Insense runtime (Figure 8.7).

```
1.  void *array_loc(IArrayPNTR a, unsigned i){
2.      if (i >= a->length){
3.          i = 0; // do safe deref of array element 0 which is always created
4.      }
5.      return ((void *) (((char *) a->data) + i * a->type_size));
6.  }
```

Figure 8.7 Adapted to multi-core architecture array_loc function.

To support understanding of source code above, we present a description of the mechanism behind instantiating and accessing arrays in Insense. Insense arrays are instantiated by the keyword new (e.g. a = new integer[5] of 0 where 5 indicates size of the array and 0 shows which value is given to all members of the array during instantiation). Elements of the array can be accessed by following a common pattern of using square brackets (e.g. a[1]). A value of an element of the array can be overwritten by using the ":=" operator (e.g. a[1] := 3). In the last example, array_loc function would be used in the runtime to access array element 1 to set it to the value 3. It returns a pointer to array element a[i] (line 5 in Figure 8.7) or a predefined default location if array index out of bounds (lines 2 - 4).

During performing tests described in Chapter 9 it was discovered that the exception handling mechanism in array_loc function is extremely inefficient for multi-core use as it accesses a global variable to indicate success of its operation. Removing this global variable improved performance on multi-core systems in situations when arrays are used. The runtime and the compiler had to be adapted.

8.1.3 Channels

Code related to channels may be found in `channel.c` and `channel.h` files.

An existing channel design and implementation provided by Andrew Bell was used in this project and explained in (Bell 2013), but an outline of the algorithms and data structures is nevertheless provided in this document for the reader's convenience.

All channels are described by the structure shown in Figure 8.8. As with components, channels are garbage collected and the `decRef` element is a pointer to the garbage collection function for channel objects. Channels have a direction, which is defined by `chan_dir direction`. The `chan_dir` enumeration can take values of `CHAN_IN` (for an incoming channel from which data may be received) or `CHAN_OUT` (for an outgoing channel over which data can be sent). A channel is assigned with a list of other channels, which it is connected to, the list is defined by `List_PNTR connections`. A number of semaphores and a mutex were added to each channel to support scheduling, their usage is described by comments in source code. Other elements of the structure can be considered as self-explanatory.

```c
struct Channel {
    void (*decRef)(Channel_PNTR pntr); // GC decRef
    chan_dir direction;    // for error checking in bind, etc.
    size_t typesize;       // how large the buffer is
    void* buffer;          // pointer to data to send/receive
    bool ready;            // ready flag
    bool nd_received;      // used by select
    List_PNTR connections; // channels we're connected to
    sem_t conns_sem;       // connections available mutex
    pthread_mutex_t mutex; // for locking the channel
    sem_t blocked;         // block component if waiting for other channel
    sem_t actually_received; // make sure data can't be changed until after a receive has completed
};
```

Figure 8.8 Structure used to describe Insense channels on Unix.

The most critical aspect of ensuring correct work of channels is avoiding problems with synchronisation, as shown in (Bell 2013). In order to support sending and receiving data between channels and their synchronous work, three semaphores and one mutex were used in the project.

Six main operations involving channels are supported by the system. They are outlined in the remaining part of this section.

Creation of channels: supported by the function `Channel_PNTR channel_create(chan_dir direction, int typesize, bool contains_pointers)`. As Figure 8.9 shows, when a new channel is created in this function, information about the channel is initialised through assigning appropriate values to elements of the `Channel` structure described in the Figure 8.8.

```
Channel_PNTR channel_create(chan_dir direction, int typesize, bool contains_pointers) {
    Channel_PNTR this = (Channel_PNTR)DAL_alloc(sizeof(struct Channel), true);
    if(this == (void*) 0){
        DAL_error(CHAN_OUT_OF_MEMORY_ERROR);
        return NULL;
    }

    this->decRef = Channel_decRef;
    this->direction = direction;
    this->typesize = typesize;
    this->ready = false;
    this->nd_received = false;
    DAL_assign(&(this->connections), Construct_List()); //List of connections

    // Initialise mutexes and semaphores
    sem_init(&(this->conns_sem), 0, 0);
    pthread_mutex_init(&(this->mutex), NULL);
    sem_init(&(this->blocked), 0, 0);
    sem_init(&(this->actually_received), 0, 0);

    return(this);
}
```

Figure 8.9 Function responsible for creation of half-channels.

Binding channels: supported by `bool channel_bind(Channel_PNTR id1, Channel_PNTR id2)`. This function connects two components with one channel. The source code of the function in Figure 8.10 ensures that such properties of channels as their directions and types of data that they carry are checked at the first stage of the logic behind this function. If their directions are equal (both have `CHAN_IN` or `CHAN_OUT` assigned to their `chan_dir direction` attributes) or if they are meant to be medium for data of different type (values of `size_t typesize` are not the same), it serves as an indication of a logical flaw, and the binding process is aborted. The other reason why the function can be aborted is if half-channels that are supposed to be connected already have each other in their `connections` lists: in this case further execution of the function is also cancelled.

```c
bool channel_bind(Channel_PNTR id1, Channel_PNTR id2) {
    pthread_mutex_lock(&conn_op_mutex);

    // Check not both CHAN_IN or CHAN_OUT
    if(id1->direction == id2->direction) {
        pthread_mutex_unlock(&conn_op_mutex);
        return false;
    }
    if (id1->typesize != id2->typesize) {
        pthread_mutex_unlock(&conn_op_mutex);
        return false;
    }

    pthread_mutex_lock(id1->direction == CHAN_IN ? &(id1->mutex) : &(id2->mutex));
    pthread_mutex_lock(id1->direction == CHAN_IN ? &(id2->mutex) : &(id1->mutex));

    // Check if channels are not already connected. Assuming bind always adds to both channels' lists, we only need to check one channel for the other
    if(containsElement(id1->connections, (void*)id2)) {
        pthread_mutex_unlock(&conn_op_mutex);
        return false;
    }

    // Add to connection lists
    insertElement(id1->connections, id2);
    insertElement(id2->connections, id1);

    // Unlock conns mutex in both channels
    int val = 0;

    sem_getvalue(&(id1->conns_sem), &val);
    // Never allow semaphore to go above 1; make it act like a mutex
    if(val == 0) {
        sem_post(&(id1->conns_sem));
    }

    sem_getvalue(&(id2->conns_sem), &val);
    if(val == 0) {
        sem_post(&(id2->conns_sem));
    }

    pthread_mutex_unlock(&(id1->mutex));
    pthread_mutex_unlock(&(id2->mutex));
    pthread_mutex_unlock(&conn_op_mutex);
    return true;
}
```

Figure 8.10 Function responsible for binding two half-channels into one complete channel.

If conditions outlined above do not hold true, the binding process can be initialised. The process behind this action is not complicated. To bind two half-channels into one complete channel, add the half-channel on the opposite side of the channel to the `connections` list. This procedure is protected by mutexes which are blocked before and unlocked after performing manipulations on the lists of connections.

Unbinding channels: supported by `void channel_unbind(Channel_PNTR id)`. The logic behind this function is similar to that of the function responsible for binding channels. The following action is performed for two half-channels that need to be unbound: a half-channel on the other side of the channel is removed from the `connections` list. This operation is performed for both half-channels and for all connections found in the `connections` list. As for the binding process, this action is protected by two mutexes and two semaphores.

Sending data over channels: supported by `int channel_send(Channel_PNTR id, void *data, void *ex_handler)`. This function is an implementation of the algorithm that was described in Figure 7.1 in Section 7.1.2.

Receiving data from channels: supported by `int channel_receive(Channel_PNTR id, void *data, bool in_ack_after)`. As with the `send` operation, this function is also based on the corresponding algorithm that was described in Figure 7.1.

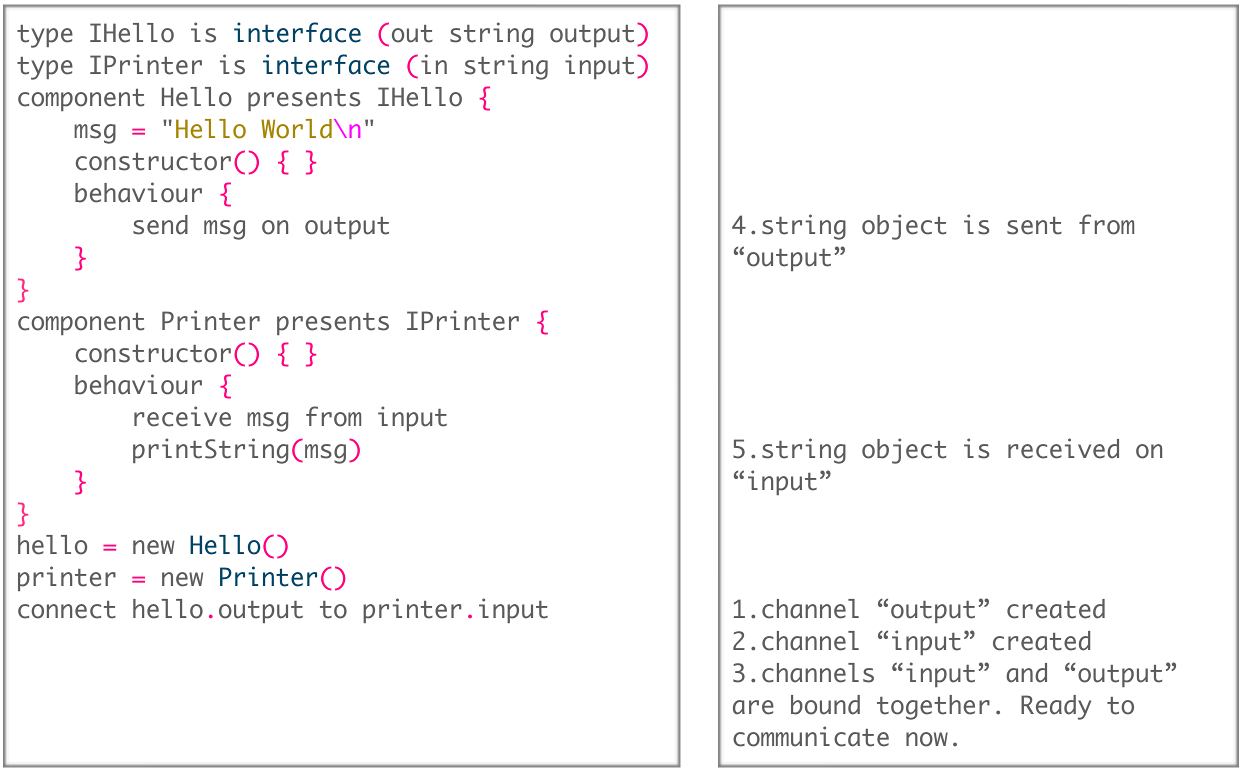
Destruction of channels (garbage collection): supported by `void Channel_decRef(Channel_PNTR this)`. Figure 8.11 shows the source code of the function. The channel is unbound from all of its connections. Then, the list of connections is garbage collected and mutexes and semaphores are destroyed.

```
void Channel_decRef(Channel_PNTR this){
    channel_unbind(this);           // Disconnect from all other channels
    DAL_decRef(this->connections);  // Garbage collect connections list
    // Now destroy mutexes and semaphores
    sem_destroy(&(this->conns_sem));
    pthread_mutex_destroy(&(this->mutex));
    sem_destroy(&(this->blocked));
    sem_destroy(&(this->actually_received));
}
```

Figure 8.11 Function responsible for destroying channels.

Let us take the "Hello" example into consideration again. Figure 8.12 describes what functions are called by the compiled executable when channels are used in components. Channels (half-channels) are created by calling the `channel_create` function when the `new` keyword is used to create a new component (that contains a description of a half-channel in its interface type, e.g. `type IHello is interface (out string output)`). Then, two half-channels are bound together by using the `connect` keyword.

Sending data is possible by utilising the `send` keyword inside of the `behaviour` function. Naturally, the name of the half-channel needs to be correct; in case of the example, it is `output`. Receiving data follows similar logic: `receive` keyword called from the `behaviour` function results in information being sent over the channel.



```
type IHello is interface (out string output)
type IPrinter is interface (in string input)
component Hello presents IHello {
    msg = "Hello World\n"
    constructor() { }
    behaviour {
        send msg on output
    }
}
component Printer presents IPrinter {
    constructor() { }
    behaviour {
        receive msg from input
        printString(msg)
    }
}
hello = new Hello()
printer = new Printer()
connect hello.output to printer.input
```

```






4.string object is sent from
"output"






5.string object is received on
"input"




1.channel "output" created
2.channel "input" created
3.channels "input" and "output"
are bound together. Ready to
communicate now.
```

(a) (b)

Figure 8.12 (a): "Hello" program that illustrates channels. **(b)**: timeline of calls of functions dealing with channels during execution of the "Hello" program.

Providing support for channels was crucial for the project since it offers possibilities to extend complexity of tests that can be run for comparing performance of private heaps / shared heap memory management schemes and static / dynamic thread placement methods.

8.1.4 Program Entry Point

Code related to the program entry point may be found in the file runtime_main.c.

The system-level entry-point to the Insense program is executed as the main thread and it calls the programmer-defined entry-point to the Insense program, which is defined by the code sequence following all component declaration.

```
1.  int main() {
2.  #if HEAPS // Private Heaps
3.     if (pthread_mutex_init(&thread_lock, NULL) != 0) { // Initialize mutex
4.          return NULL;
5.     }
6.  #else // Shared Heaps
7.     if (pthread_mutex_init(&alloc_lock, NULL) != 0) { // Initialize mutex
8.          return NULL;
9.     }
10. #endif
11.
12.     mainThread = pthread_self(); // Note the ID of the main thread.
13.     // Create a list for storing references to p-threads
14.     threadList = listCreate();
15.     // Create map used to store memory locations of private heaps
16.     SHList = listCreate();
17.     // Create map used to store locations of what is allocated using malloc()
18.     mallocList = listCreate();
19.
20.     primordial_main(NULL); // Call primordial_main.
21.
22.     if (threadList != NULL ) {
23.         listJoinThreads(threadList); // Join all p-threads
24.     }
25.
26.     // Destroy lists, mutexes and free memory
27.     listDestroy(threadList);
28.     listDestroy(SHList);
29.     listDestroy(mallocList);
30.     pthread_mutex_destroy(&thread_lock);
31.     pthread_mutex_destroy(&alloc_lock);
32.     return 1;
33. }
```

Figure 8.13 The main function in Insense runtime.

As all program written in C, the Insense runtime has the main function, which serves as a point of entry to the runtime. Figure 8.13 shows the source code of it. Let us discuss this function in detail. The first stage in the function is initialisation of global variables and data structures. Lines 2 - 10 handle initialisation of mutexes used to avoid problems with 1. scheduling that occurs during the component creation stage and 2. allocation and deallocation of memory into the shared heap (see Section 8.1.1 for more details). Code in line 12 initilises a global variable that stores ID of the main thread. Lines 13 - 18 are responsible for creating global thread-safe linked lists that store records of: POSIX threads, private heaps, and memory allocated with malloc.

```
void primordial_main( void *this ) {
    HelloPNTR hello_glob = NULL;
    DAL_assign(&hello_glob , component_create( Construct_Hello0,
sizeof( HelloStruct ) , 46, 0, NULL ) );
    component_yield( ) ;
    PrinterPNTR printer_glob = NULL;
    DAL_assign(&printer_glob , component_create( Construct_Printer0,
sizeof( PrinterStruct ) , 46, 0, NULL ) );
    component_yield( ) ;
    channel_bind( hello_glob->output_comp,printer_glob->input_comp ) ;
    component_exit( ) ; // For compatibility with InceOS.
}
```

Figure 8.14 The primodial_main function generated for the “Hello” Insense runtime.

Figure 8.14 shows the primordial_main function generated for the “Hello” test program. This function is executed from the main function. It is the programmer-defined entry point to the Insense program. One may see from the example above that components are created in this function, in the case of this example, two components are created, namely, Hello and Printer. Additionally, the only channel between these components that is described in the program is bound in this function. Calls of component_yield do nothing. A compiler could have been modified to remove these calls, but it was decided to keep the compiler as similar as possible to the Insense compiler for InceOS.

After execution of the primodial_main function, the main thread waits for the POSIX-type threads defined for Insense components by the use of pthread_join function. Figure 8.13 demonstrates this in lines of code 22 - 24. Lines 26 - 32 are for destroying no longer needed objects, mutexes and freeing memory.

8.1.5 Memory Management & Garbage Collection

The original version of Insense that was used with InceOS and Contiki works on 16-bit devices. As Figure 8.15 shows, the memory header had to be changed to cater for the 64-bit architecture used in machines in the lab used to conduct testing. Size of the ref_count, which corresponds to a number of references of the object stored in the allocated piece of memory that follows the header, in the modified version is 63 bits.

```
typedef struct MemHeader {
    unsigned long ref_count :63; // 63 bits used in 64-bit architecture.
    unsigned short mem_contains_pointers :1;
}*MemHeader, MemHeaderStruct;
```

Figure 8.15 Insense memory header structure adjusted for the 64-bit architecture.

Otherwise, all remaining aspects of the garbage collection scheme used in Insense were not changed and they were taken from the versions of Insense deployed for InceOS and Contiki.

```
type IHello is interface (out string output)
type IPrinter is interface (in string input)
component Hello presents IHello {
    msg = "Hello World\n"
    constructor() { }
    behaviour {
        send msg on output
    }
}
component Printer presents IPrinter {
    constructor() { }
    behaviour {
        receive msg from input
        printString(msg)
    }
}
hello = new Hello()
printer = new Printer()
connect hello.output to printer.input
```

(a)

```
1. New string object is created.
Reference count = 1.

2. A copy of a string object is
constructed prior to send. The
reference count is incremented
incremented to '1' on the sender
side and then sent over the
channel. Action is repeated on
each iteration of the behaviour.
3. A copy of a string object is
received from input. The reference
count is decremented to '0' on the
receiver side at the end of the
behaviour loop thereby forcing the
string object to be garbage
collected.
```

(b)

Figure 8.16 (a): “Hello” program that illustrates memory management and garbage collection. **(b)**: descriptions of behaviour of the garbage collector.

To further explain principles behind work of garbage collection in Insense, let us again take the “Hello” program as an example. Figure 8.16 outlines a sequence of actions that are undertaken by the garbage collector to transfer a string object between two components. Each time an object of any type needs to be passed between components via a channel, instead of moving the object from one component to another, the copy of it is transferred through the channel. In the case of the “Hello” program, the `Hello` component sends a string object `msg` to the `Printer` component. Step 1 indicates a point in time when memory for the string object is allocated. The reference count at this stage is 1. Step 2 in the timeline shows that instead of physically moving the string and changing its memory location, a copy of the object is created. Also, the reference count is incremented: at this stage two copies of the string exist in the program each with a reference count of 1. The copy of the string is sent over the channel rather than the original. Then, in step 3, the object is received by the `Printer` component. The reference count of the received object copy is then decremented and set back to 0 at the end of the receiver's behaviour loop forcing the copy of the string to be garbage collected. These actions are repeated on each iteration of the behaviour. The string object itself is garbage-collected on the exit from the program.

8.1.6 Thread-safe List

Code related to Insense components may be found in `ThreadSafeList.c` and `ThreadSafeList.h` files.

This project involves concurrent execution of multiple POSIX threads. Any data structures that are used by the runtime environment and the compiler consequently need to be thread-safe. On multiple occasions a linked list had to be utilised, namely for: storing a list of POSIX threads running in the program, keeping track of private heaps used by threads, and for noting what memory is allocated by calls of the `malloc` function. A custom-built thread-safe list was developed as a part of this project to handle aforementioned tasks. Its implementation is trivial. Figure 8.16 shows that the structure which is used to describe the list contains one mutex that is utilised to avoid problems caused by concurrency issues.

```c
struct threadSafeList {
    int count;              // Number of elements in the list.
    struct listItem *head; // First element in the list.
    struct listItem *tail; // Last element in the list.
    pthread_mutex_t mutex; // Mutex for locking functions. Makes it thread-safe.
};
```

Figure 8.17 Structure that describes the thread-safe linked list.

All elements of the list contain a their value in `void *value`, pointers to the previous element `prev` and to the next element `next`. The first element in the list has its `prev` attribute set to `NULL`, the `next` attribute of the last element equals `NULL` as well.

```
struct listItem * listAdd(struct threadSafeList *l, void *content) {
    struct listItem *listItem;
    pthread_mutex_lock(&(l->mutex)); // Lock mutex.
    // Allocate memory.
    listItem = (struct listItem *) malloc(sizeof(struct listItem));
    listItem->value = content;
    listItem->next = NULL;
    listItem->prev = l->tail;
    // Handle tail and head.
    if (l->tail == NULL ) {
        l->head = l->tail = listItem;
    } else {
        l->tail->next = listItem;
        l->tail = listItem;
    }
    l->count++; // Increase count of objects in the list.
    pthread_mutex_unlock(&(l->mutex));     // Unlock mutex.
    return listItem;
}
```

Figure 8.18 Thread-safe "add" operation in the linked list.

Figure 8.18 shows an example of a thread-safe implementation of the "add" function. The mutex is locked in the beginning of the function and unlocked in the end of it. Otherwise, implementation of this function and its behaviour follows conventions.

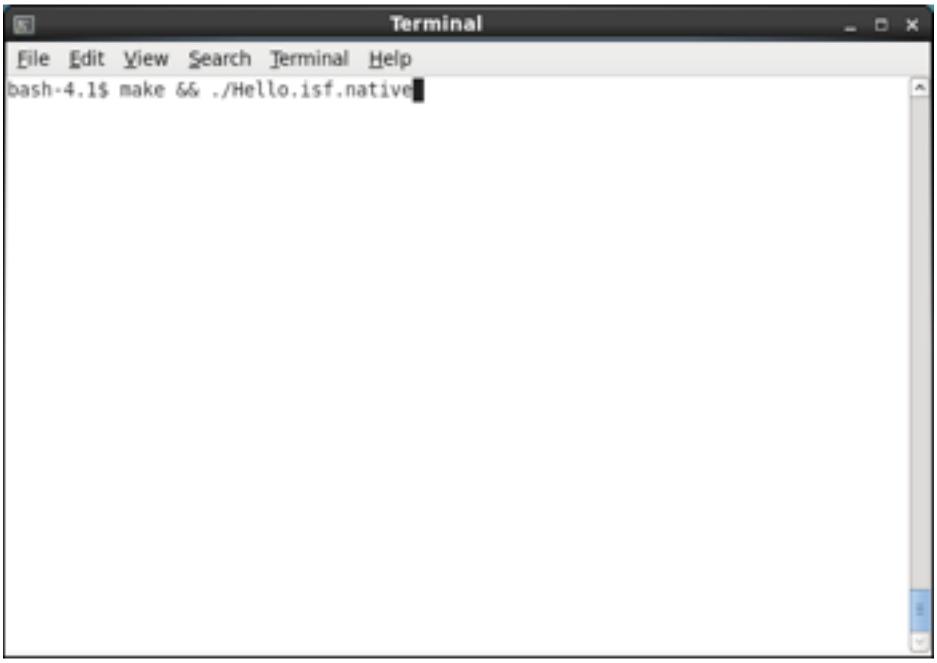
8.1.7 Build System

The Insense compiler is written in Java, which generates C code. Generated code can then be compiled by using `gcc` and is linked with the runtime library which is also written in C.

Insense programs are built into runnable files in a series of stages:

1. Compilation is initialised. C files are compiled along with a `makefile`, which allows to type in the `make` command in the Terminal on a Unix-based machine and create an executable. Figure 8.19 demonstrates how this operation can be performed in Scientific Linux. Also, an address of Insense runtime needs to be defined in order to make Insense programs. It is achieved by the following terminal command: `export INSENSE_RUNTIME_INCEOS="/Address_to_runtime/"`.
2. An executable file has now been created. Insense program can be executed. Figure 8.19 demonstrates how the program can be run on Scientific Linux (after the and "`&&`" symbol,

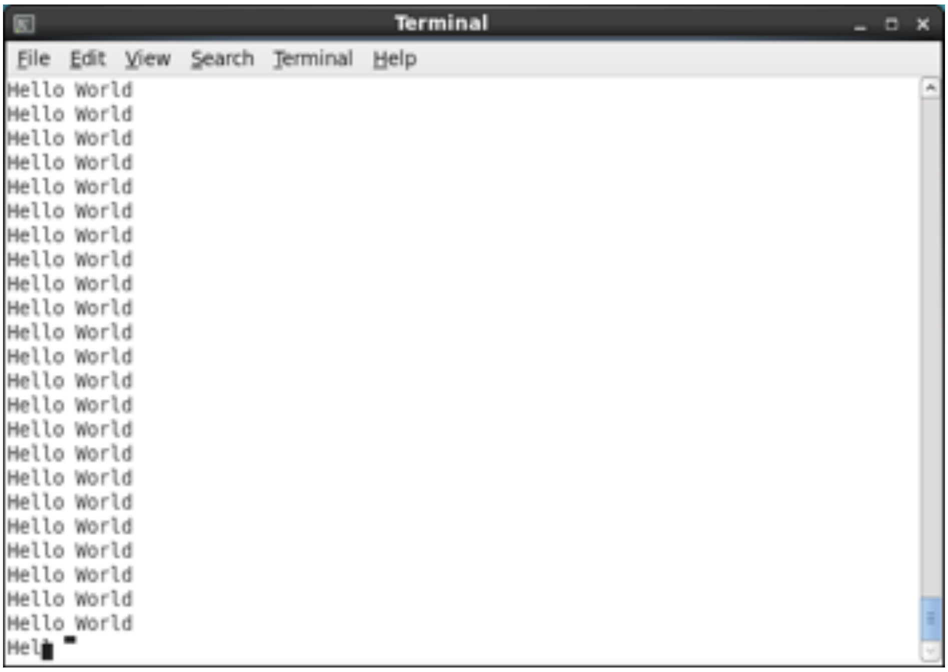
running a compiled C program on Linux is done by using the "`./NAME_OF_THE_EXECUTABLE`" operation).



```
bash-4.1$ make && ./Hello.isf.native
```

Figure 8.19 Command to "make" and run the "Hello" Insense program on Linux.

Figure 8.20 shows an excerpt from the output of the "Hello" program. A string "Hello World" is received through a channel and outputted onto the screen on each iteration of the `behaviour` function of the `Printer` component.



```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hel
```

Figure 8.20 Output of the “Hello” program after successful compilation and execution.

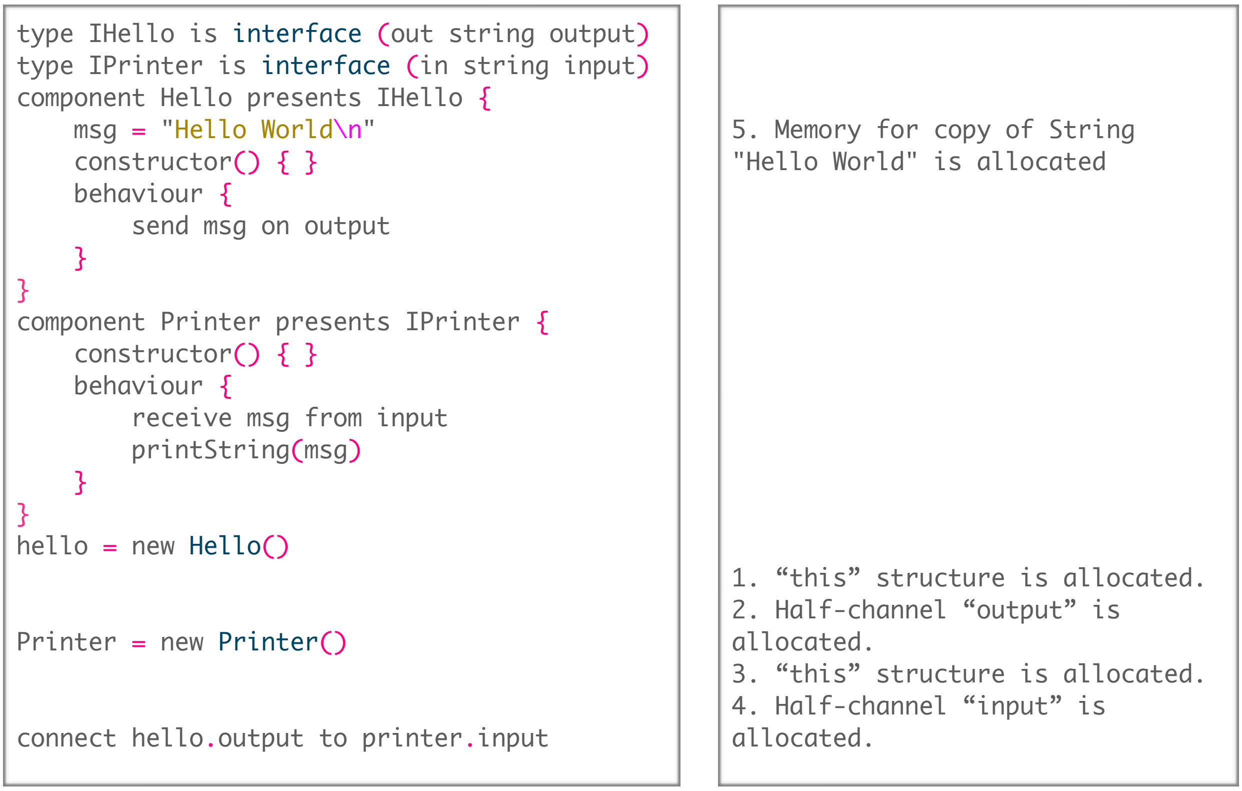
By default, `behaviour` functions of Insense components are executed indefinitely. A user may either modify source code so the component stops after a certain amount of time/number of executions of `behaviour` or force to stop execution by pressing `ctrl+c` in the terminal, from which the program is run.

8.2 Data Placement Schemes

Code related to data placement may be found in files `DAL_mem.h`, `DAL_mem_common.c`, `DAL_BH_mem_common.c`, and `DAL_SH_mem_common.c`.

Successfully reaching a point where a simple Insense program could be run on Unix provided a base implementation for the project. The main part of work was centered around experimenting with memory placement and thread affinity schemes described in Chapter 7. Working on this stage of the project took about three months.

Two memory placement schemes were implemented in the scope of this work: allocation of memory into one shared heap and into multiple private heaps.



```
type IHello is interface (out string output)
type IPrinter is interface (in string input)
component Hello presents IHello {
    msg = "Hello World\n"
    constructor() { }
    behaviour {
        send msg on output
    }
}
component Printer presents IPrinter {
    constructor() { }
    behaviour {
        receive msg from input
        printString(msg)
    }
}
hello = new Hello()


Printer = new Printer()


connect hello.output to printer.input
```

```



5. Memory for copy of String
"Hello World" is allocated













1. “this” structure is allocated.
2. Half-channel “output” is
allocated.
3. “this” structure is allocated.
4. Half-channel “input” is
allocated.
```

(a) (b)

Figure 8.21 (a): "Hello" program that illustrates allocation of memory with the shared heap. **(b)**: timeline of allocation of data.

The "Hello" test program is also utilised here to show allocation of data. Figure 8.21 outlines a timeline of allocations of memory as requested by the program. This example will be used to demonstrate a sequence of function calls that take place when new Insense components and channels are created. One may see that the first two allocations of memory take place when Insense components are created with the `new` keyword. This is the first allocation of memory that takes place when an Insense component is created. The `this` structure (described in details in sections 7.1.1 and 8.1.1) is allocated on the shared or private heap, depending on the active data placement scheme. Allocation of space needed for `this` is handled inside of the `component_create` function described in Section 8.1.1.

```
1.   void Hello_init_globals(HelloPNTR this) {
2.       this->decRef = decRef_Hello;
3.       this->output_comp = channel_create(CHAN_OUT, sizeof(StringPNTR), true) ;
4.       DAL_assign(&this->msg_comp, Construct_String0("Hello World\n")) ;
5.   }
```

Figure 8.22 Compiled code of the "Hello" example program dealing with allocation of memory for a channel and a local variable.

The previous paragraph focused on allocation of memory for a newly created component. After that process has been completed, channels are put on the heap by calling the `channel_create` function (described in Section 8.1.2). In case of this example, channels `output` (in the `Hello` component, line 3 in Figure 8.22) and `input` (in the `Printer` component) are assigned with space in main memory. Finally, space required for component variables is allocated. Figure 8.22 demonstrates the `void Hello_init_globals(HelloPNTR this)` function taken from the compiled code of the "Hello" program. It shows that the "Hello" program asks for memory for a string "Hello World" created inside of the "`Hello`" component, line 4. It can be noticed that allocation of memory required for channels and component variables is handled by calls of functions `component_create`, `channel_create`, and `DAL_assign` (which is described in Section 8.2.2). Allocation of memory required for components is handled by the thread that serves as the entry point to the program (sections 7.1.3 and 8.1.4 focus on it), but allocation of memory for channels and component variables takes place from the actual POSIX-type thread assigned to the component.

```
void *DAL_alloc(size_t size, bool mem_contains_pointers) {
    void* pntr = NULL;
#if HEAPS // Private Heaps
       // 2 is passed to indicate that a new private heap needs to be created.
    if(mem_contains_pointers == 2) {
        pntr = (void *) SH_create_small_heap(); // Create a new private heap.
    } else {
        // Allocate into an existing private heap.
        pntr = (void *) SH_alloc(size + sizeof(MemHeaderStruct) );    }
#else // Shared Heap
    // Allocate memory into the shared heap.
    pntr = (void *) BASE_mem_alloc( size + sizeof(MemHeaderStruct) );
    // Note down that memory was allocated with malloc().
    listAdd(mallocList, pntr);
    // Log into a file, if required.
    log_into_file("malloc     at", pntr, size + sizeof(MemHeaderStruct));
#endif

#if DEBUG == 1
  if(pntr > last_max_malloc){
      last_max_malloc = pntr;
      printf("\nm(%u): %p", size, pntr);
  }
#endif
  if(pntr == NULL){
      printf("DAL_alloc pointer is NULL.\n");
    DAL_error(OUT_OF_MEMORY_ERROR);
    return NULL;
  }

#if DEBUG == 2
  printf("\nm(%u): %p", size, pntr);
#endif
  // Zero memory area to avoid having to set all pointer types to NULL prior to DAL_assign
  memset(pntr, 0, (size + sizeof(MemHeaderStruct)) );
  ((MemHeader) pntr)->ref_count = 0;
  ((MemHeader) pntr)->mem_contains_pointers = mem_contains_pointers;
  if (mem_contains_pointers == 2)
       ((MemHeader) pntr)->mem_contains_pointers = true;
  return ((pntr + sizeof(MemHeaderStruct)));
}
```

Figure 8.23 Modified and extended function for dynamically allocating memory in Insense.

Further, Figure 8.23 shows a version of the function void `*DAL_alloc(size_t size, bool mem_contains_pointers)` that was modified to accommodate for both the "shared heap" and "private heaps" schemes. This function serves as an entry point to dynamic allocation of memory in all Insense programs. It is called whenever dynamic allocation of memory is required in runtime and in generated C code. The C preprocessor conditions (e.g. “#if HEAPS”) were used to switch between using the shared heap and private heaps for memory management. There are two possible scenarios of working with private heaps that are considered by the function DAL_alloc. Firstly, at the component creation stage, when the function is called by a thread that has not been linked to a private heap (line 3 in Figure 8.3), a new private heap is created and gets linked to that POSIX thread. Instead of altering the InceOS Insense compiler in multiple places, a single alteration was made to pass special flag “2” as value for the argument bool mem_contains_pointers to the function to indicate that a new private heap needs to be created: DAL_alloc(0, 2). Secondly, a more common scenario is when the function is called by a thread that has already been assigned a particular private heap in which any thread-local dynamic allocation and deallocation can take place. In this case, the function void * SH_alloc(unsigned size) is called and it allocates memory within the private heap associated with the calling thread.

8.2.1 Shared Heap

The key point for implementing this scheme was making sure that the appropriate dynamic memory allocator is called whenever Insense structures, arrays or channels are constructed at runtime. It was achieved by ensuring that the C malloc function (“*malloc(3)*” 2013) is called in all of the aforementioned cases. The malloc function requests allocation of a block of memory in the heap. In case of a successful request, the operating system reserves the requested amount of memory.

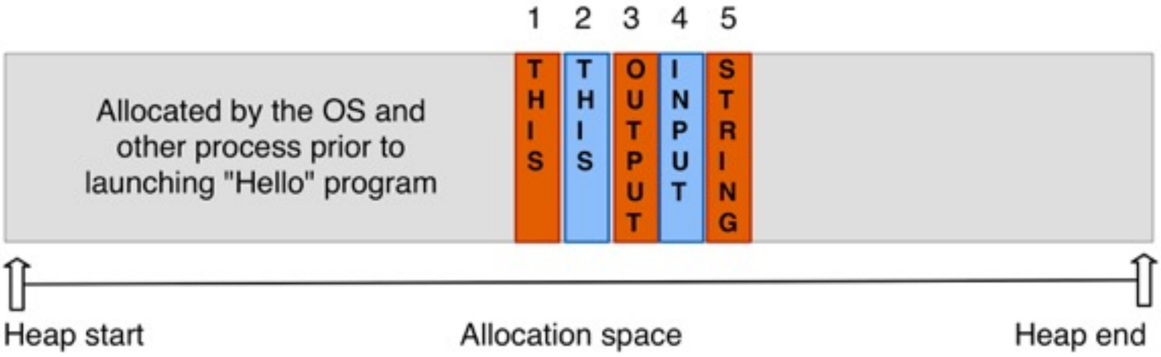

1
2
3
4
5
Allocated by the OS and other process prior to launching "Hello" program
THIS
THIS
OUTPUT
INPUT
STRING
Heap start
Allocation space
Heap end

Figure 8.24 Allocation of memory required by the “Hello” program in the shared heap.

Let us take the “Hello” example into consideration. Figure 8.21 shows at which stages of the program’s execution memory is allocated and for which data structures. Figure 8.24 describes how memory is allocated for two “this” structures, two channels, and one local variable. Red colour represents memory allocated for the `Hello` component, blocks coloured with blue indicate memory given to data that originates from the `Printer` component. Numbers above the allocation space indicate a sequence of allocations.

Figure 8.24 indicates the possibility that memory allocated for components is not contiguous, relative to other pieces of data allocated by the same component. In this test case, after allocation of memory for the `this` structure of the `Hello` component inside of the `component_create` function, the processor did not switch to the newly created POSIX thread that was running `behaviour` of `Hello` such that memory for the `this` structure of the `Printer` component, is still allocated from the entry point thread (Section 8.1.1 describes component creation). Similar behaviour can be seen when allocating memory for channels and the string object. Over time such behaviour resulting in memory allocated in an noncontiguous manner may become significantly more complicated when a larger number of Insense components are used.

8.2.2 Private Heaps

In this scheme allocation of memory is supported by multiple heaps, instead of a single large heap.

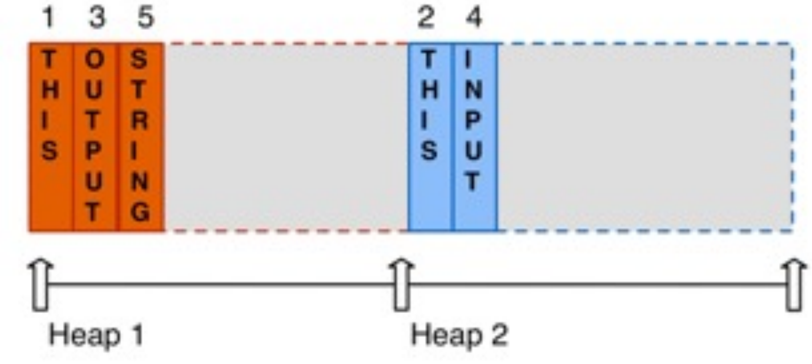


Figure 8.25 Allocation of memory required by the “Hello” program in two private heaps.

Let us again look at the “Hello” program as an example. Figure 8.21 shows at which stages of the program’s execution memory is allocated and for which data structures. Figure 8.25 describes how memory is allocated for two `this` structures, two channels, and one local variable, when private heaps are utilised. As before, red colour represents memory allocated for the Hello component, blue blocks correspond to the Printer component. Numbers above the allocation space indicate a sequence of allocations, sorted by time.

Compared to the "shared heap" allocation in 8.2.1 above, here memory allocated for components is contiguous, relative to other pieces of information allocated by the same component. Similar to the case with the shared heap, in this test case, after allocation of memory for the this structure of the Hello component inside of the component_create function, the processor switches back to the main thread and allocates memory for the this structure of the Printer component. Nevertheless, this time memory requested for the 2nd this structure is allocated on a separate, second, heap. Memory for channels and the string object is subsequently allocated. Memory allocated for a particular thread is always put into the private heap assigned to the thread (component) in question.

Also, dynamically allocated storage for data structures on a sender-side private heap is not copied to a receiving component's heap in this implementation. A receiver of an object will receive a reference to a copied object located in the sender-side heap. In contrast, data of type integers, real and of other basic types is sent by value, i.e. in this case no cross-references between heaps exist.

The rest of this section describes changes and additions to existing Insense runtime library code and newly created functions that support the “private heaps” memory allocation scheme.

8.2.2.1 Creation of a New Private Heap

Creation of private heaps is supported by the void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off) function (“*mmap*” 2004). This function offers a way to map data or files to memory. Most implementations of malloc use mmap for allocation. The main reason behind using mmap in case of Insense is that it allows contiguous allocation of data into a fixed region of memory protected from being used by the operating system and external processes.

```
void * SH_create_small_heap() {
    void * heap = mmap(NULL, heapSize, PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANON, -1, 0);
    if (heap == MAP_FAILED)
        PRINTFMC("MMAP failed");
    return heap;
}
```

Figure 8.26 Function for creation of a new private heap.

Figure 8.26 provides a listing of the function where creation of private heaps is implemented. `heapSize` is set statically to a default value. The following flags are utilised with the call of the `mmap` function :

1. `MAP_SHARED | MAP_ANON` - the `MAP_SHARED` flag indicates that changes made to memory after the `mmap()` call are visible in the mapped region. The `MAP_ANON` flag demonstrates that mapping is anonymous, meaning that no file is mapped into memory.
2. `PROT_READ | PROT_WRITE` - data inside of the mapped region can be read and overwritten. ("*mmap*" 2004)

8.2.2.2 Allocation and Deallocation of Memory within the Private Heaps

The C language does not offer standard tools that can be used for memory management in case of using multiple private heaps. Hence, functions that can support allocation and deallocation of memory within a private heap had to be developed. Extensive amount of research on the methods that could be used for supporting such tasks has already been conducted by the scientific community (Knuth 1997). Methods described by Knuth and Ervin were used as the underlying base for developing the functions that allocate and deallocate memory within a pre-allocated private heap. An existing implementation of a publicly available custom-built allocator and deallocator was taken as a base for functions implemented in this project ("*User mnicky*" 2013).

The solution presented in the project and described lower may not be the fastest possible approach. However, it is rather easy to understand and implement and the solution has a relatively low space complexity, which may be beneficial for improving results of experiments.

```
struct shMapType {
    pthread_t thread_id; // ID of the thread that is linked to the private heap
    unsigned int * memArea; // The actual memory range used for allocating memory within the private heap
    unsigned int memAreaSize; // Size of the private heap
    unsigned int available;   // How much memory is available for allocation
};
```

Figure 8.27 A structure that stores meta information for a new private heap.

Let us continue with the description of implementation of the system used for allocating and deallocating memory. The first step that needs to be done is setting up a structure that stores meta data about a newly initialised private heap. Figure 8.27 outlines the structure in the way it was implemented. The following information is initialised on this stage:

1. `pthread_t thread_id` - ID of the POSIX thread that is linked to the private heap.
2. `unsigned int * memArea` - pointer to the start of the memory range used for allocating memory within the private heap.
3. `unsigned int memAreaSize` - size of the private heap.
4. `unsigned int available` - amount of memory available for allocation.

```
struct shMapType * SH_init(void *ptr, pthread_t thread) {
    struct shMapType * shMapElement = malloc(sizeof(struct shMapType));
    shMapElement->mem = (MEMTYPE *) ptr;
    shMapElement->memSize = heapSize / sizeof(MEMTYPE);
    shMapElement->mem[0] = heapSize / sizeof(MEMTYPE) - 1;
    shMapElement->mem[shMapElement->memSize - 1] = heapSize /
sizeof(MEMTYPE);
    shMapElement->thread_id = thread;
    shMapElement->avail = 0;
    return shMapElement;
}
```

Figure 8.28 Initialisation of a structure that stores meta information for a new private heap.

Figure 8.28 shows a listing of code used to provide initialisation of aforementioned attributes that characterise a new private heap. The heap can be considered as initialised when: it is given a size, a thread/component is linked to the heap, and a starting location of the heap in main memory is know. All private heaps are given the same size. The current implementation of the project assumes this size to be 4,000,980B (defined by the `heapSize` variable previously mentioned in Figure 8.26). This number was achieved through performing experiments. It was concluded that this size is large enough to accommodate memory needs of components of different nature. In the implementation for this project which is tailored towards running specific experiments, heaps are not extendable, and the implementation does not contain a mechanism for estimating size that an Insense component would need.

After a new heap has been created and initialised, allocation of memory within that private heap becomes possible. Figure 8.29 outlines implementation of the function `void * SH_alloc(unsigned size)` that calls an appropriate allocator to allocate requested memory.

At its entry point the function attempts to retrieve information about the heap that is linked to the thread that the function was called from. It is achieved by calling the `listGetMemoryLocation` function which traverses the thread-safe linked list `SHList` in search of an entry that contains a reference to the calling thread. If the function fails to find such entry, `NULL` is returned.

```
void * SH_alloc(unsigned size) {
    // Return NULL pointer after attempt to allocate 0-length memory
    if (size == 0) {
        return NULL ;
    }
    struct shMapType * shMapEntry = listGetMemoryLocation(SHList, (unsigned)
pthread_self()); // Receive metadata about the current heap
    if (shMapEntry == NULL ) {
        if ((unsigned) mainThread == (unsigned) pthread_self()) {
            // If it is the main thread - malloc space
            void * result = (void *) malloc(size + sizeof(MemHeaderStruct));
            listAdd(mallocList, result);
            return result;
        }
        // If nothing can be done - return NULL
        return NULL ;
    } else {
        return SH_alloc_at_base(size, shMapEntry);
    }
}
```

Figure 8.29 Allocation of memory into a private heap assigned to the current POSIX thread.

The function was implemented in a way that allows splitting it into two sub-routines:

1. *Calling malloc when allocation is required by the main thread*: preceded by `NULL` being returned by the call of `listGetMemoryLocation`. Memory requested by the main thread is allocated using the `malloc` allocator, since, as was justified in Section 7.1.3, amount of space needed for the main thread is relatively small and location in memory of data allocated by the thread serving as the entry point has low impact on performance. The ID

of the main thread is established at runtime by code in the file `runtime_main.c` (described in more details in Section 8.1.4).

2. *Allocation into an existing private heap*: Alternatively, if the heap for the calling thread is found then `SH_alloc_at base` is used to allocate the requested amount of memory in that private heap.

As stated in point 2 above, allocation of memory within a specific private heap is performed by the function with a signature that has the following form: `void *SH_alloc_at_base (unsigned int size, struct shMapType * shMapEntry)`. The function was implemented to require storing as little amount of meta information as possible ("*User mnicky*" 2013). The first node of every allocated range stores an integer. This number identifies the number of allocated nodes that follow in the range in question.

Figure 8.30 presents a diagram that gives insight into this mechanism. In this figure white blocks indicate free space and red blocks - allocated memory. Three scenarios are taken into account: *A.* represents the initial situation where the whole allocation range contains only free memory blocks; *B.* shows the same allocation space, but with four allocated blocks of memory: blocks 0 - 3; *C.* outlines a scenario where three blocks have been allocated with data (blocks 3 - 5) and the memory allocated in scenario B has been deallocated.

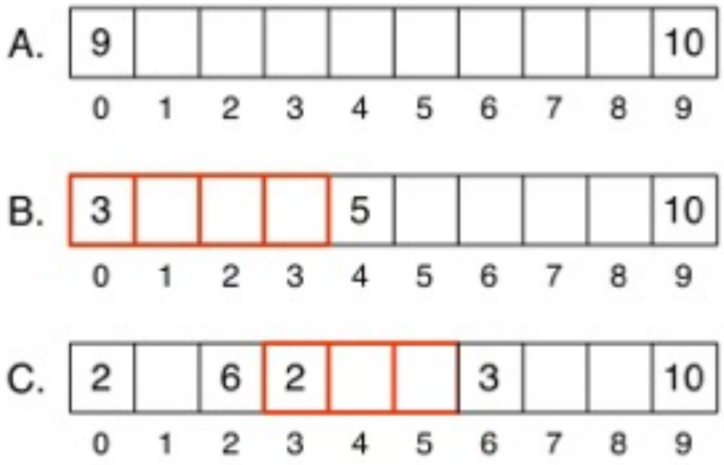


Figure 8.30 Allocation of memory with `SH_alloc_at_base`.

A similar approach to keeping track of allocated memory logic is also used for marking free space. A number of free blocks that the free space contains is stored in the first node of the free range, similarly to storing a number of allocated blocked in the first node of an allocated region. For example, in Figure 8.30, in situation *A,* the 0th block indicates that there are exactly nine free block that follow; in situation *B,* after allocating four blocks of memory, the 0th block now carries information about the allocated range, and the 4th block now contains "5", which indicates that the cell is followed by five free blocks ready to be allocated.

The last node of the free range points to the ID of the starting node of the next free range. Figure 8.30 gives an example of it: in situation *A,* the 9th block contains an integer 10 - it indicates that the next free block is outside of the allocation space in question. In situation *C,* the 2nd block that is the last block in the free region contains "6" and, indeed, the next free region starts from the 6th block.

Finally, freeing allocated memory had to be designed and implemented as well ("*User mnicky*" 2013). The function's signature is: `void SH_free(void *ptr)`. The source code for this function is shown in Appendix 5. A high-level description of the steps performed during de-allocation is shown below:

1. Check if `pntr` refers to a memory location that is inside of the shared heap. If yes, use `free()` function offered by C.
2. Otherwise, find which private heap contains the piece of memory that needs to be freed.
3. Then, free memory by altering relevant free and used entries in the data structure that outlines allocated and free regions of the heap.

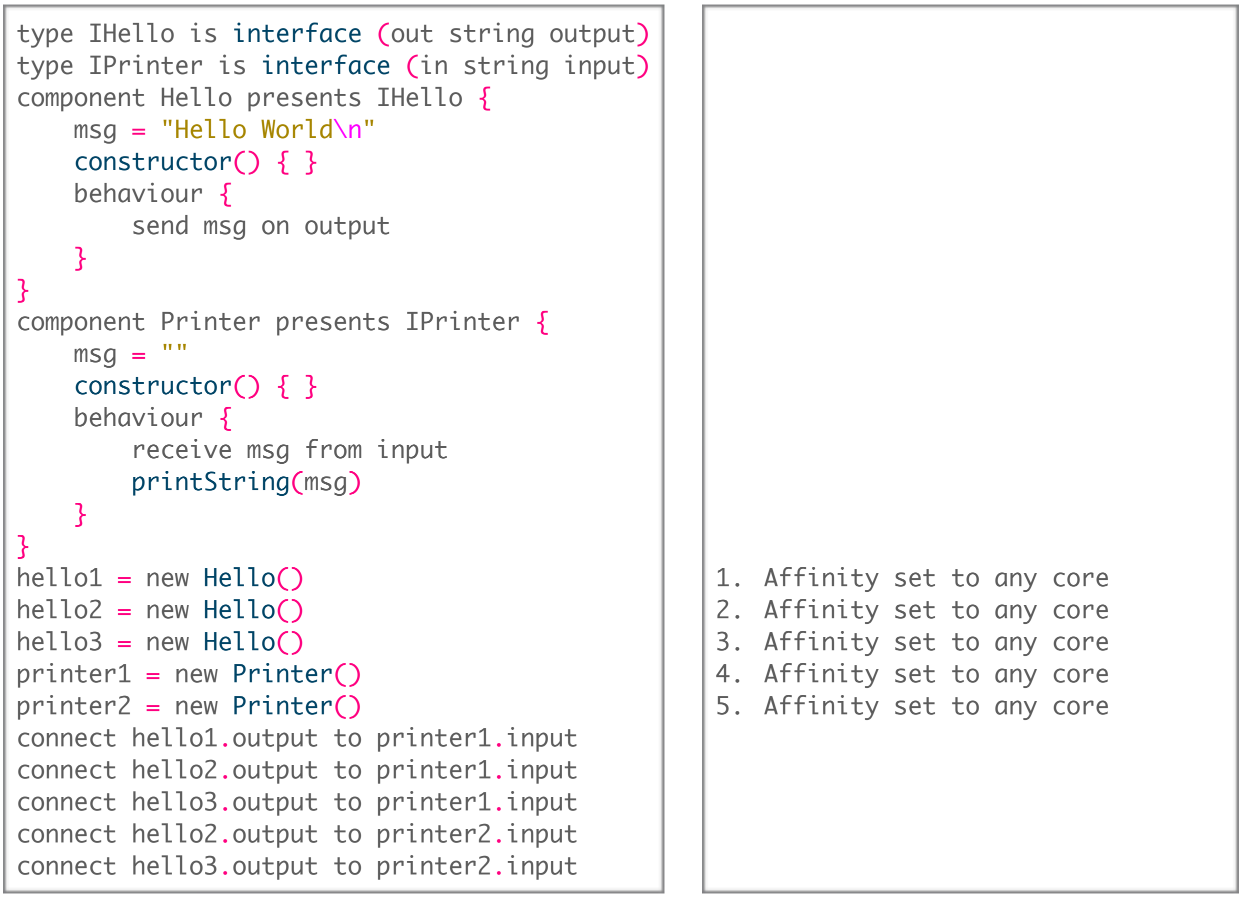
8.3 Thread Placement Schemes

Another important part of the project is investigation of possible thread placement schemes. Code related to Insense components may be found in `affinity.c` and `affinity.h` files. This section discusses the process of implementation of methods described in Section 7.3.

8.3.1 Dynamic Placement

Let us take a modified version of the "Hello" program as an example. Figure 8.31 shows how affinity is set for six instances of two components. To use an example that should hopefully be familiar to the reader, this figure outlines an Insense program that is a realisation of the network topology presented in Figure 3.3 in Section 3.2.

Figure 8.31 (b) shows that affinity of component instances is set dynamically: affinity is decided by the system and the programmer/computer program has no control of the mechanisms behind adjusting affinity of threads.



```
type IHello is interface (out string output)
type IPrinter is interface (in string input)
component Hello presents IHello {
    msg = "Hello World\n"
    constructor() { }
    behaviour {
        send msg on output
    }
}
component Printer presents IPrinter {
    msg = ""
    constructor() { }
    behaviour {
        receive msg from input
        printString(msg)
    }
}
hello1 = new Hello()
hello2 = new Hello()
hello3 = new Hello()
printer1 = new Printer()
printer2 = new Printer()
connect hello1.output to printer1.input
connect hello2.output to printer1.input
connect hello3.output to printer1.input
connect hello2.output to printer2.input
connect hello3.output to printer2.input
```

```
1. Affinity set to any core
2. Affinity set to any core
3. Affinity set to any core
4. Affinity set to any core
5. Affinity set to any core
```

Figure 8.31 (a): “Hello” program that illustrates dynamic thread affinity. **(b)**: setting affinity to instances of Insense components on a machine with 4 cores.

To go into more details. Components are created in the function `component_create` (described in sections 7.1.1 and 8.1.1). When this function is called, it starts a POSIX thread. The thread does not have any affinity set up thereby permitting the OS to decide which core to use for this thread.

8.3.2 Static Placement

This approach relies on taking control of affinity of threads. The static placement of threads depends on manually setting affinity to POSIX threads assigned to Insense components.

Figure 8.32 presents a listing of the functions used for implementation of thread affinity. The function `int setAffinity(pthread_t thread)` is responsible for the actual process of setting affinity of a given thread to a particular core. The strategy for choosing cores that the thread is pinned to is chosen through use of the C preprocessor conditions (e.g. `#if AFFINITY_ALGO == 1`), where `AFFINITY_ALGO` is a constant that indicates which affinity algorithms needs to be used in the current run of the program. It receives the ID of the thread that needs to have its affinity set.

```
/*
 * Set affinity of POSIX thred with pthread_t thread.
 */
int setAffinity(pthread_t thread) {
#if AFFINITY_ALGO == 1
    // Round-Robin
    return setAffinityToCore(thread, receiveCoreIdForThread(thread));
#elif AFFINITY_ALGO == 2
    // Set all threads to one single core
    return setAffinityToCore(thread, 0);
#else
    // Set affinity to a random core
    return setAffinityToCore(thread, rand() % receiveNumberCores());
#endif
}

/*
 * Set affinity of POSIX thread "thread" to core with ID “core".
 */
int setAffinityToCore(pthread_t thread, int core) {
    // First check if value of core exceeds a number of cores in CPU
    if (core >= receiveNumberCores())
        return -1;

    // Define CPUSET which describes the CPU
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(core, &cpuset);

    // Set affinity
    int s = pthread_setaffinity_np(thread, sizeof(cpu_set_t), &cpuset);
    if (s != 0) // Check for errors
        handle_error_en(s, "pthread_setaffinity_np");
    return s;
}
```

Figure 8.32 Functions used for setting affinity of threads.

The function `receiveCoreIdForThread` returns the core that a thread in question is to be pinned to, based on the algorithm in use. This approach works with all algorithms of static placement described in this study.

Another function setAffinityToCore is where the actual alteration of affinity of a given thread takes place. It uses functionality provided by the standard POSIX library. If it encounters an error, a “-1” value is returned; otherwise, a value yielded by the call of pthread_setaffinity_np is outputted from this function.

```
#if !(AFFINITY_ALGO == 0)
    if (core != -1) {
        // Use passed ID of a core.
        setAffinityToCore(this_ptr->behav_thread, core);
    } else {
        setAffinity(this_ptr->behav_thread);
    }
#endif
```

Figure 8.33 Part of component_create function responsible for changing affinity of the newly created POSIX thread.

Figure 8.32 illustrates a part of component_create where a newly created p-thread, which is responsible for executing the behaviour function of a just created component, is statically pinned to a particular core. A parameter core is passed to component_create function. This variable may be used by a developer who wishes to take full control of thread pinning for a particular component instance. “-1” is given as a default value of core when Insense programs are compiled: it indicates that the the parameter needs to be ignored and a value of AFFINITY_ALGO is to be used for choosing an algorithm that handles static affinity. If one wishes to use the core parameter, its value has to be changed in compiled code. As a note, setCore function was also added to Insense, it allows setting affinity to components inside of Insense programs.

8.3.2.1 Even Distribution of Threads Among Cores

The first and, perhaps, the most straight-forward approach to static thread placement is distributing threads evenly among cores by setting AFFINITY_ALGO to “1”. The “Hello” program will serve as an example of how it works. Threads are assigned to cores from the pool of available units of computation, one by one. Figure 8.31 demonstrated how affinity of threads is set to cores dynamically. A reader may use this figure as an example for this scheme as well. In this case instances of Hello component would be set to cores 1, 2, and 3 respectively; instances of Printer: 4, 1, and 2 respectively Providing that threads perform computations of the same level of complexity, this approach attempts to make equal use of all cores in the system. If a number of threads is not a multiple of the amount of cores, certain cores may be assigned to one extra thread.

8.3.2.2 Static Placement of Threads to a Single Core

In this vision of static placement all threads are pinned to a single core. In order to use this scheme `AFFINITY_ALGO` needs to be set to "2". Once again, Figure 8.31 may be seen as an example. Only, in this case all instances of Insense components would be pinned to a single core (e.g. "Core 3"). On systems where all cores inside of a processor are equal and represent essentially the same piece of hardware, the ID of the core that is chosen for changing affinity of threads has no impact on the program.

8.3.2.3 Static Placement of Threads Based on Communication

With this algorithm, threads are linked to cores based on the nature of communication with other Insense components that they undergo. Benefits of using this system may be shown when threads that have high levels of communicating with each other are set to the same core. Section 7.3.2 outlines grouping components into five categories.

9 EXPERIMENTS

All experiments were performed on identical pieces of hardware. The computers available in the MSc laboratory of St Andrews University were used in the project. They have multi-core processors with four cores (Intel® Core™ i5-3470S, 2.90GHZ) and 8GB of RAM. The processor supports Intel® Hyper-Threading Technology (Intel® HT Technology) that allows an execution core to function as two logical processors. While some execution resources such as caches, execution units, and buses are shared, each logical processor has its own architectural state with its own set of general-purpose registers and control registers. This feature must be enabled using the BIOS and requires operating system support.

Experiments were performed using Scientific Linux 6.3. The `-O0` flag of the compilation command was used to remove effect of optimisation of gcc compiler ("*gcc(1)*" 2013), it was done to be able to have full control of thread placement and memory management. Execution time of experiments was measured using the `time(1)` ("*time(1)*" 2013) function, which shows both CPU and real time. For the sake of brevity, the results of timing experiments presented in this section contain only *real* time measurements (outputted by `time(1)` function). Appendix 9 contains more detailed graphs that also feature *system* and *user* time in case the reader wishes to see these as well.

9.1 Effects of Memory Management Schemes on Allocation and Deallocation of Memory

The main goal of this experiment is to look into benefits of using multiple private heaps for allocation and deallocation of memory.

9.1.1 Design of Experiments

Experiments:

- Different numbers of component instances - one and four - are instantiated. They allocate space for a large 2-dimensional (2D) array in each iteration of `behaviour` - dynamic allocation takes place. Both memory management schemes are tested: private heaps / shared heap. Instances are run on a single core, their affinity is set dynamically, and when they are evenly distributed between cores.

Parameters affecting results:

- Size of the 2D array. Sizes tested: 100 by 5 = 500 integers and 1000 by 5 = 5000 integers.

Evaluation:

- Compare execution time.

Expected results:

- “Lock behaviour” is expected, i.e. in case of using a single shared heap, instances of components need to wait their turn to perform allocation and deallocation due to the heap being locked by another core. Private heaps should perform considerably better.
- Also, fragmentation will have its effect: allocation will be further slowed down since more time will be spent on looking for a free space in the heap. Fragmentation is more likely to be visible if the experiment has more than a single component running on the same core. Therefore, this experiment has 4 components for running on a 4-core processor.

9.1.2 Program Used

Appendix 6 presents source code of a program that was used to conduct experiments outlined in Section 9.1. This simple program contains a single Insense component that creates a matrix of integers inside of its `behaviour`. A number of instances of the component are created, each creates an array on every iteration of `behaviour`. The array is then garbage collected (memory assigned for it is deallocated) at the end of the `behvaiour` prior to the next iteration of the `behaviour` loop.

9.1.3 Experimental Results

The only parameter varied in this experiment is size of the array `a` in the `Comp` component. Figure 9.1 shows results of the first experiment that was run: comparing performance of the "shared heap" ("SH") and "private heaps" ("PH") schemes when used with a single component instance, for which affinity is dynamically chosen by the OS (since there is only one instance, other thread placement schemes considered in the study do not produce different results). "*Dynamic*" indicates dynamic placement of threads, where the OS takes care of thread placement. This notation is used throughout this chapter.

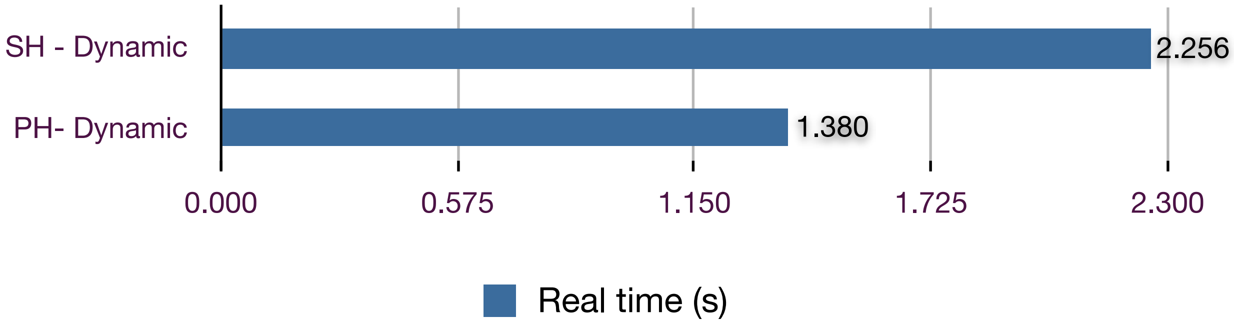


Figure 9.1: Results of running an experiment with utilising multiple heaps for (de-)allocation of memory. One component.

Figure 9.2 below shows execution times of running various combinations of memory and thread placement schemes, when four component instances are used. "*1 core*" refers to a scheme of thread placement where all threads are pinned to a single core; "*RR*" refers to the round-robin scheme were threads are evenly distributed over cores, these schemes are described in more detail in Section 7.3. This notation is also used throughout this chapter. In total six cases were considered for this experiment.

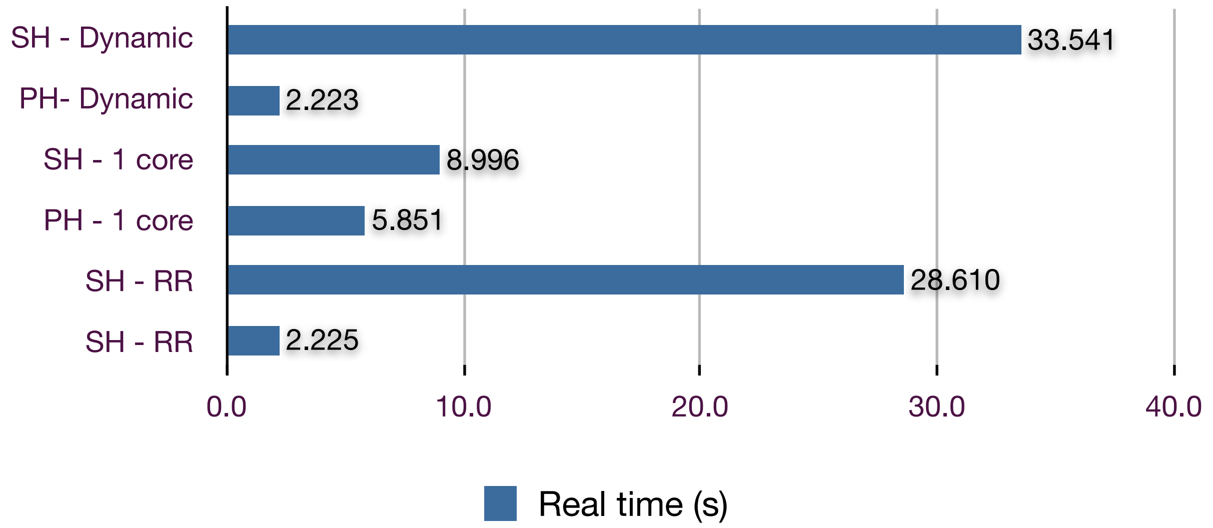


Figure 9.2: Results of running an experiment with utilising multiple heaps for (de-)allocation of memory. Four component instances and size of array: 1000x5 elements.

These results are discussed in the following section.

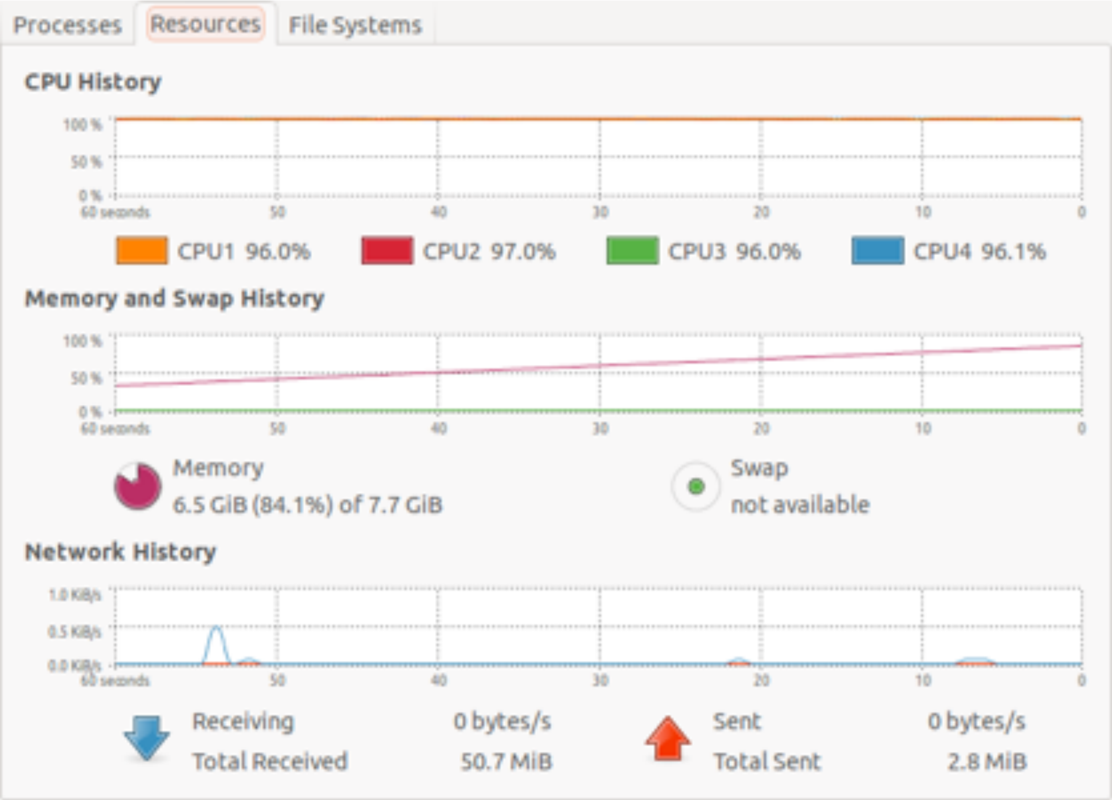
9.1.4 Discussion of Results

Figure 9.1 shows results of running a single component that performs a large number of allocations and deallocations. It outlines that the "private heaps" scheme offers speedup of a factor 1.5 compared to the "shared heap" scheme in this base example where no parallel execution of Insense components takes place. This difference in performance may be explained by fragmentation of memory that occurs when a single shared heap is utilised.

Next, Figure 9.2 contains results of executing a more complex example where four component instances allocate and deallocate space for the array in their `behaviour` functions. First, comparison of execution time of the "*SH - Dynamic*" and "*PH - Dynamic*" configurations is given (notation used in the figures is described in Section 9.1.3). This example shows that utilising the "private heaps" scheme along with dynamic placement of threads offers a considerable speedup by a factor of 16, compared to using the "shared heap" scheme. Degradation in performance for the "shared heap" scheme is likely to be due to multiple threads trying to access a single shared heap concurrently from different processor cores. Appendix 9.2, which contains measurement of CPU time (user time + system time) for "*SH - Dynamic*", shows that most of the CPU time is spent in kernel mode, which can be explained by reduction of performance caused by locking that occurs in the dynamic memory allocation routines; in case of Doug Lea's malloc implementation - by using mutexes (Lea 1996). Such locking is not present when allocating into a private heap, in which case each core can allocate and deallocate memory concurrently with other allocation and deallocation calls taking place in other private heaps.

Further, improved performance of "*SH - 1 core*" compared to the "*SH - Dynamic"* can be explained by the fact that allocations and deallocations do not take place at the same time. If all Insense component are run on the same core, there is less chance that locking must be imposed to secure a shared heap: if there is less chance, less time is spent on locking. "*PH - 1 core*" still performs better that "*SH - 1 core*". The reason behind: even though threads are interleaved, there is no concurrent access to the heap. With the "private heaps" scheme time is not lost because no locking due to mutex/semaphore protection takes place.

Lastly, results achieved by using the "*PH - RR*" configuration are of similar order as with "*PH Dynamic*". In this case, multiple cores are also trying to access shared heap concurrently. "*PH - RR*" offers a speedup by a factor of 12 compared to "*SH - RR*".



CPU1 96.0%
CPU2 97.0%
CPU3 96.0%
CPU4 96.1%

Memory
6.5 GiB (84.1%) of 7.7 GiB

Swap
not available

Receiving 0 bytes/s
Total Received 50.7 MiB

Sent 0 bytes/s
Total Sent 2.8 MiB

Figure 9.3: Running out of memory with an array of size 850x5 elements.

Another issue when using "shared heap" allocation scheme was detected: all programs suffered from gradual increase in memory usage. In certain cases, meaning that the computer would run out of memory. It was put down to inefficient usage of the shared heap. In these cases Valgrind ("*Valgrind's homepage*" 2012) was used to verify that there was no actual memory leak in the program. Figure 9.3 shows a screenshot of the system state before the OS halted.

9.2 Effects of Memory Management Schemes on Cache Usage

The goal of this experiment is to investigate benefits of using multiple private heaps for improving memory cache usage. Size of Level 1 cache on the machine used for this experiment is 32KB and cache line size is 64B.

9.2.1 Design of Experiments

Experiments:

- A component has four instances, which increment values in the 2-dimensional array during each iteration of its `behaviour` function. The instances are placed on a single core, their affinity is set dynamically, and placed evenly. Both memory management schemes are tested: "shared heap" and "private heaps". Two configurations of the size of the array are used: ~93% and ~24% of the size of L1 cache.

Parameters affecting results:

- Size of the array inside of the component.

Evaluation:

- Compare rate of Level 1 cache misses. Section 7.2.3 introduces the concept of cache misses. The Cachegrind tool, which is a part of Valgrind ("*Valgrind's homepage*" 2012) distribution, was utilised to measure cache misses. Cachegrind can be used to run experiments for L1 cache, but not L2 cache.
- Compare execution time.

Expected results:

- Private heaps are expected to perform better because of a reduced number of cache misses and, hence, faster performance. The expectation is that interleaving allocation of memory, as discussed in Section 7.2.2, will result in the less optimal use of directly mapped and set-associative cache.

9.2.2 Program Used

Appendix 7 presents source code of a program that was used to conduct experiments outlined in Section 9.2. There is a single Insense component in this program. This components contains a 220 by 5 2-dimensional array `a` that contains integers, which for an Insense 2-D Array struct occupies 7,928B (24% of the size of L1 cache used in the testing environment). Another configuration used in the experiment is 850 by 5 (93% of the size of L1 cache). The component performs incrementation of elements in the array during each iteration of `behaviour`.

9.2.3 Experimental Results

The parameter that affected results achieved in this experiment is dimensions and size of the array `a` of integers in the component `Comp`. Various sizes of the array were experimented with. Achieved results indicated a pattern that is consistent throughout all configurations used. Figure 9.4 shows results of the experiment where a 2-dimensional array of size 220 by 5 was put to use: comparing rate of cache misses for "private heaps" and "shared heap" memory management schemes under three thread placement models: "*Dynamic*", "*1 core*", and "*Round-Robin*" ("*RR*"); these thread placement schemes are described in detail in Section 7.3, notation used in Figure 9.3 is described in Section 9.1.3. Appendix 11 presents full output of Cachegrind for this test.

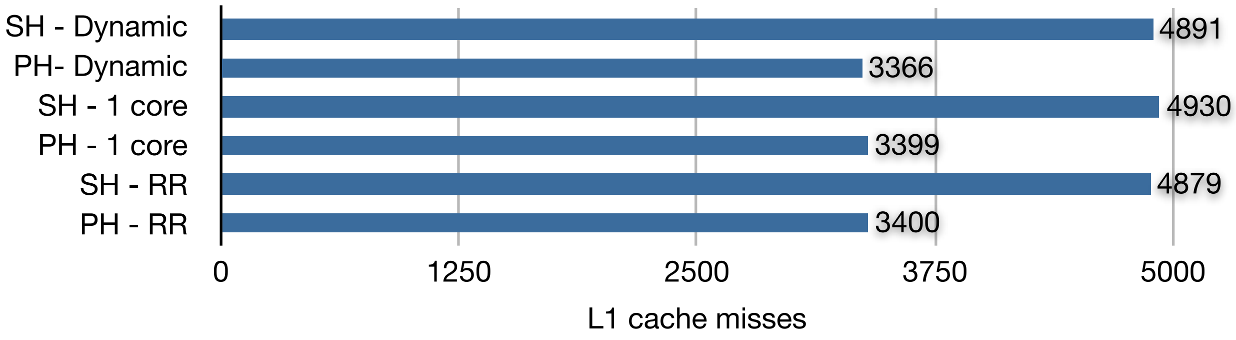


Figure 9.4: Amount of cache misses with different memory and thread placement schemes.

Another metric used to evaluate performance is execution time. Figure 9.5 shows a graph of execution time measurements with different memory management and thread placement schemes. In this case the program operates with a 2D array of size 30,608B, which is ~93% of L1 cache size.

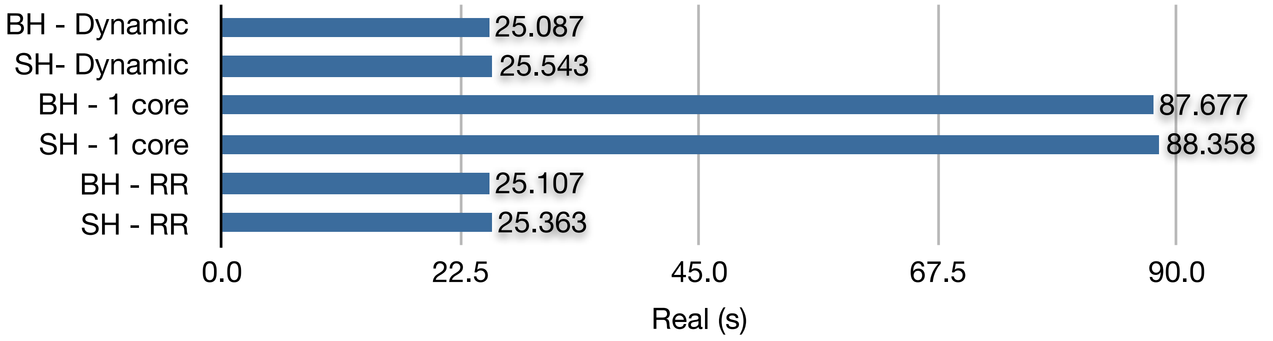


Figure 9.5: Execution time of `test_cache` program with an array of size 850 by 5 and 500,000 `behaviour` cycles.

Figure 9.6 also present readings of execution time of the program outlined in Section 9.2.2. In this case the program operates with an array of size 9,008B (~27% of L1 cache).

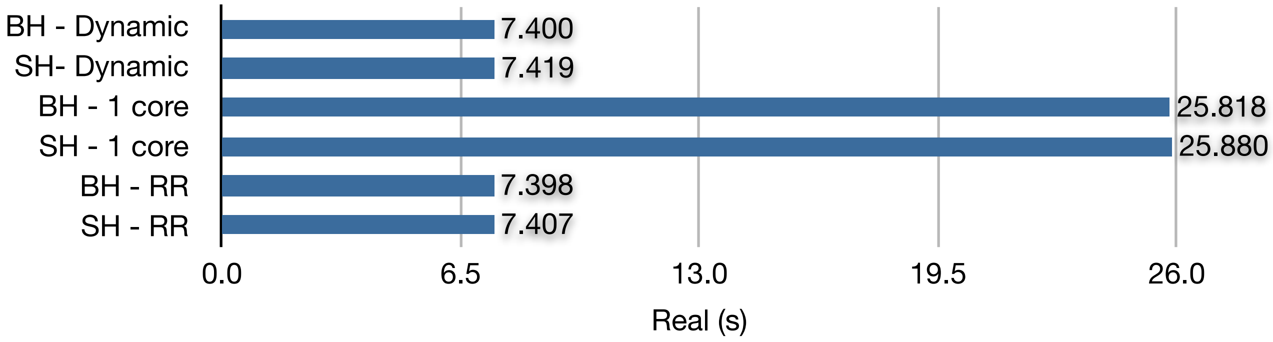


Figure 9.6: Execution time of `test_cache` program with an array of size 220 by 5 and 500,000 `behaviour` cycles.

These results are discussed in the following Section.

9.2.4 Discussion of Results

The conducted experiment shows that when a single shared heap is utilised for memory management, cache lines in a cache of a particular core are overwritten by data that is not desired by the unit of computation in question. Furthermore, due to potential fragmentation of memory allocations, as described in Section 7.2.2, a situation may occur where a certain element of the 2D array inside of the component that is run on the core maps to the same cache block as another element. As a result, accesses to that element of the array will evict other array elements from the cache, resulting in a cache miss when trying to access the evicted member on later stages of execution of the program.

On the other hand, for the private heaps an expectation was to achieve less fragmentation: each POSIX thread allocates into its own private heap, so regardless of allocation order, all allocated memory is contiguous. Additionally, L1 caches are private for each CPU core. Therefore, in this situation, a better use of the cache and barring capacity misses (when the cache is too small for the array), the expectation was to receive fewer conflict misses.

And, indeed, results described in Section 9.2.3 demonstrate that a larger number of cache misses occurs when a single shared heap is utilised for memory management. One can see that under all thread placement schemes "private heaps" had fewer cache misses. The achieved difference in readings of cache missed in the "shared heap" and "private heaps" schemes is ~30%, in favour of "private heaps". Figures 9.5 and 9.6 demonstrate that no significant reduction of execution time of programs is achieved by utilising private heaps for these particular examples. A couple of speculations about why this result did not match expectations may be presented: 1. the Valgrind simulation is not precise enough, or a better set of parameters could be used; 2. the 256kB unified L2 cache (private to each core) has a pre-dominant effect on performance.

9.3 Effects of Thread Placement and Communication Between Components on Performance

The main goal of this experiment is to investigate effects of thread placement and communication between components on performance.

9.3.1 Design of Experiments

Experiments described lower follow communication patterns, which are introduced in Section 7.3.2. All results reported here are from experiments run with the "shared heap" memory management scheme. Through conducting experimentations, it was shown that choice of the memory management scheme does not affect results achieved through running experiments described in this section.

Experiments:

1. *“Sender - Receiver”* pattern. This experiment was run with two senders and two receivers. Each sender is connected to one receiver, no receiver is connected to more than one sender.
2. *“Client - Server”* pattern. Configuration: three servers and twelve clients where each server is connected to 3 clients, no client is connected to more than one server.
3. *“Dispatch - Worker”* pattern. Configuration: three dispatch instances and twelve workers where each dispatch instance is connected to three workers, no worker is connected to more than one dispatch instance.
4. *“No communication”* pattern. Operates with four component instances. They perform computation (incrementation of elements in an array defined inside of the component). As the name suggests, no inter-component communication takes place.

All experiments are run on all three thread placement schemes taken into account in this study (Section 7.3 describes them).

Parameters affecting results:

- Amount of computation performed by instances.
- Amount of overhead caused by channel communication.

Evaluation:

- Compare execution time.

Expected results:

- In cases of experiments where component instances have inter-component channel communication, running instances that have channel communication between each other on the same core is expected to be either as efficient or more efficient than running on multiple cores, due to reduced channel communication overhead when communicating between components that are on different cores.
- Results of running the “*Client - Server*” and “*Dispatch - Worker*” experiments dealing with the “N:M” communication scheme should be similar.

9.3.2 Programs Used

Appendix 8 presents source code of programs that were used to conduct experiments outlined in the previous subsection.

Appendix 8.1 demonstrates a program used to test the impact of thread placements schemes on the "*Sender - Receiver*" communication pattern. Neither receiving nor sending components perform computation. This program is used to experiment with adjusting the level of channel communication and observ impact of this action on performance of thread placement schemes.

The program listed in appendix 8.2 focuses on the "*Client - Server*" communication pattern. Instances of the `Client` component perform no calculation. Their job is to determine how much computation connected `Server` instances must undertake. Instances of the `Server` component perform incrementation of elements in the array `a`. This operation is carried out `runTime` number of times. The amount of computation completed during each iteration can be adjusted by modifying value of the variable `calcTime` sent by a client: it indicates how many times values in the array are incremented. As shown in the code listing, `Server` instances may be connected to multiple `Client` instances.

Appendix 8.3 outlines source code of a program used to test the "*Dispatch - Worker*" communication pattern. This program is identical to what is shown in appendix 8.2, except in this configuration a single instance of `Dispatch` is connected to multiple instances of `Worker`. The `Worker` components performs incrementation of the array and the `Dispatch` instance has to wait to send data until at least one `Worker` component is ready to accept data.

A program shown in Appendix 8.4 deals with the "*No communication*" pattern. This program allows to test a situation where no overhead caused by inter-component communication takes place. Instances of the `Server` component perform incrementation of elements of the array `a`. A number of times incrementation needs to take place is defined by the `runTime` variable.

9.3.3 Experimental Results

Firstly, Figure 9.7 shows results of running an experiment with the "*Sender - Receiver*" communication pattern where all instances of `Sender` and `Receiver` components that are part of the communicating pair of components are running on the same core ("*Same core"*), on different cores ("Different cores"), or placed dynamically ("Dynamic").

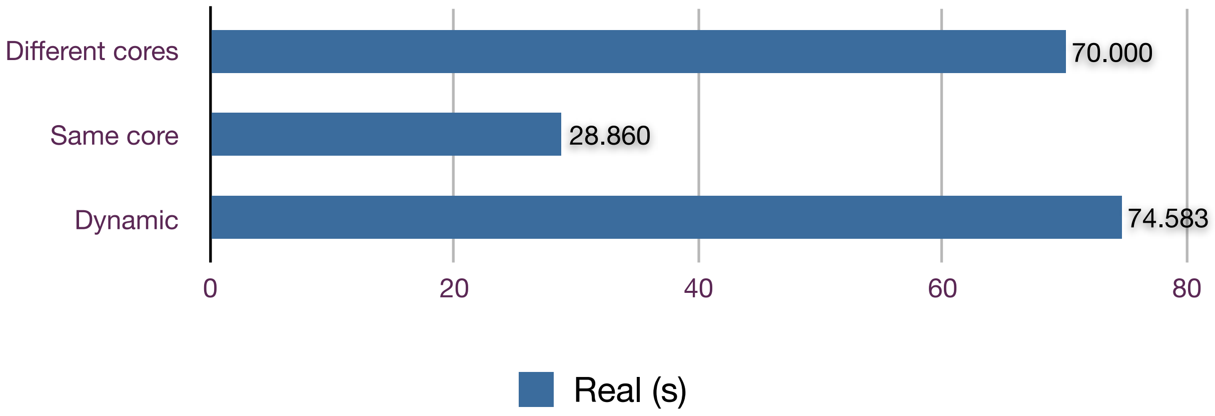


Figure 9.7: Results from "*Sender - Receiver*" example with two senders and two receivers.

Secondly, Figure 9.8 below shows results of running an experiment with the "*Client - Server*" communication pattern. In this situation, four thread placement schemes are taken into account: "*Dynamic*" indicates dynamic placement of threads, where the OS takes care of thread placement; "*1 core*" refers to a scheme of thread placement where all threads are pinned to a single core; "*Servers on individual cores and all clients on 1 core*" is a scheme where all servers (in this case there are three of them) are placed on three cores, one server per core, and all twelve clients are pinned to a single forth core; "*1 server and 3 clients on the same core*" points to a configuration where groups of components that have communication between each other are placed on the same core: there are four groups of such components (one server and three clients in each) in this set-up. Values on the Y-axis represent configurations used in the experiments. The first integer in the pair of numbers (e.g. "400, 100") represents a number of `behaviour` cycles of a single `Server` instance, hence it defines how much channel communication takes place. The second number shows how many times incrementation of elements in the array is performed during one `behaviour` cycle, thus establishing amount of computation performed. These values are consistent throughout all `Server` instances.

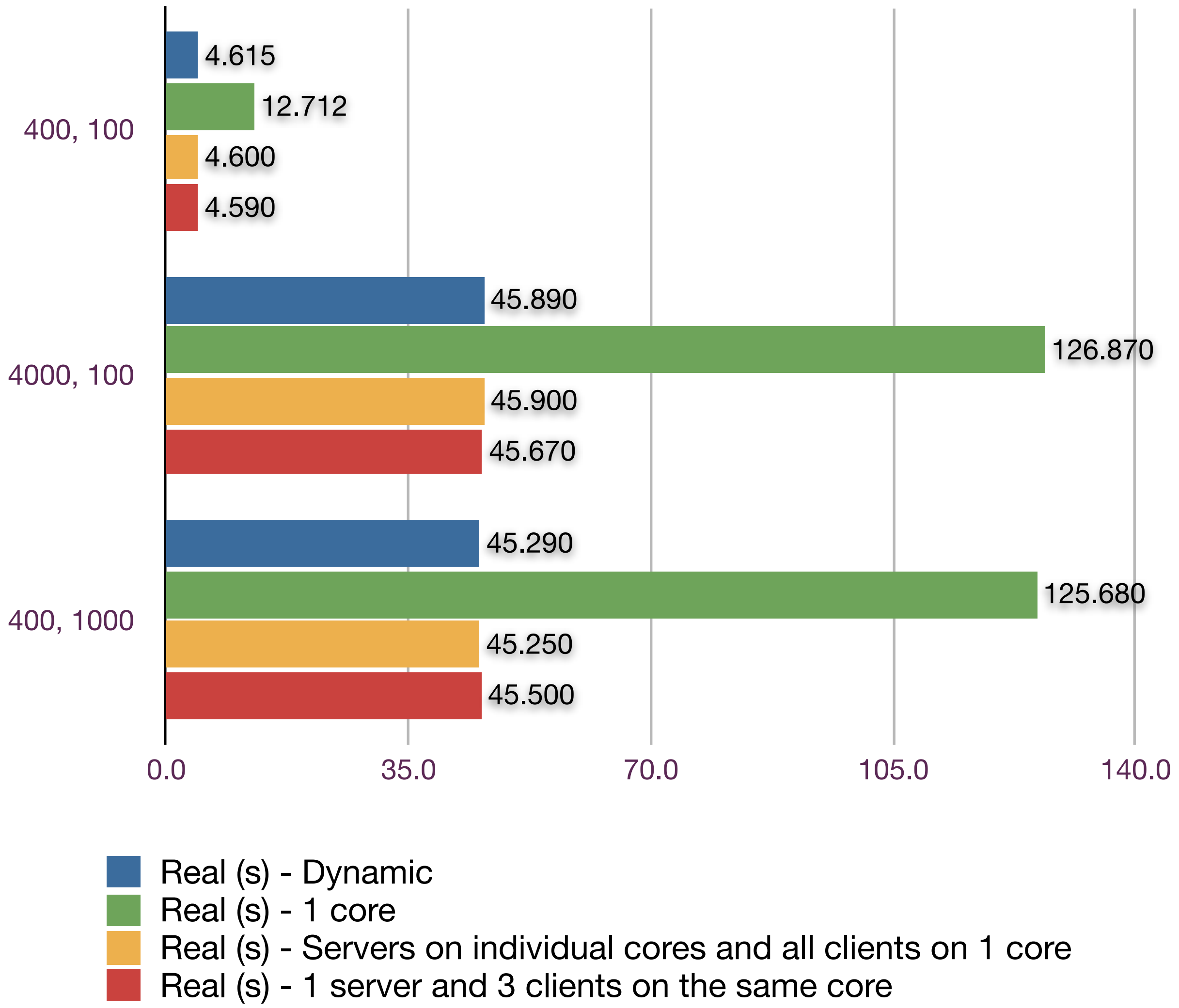


Figure 9.8: Results from “*Client - Server*” example with 12 clients and 3 servers.

Thirdly, Figure 9.9 contains results of running an experiment with the “*Dispatch - Worker*” design pattern (Section 7.3 introduces this pattern). In this pattern, four thread placement schemes are taken into account: “*Dynamic*” and “*1 core*” (both are described in the previous paragraph); “*3 workers a core on 3 cores and all dispatch instances on 1 core*” is a scheme where all workers (in this case there are twelve of them) are placed on three cores, three workers per core, and all three dispatch instances are pinned to a single forth core; “*1 dispatch and 3 workers per core on 3 cores*” is a scheme where each group of one dispatch instance and three workers are placed on a different core (within a group, worker and 3 dispatch instances are on the same core). Note that in the last scheme, one server is left without any threads pinned to it. As with the previous figure, values on the Y-axis represent configurations used in the experiments. The first integer in the pair of numbers (e.g. “400, 100”) represents a number of `behaviour` cycles of a single `Worker` instance. The second number shows how many times incrementation of elements in the array is performed during one `behaviour` cycle. These values are consistent throughout all `Worker` instances.

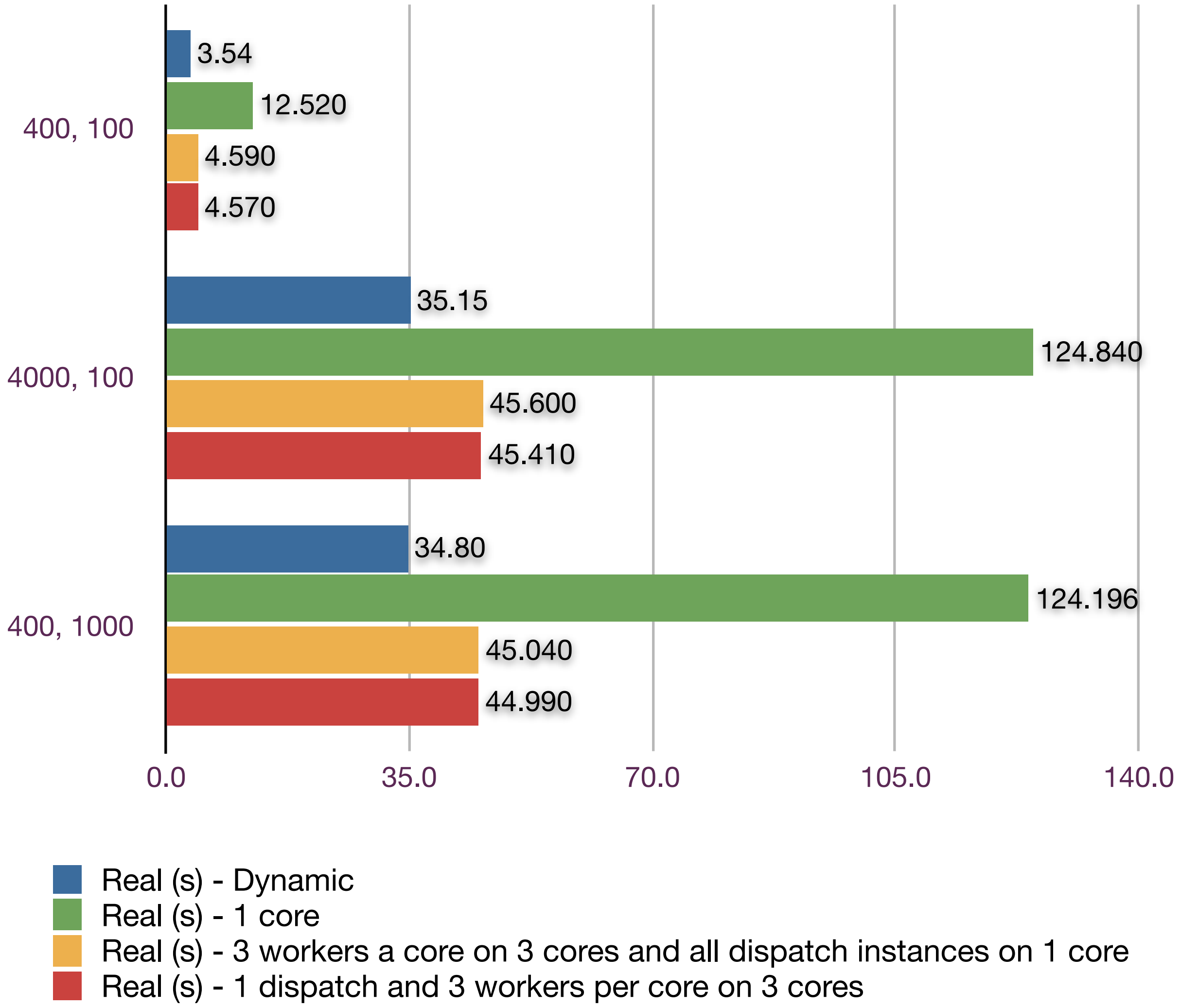


Figure 9.9: Results from "*Dispatch - Worker*" example with 3 dispatch instance and 12 workers.

Finally, Figure 9.10 below shows results of running an experiment with the "*No communication*" pattern. In this case, three thread placement schemes are taken into account: "*Dynamic*", "*1 core*", and "*Round-Robin*". All of these schemes are described briefly in the beginning of this Section and in more details in segment 7.3.2. Once again, values on the Y-axis represent configurations used in the experiments. The first integer in the pair of numbers (e.g. "400, 100") represents a number of `behaviour` cycles of a single component instance The second number shows how many times incrementation of elements in the array is performed during one `behaviour` cycle. These values are consistent throughout all instances.

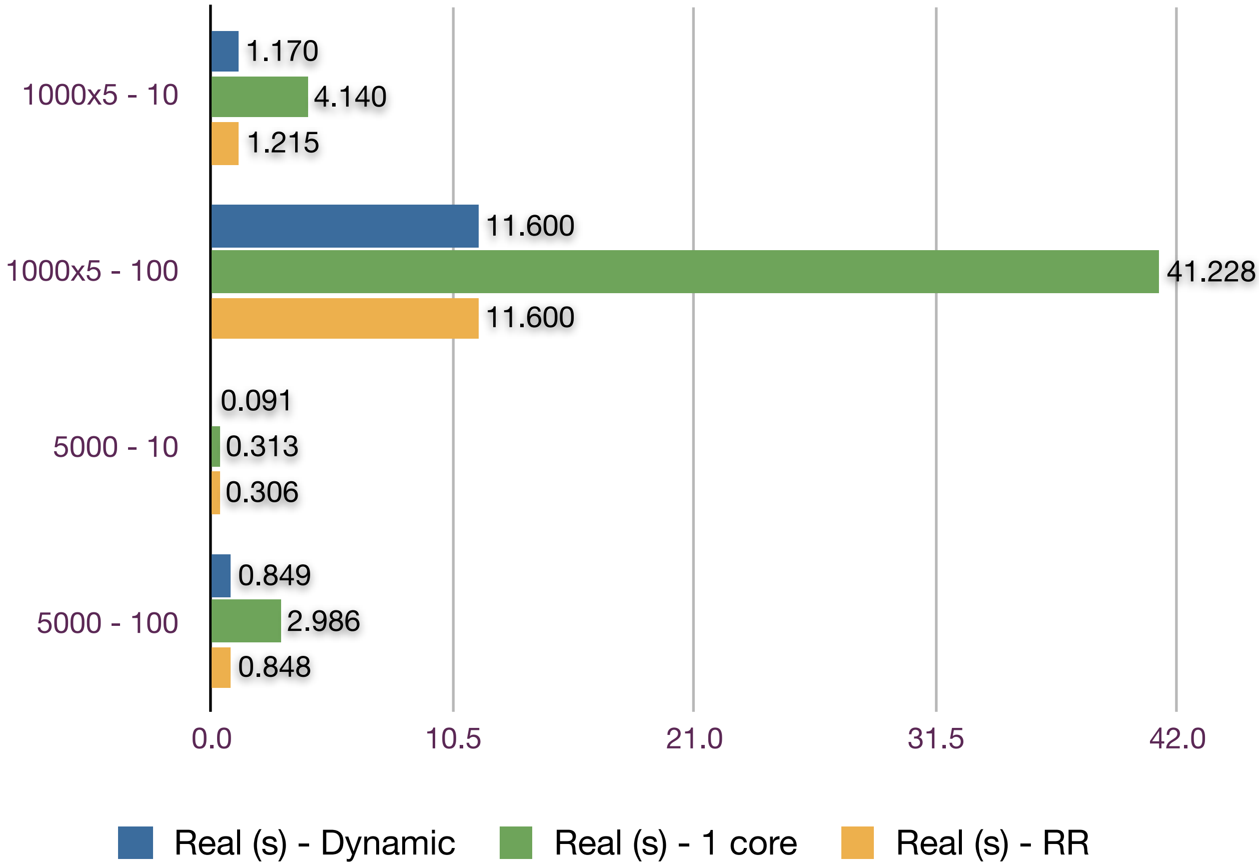


Figure 9.10: Results from "*No communication*" example with 4 workers and two sizes of the array.

These results are discussed in the following Section.

9.3.4 Discussion of Results

With the "*Sender - Receiver*" example (see Figures 9.7 for results) received results show that utilising a single core for placement of pairs of connected `Sender` and `Receiver` instances is more efficient than both placing them on different cores and using dynamic thread placement. It may be explained by the fact that component instances used in the test perform no heavy computation and their execution time depends heavily on the overhead caused by inter-component communication with another instances. Appendix 9.3 makes it visible that system time spent with the "*Different cores*" scheme is relatively large, it can be explained by threads being stalled while cache coherence is assured. Cache coherency is assured due to one thread having access to data that other thread has access to, this is due to accessing semaphores and mutexes in the channel synchronisation code. Copying memory during channel communication between sender and receiver also takes place, which has further impact on total execution time. “*Dynamic*” scheme was the slowest. It may be explained by the fact that in order to balance load, the OS may decide to place the components on different cores, which in turn slows down inter-component component communication.

With the “*Client - Server*” example (see Figure 9.8) received results show that utilising a single core for placement of components that are engaged in the M:1 communication scheme is not reasonable. It may be explained by the following logic: servers perform considerable amounts of computation and when they are placed on the same core as their clients and other servers only one server can execute instructions at any given moment in time. In two static placement schemes that were tested in this experiment an attempt was made to anticipate the load due to work and communication and then distribute servers and clients over cores. Results showed that the OS provides efficient load balancing in this case, which is similar to the custom-defined load balancing static configuration mentioned before.

The “*Dispatch - Worker*” example. As expected, results achieved through conducting experiments with this scheme (see Figure 9.9) mirror those of the “*Client - Server*” example: utilising a single core for placement of components that are engaged in the N:M communication scheme is not reasonable. Dynamic thread placement scheme showed the best results. Once again, anticipation of where to place load statically was attempted. However, it is impossible to know what other processes take place on the machine during execution of the experiment. In this case dynamic always outperforms static placement schemes.

Lastly, Figure 9.10 shows results of running the “*No communication*” example. This experiment did not have a surprising outcome. As expected, placing component instances that do not rely on channel communication on a single processor core resulted in the total execution time being a multiple of the number of cores (in this case - 4) higher than what is achieved by dynamic thread placement. The “*Round-Robin*” scheme produces results that are of similar order as those from the dynamic placement.

10 CONCLUSIONS AND FURTHER WORK

This project involved adapting the Insense compiler and runtime to run on Unix-based systems and make use of the a variety of thread placement and memory allocation schemes. The main objective of the project was to compare the efficiency of these schemes on multi-core systems for the Insense programming language. All primary objectives outlined in Chapter 2 were successfully completed. The secondary objective, which was to evaluate the potential benefit of strong encapsulation of components, was not completed because of lack of time.

Memory placement schemes under investigation included: utilisation of private private heaps for component instances (one heap per component) compared to permitting memory to be allocated on the shared large heap. With regard to allocation and deallocation of memory taking place in component instances, on different cores, the efficiency of using a private heap for each component resulted in speedup up to a factor of 16, compared to utilising a shared large heap. Using

Thread placement schemes under examination were: distributing component instances over cores, statically placing them on a single core, distributing over cores according to communication pattern, and allowing the OS to set their affinity depending on the load. Distributing components over cores according to communication patterns, for the most part performed similar to trusting the OS to take care of thread placement, apart from the "*Sender - Receiver*" scheme in which static placement outperformed due to the fact that the computational load of components was almost entirely due to channel communication. This most likely caused the OS to place components on different cores, thereby slowing down channel communication. As a result, dynamic placement of components by the OS resulted in a performance drop by a factor of 2.6 when compared to the static placement scheme

The accuracy of data received through experiments outlined in this document can be improved if a single user mode ("*Single User*" 2006) available on Linux was used. It would reduce impact on outcomes caused by the OS and other processes that run on the machine during experiments. The laboratory set up used for testing did not permit to use this mode easily.

Further work that can be done in this project may involve: adapting compiler to place components according to statically known communication patterns; modifying the channel communication to be more efficient for multi-core architecture; allowing private heaps to be elastic - giving heaps ability to change their size at runtime; due to the design of the Insense language (prevention of nested structs, fixed compile-time specification of array sizes), the Insense compiler could also be adapted to work out the worst-case dynamic memory usage which could then be used when allocating a private heap for a component; different options in the channel implementation could be explored in which dynamically allocated data types may be copied by value to another heap to avoid inter-heap references; providing support for other operating systems such as Mac OS; a number of other optimization such as further improving access to arrays.

11 BIBLIOGRAPHY

Bell, A. 2013, *Insense on Unix*, University of St Andrews.

Daconta, M.C. 1993, C pointers and dynamic memory management, QED, London.

Dearle, A. 2011, *Insense Tutorial, University of St Andrews.*

Dearle, A., Balasubramaniam, D., Lewis, J. & Morrison, R. 2008, *A Component-Based Model and Language for Wireless Sensor Network Applications*, Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference, IEEE Computer Society, Washington, DC, USA, pp. 1303.

Devarakonda, M. & Mukherjee, A. 1992, Issues in Implementation of Cache-Affinity Scheduling.

Garg, A. 2009, *Real-Time Linux Kernel Scheduler* [Linux Journal]. Available: http://www.linuxjournal.com/magazine/real-time-linux-kernel-scheduler. Updated 1.9.2009. Referred 22.5.2013.

gcc(1) - Linux man page [die.net] 2013. Available: http://linux.die.net/man/1/gcc. Referred 19.6.2013.

Geer, D. 2005, *Chip makers turn to multicore processors*, Computer, vol. 38, no. 5, pp. 11-13.

Ghosh, S., Martonosi, M. & Malik, S. 1997, *Cache miss equations: An analytical representation of cache misses*, Proceedings of the 11th international conference on SupercomputingACM, pp. 317.

Gove, D. 2011, *Multicore application programming : for Windows, Linux, and Oracle Solaris*, Addison-Wesley, Upper Saddle River, NJ., London.

Gupta, A., Tucker, A. & Urushibara, S. 1991, *The impact of operating system scheduling policies and synchronization methods of performance of parallel applications*, ACM SIGMETRICS Performance Evaluation Review, ACM, pp. 120.

Halfacree, G. 2012, *IBM releases "world's most powerful" 5.5GHz processor* [Bit-tech]. Available: http://www.bit-tech.net/news/hardware/2012/08/29/ibm-zec12. Updated 29.8.2012. Referred 23.5.2013.

Handy, J. 1998, *The cache memory book*, 2nd edn, Morgan Kaufmann Pub.

Harvey, P., Dearle, A., Lewis, J. & Sventek, J.S. 2012, *Channel and Active Component Abstractions for WSN Programming - A Language Model with Operating System Support*, SENSORNETS, eds. M. van Sinderen, O. Postolache & C. Benavente-Peces, SciTePress, pp. 35.

Hennessy, J.L. 2007, *Computer architecture : a quantitative approach*, 4th edn, Morgan Kaufmann, London.

Hill, M.D. 1988, *A case for direct-mapped caches*, Computer, vol. 21, no. 12, pp. 25-40.

Intel Introduces The Pentium® 4 Processor 2000. Available: http://web.archive.org/web/20070403032914/http://www.intel.com/pressroom/archive/releases/dp112000.htm. Updated 20.11.2000. Referred 29.5.2013.

Knuth, D.E. 1997, *The art of computer programming. Volumes 1-4*, Addison-Wesley, Upper Saddle River.

Lam, M.D., Rothberg, E.E. & Wolf, M.E. 1991, *The cache performance and optimizations of blocked algorithms*, ACM SIGARCH Computer Architecture NewsACM, pp. 63.

Lea, D. 1996, *A Memory Allocator*. Available: http://g.oswego.edu/dl/html/malloc.html. Referred 14.6.2013.

Lewis, J. 2013, *Repository of the Insense compiler for InceOS* [University of St Andrews]. Available: http://quicksilver.hg.cs.st-andrews.ac.uk/InsenseCompilerInceOS/file/0ed57d153be3/insense_progs. Updated 7.5.2009. Referred 30.5.2013.

Lewis, J. & Dearle, A. 2011, *High-level abstractions for programming self-managing wireless sensor network applications*.

*malloc(3) - Linux man pag*e [die.net] 2013. Available: http://linux.die.net/man/3/malloc. Referred 24.5.2013.

May, E.L. & Zimmer, B.A. 1996, *The Evolutionary Development Model for Software*, Hewlett-Packard Journal, vol. 8.

McKee, S.A. 2004, *Reflections on the memory wall*, Proceedings of the 1st conference on Computing frontiersACM, pp. 162.

mmap [The Open Group] 2004. Available: http://pubs.opengroup.org/onlinepubs/009695399/functions/mmap.html. Referred 5.5.2013.

Park, N., Hong, B. & Prasanna, V.K. 2003, *Tiling, block data layout, and memory hierarchy performance*, Parallel and Distributed Systems, IEEE Transactions on, vol. 14, no. 7, pp. 640-654.

Philbin, J., Edler, J., Anshus, O.J., Douglas, C.C. & Li, K. 1996, T*hread scheduling for cache locality*, SIGOPS Oper.Syst.Rev., vol. 30, no. 5, pp. 60-71.

pthread_create(3) - Linux man page [die.net] 2013. Available: http://linux.die.net/man/3/pthread_create. Referred 31.5.2013.

Rouse, M. 2007, *Definition of multi-core processor* [TechTarget]. Available: http://searchdatacenter.techtarget.com/definition/multi-core-processor. Updated 27.3.2009. Referred 23.5.2013.

Salehi, J.D., Kurose, J.F. & Towsley, D. 1995, *The performance impact of scheduling for cache affinity in parallel network processing*, High Performance Distributed Computing, 1995., Proceedings of the Fourth IEEE International Symposium onIEEE, pp. 66.

Schauer, B. 2008, *Multicore processors–A necessity*, ProQuest Discovery Guides 1–14.

Schmid, P. 2004, *NetBurst Architecture: Now 31 Pipeline Stages* [Tom's hardware]. Available: http://www.tomshardware.com/reviews/intel,751-5.html. Updated 2.1.2004. Referred 29.5.2013.

Sharma, O., Lewis, J., Miller, A., Dearle, A., Balasubramaniam, D., Morrison, R. & Sventek, J. 2009, *Towards Verifying Correctness of Wireless Sensor Network Applications Using Insense and Spin*, Proceedings of the 16th International SPIN Workshop on Model Checking SoftwareSpringer-Verlag, Berlin, Heidelberg, pp. 223.

Silberschatz, A. 2009, *Operating system concepts*, 8th edn, John Wiley, Hoboken, N.J.

Single User Mode Definition [The Linux Information Project] 2006. Available: http://www.linfo.org/single_user_mode.html Updated 2013. Referred 1.6.2013.

Sommerville, I. 2011, *Software engineering*, 9th edn, Pearson, Boston, Mass., London.

Stallings, W. 2009, *Operating Systems: Internals and Design Principles*, 6/E, Pearson Education India.

time(1) - Linux man page [die.net] 2013. Available: http://linux.die.net/man/1/time. Referred 10.4.2013.

Understanding the pros and cons of the Waterfall Model of software development [TechRepublic] 2006. Available: http://www.techrepublic.com/article/understanding-the-pros-and-cons-of-the-waterfall-model-of-software-development/6118423. Updated 22.9.2006. Referred 3.2.2013.

User mnicky [Stack Overflow] 2013. Available: http://stackoverflow.com/users/626431/mnicky. Updated 8.6.2013. Referred 8.6.2013.

Valgrind's homepage 2012. Available: http://valgrind.org. Referred 16.6.2013.

Vaswani, R. & Zahorjan, J. 1991, *The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors*, SIGOPS Oper.Syst.Rev., vol. 25, no. 5, pp. 26-40.

Wall, D.W. 1991, *Limits of instruction-level parallelism*, ACM.

Wilson, P.R., Johnstone, M.S., Neely, M. & Boles, D. 1995, *Dynamic storage allocation: A survey and critical review* in Memory Management Springer, pp. 1-116.

Wulf, W.A. & McKee, S.A. 1995, *Hitting the memory wall: implications of the obvious*, ACM SIGARCH computer architecture news, vol. 23, no. 1, pp. 20-24.

Yang, B. 2010, *Assign Processor Affinity to Improve Performance* [Tune Up Blog]. Available: http://blog.tune-up.com/windows-insights/assign-processor-affinity-to-improve-performance. Updated 27.8.2010. Referred 31.5.2013.

APPENDICES

APPENDIX 1: List of Figures

Figure 3.1 Three components connected by two channels. 5

Figure 3.2 Topology with two components connected by a single channel. 6

Figure 3.3 Topology with five components connected by nine channels. 7

Figure 3.4. "Hello World" program written with Insense language. 8

Figure 3.5 Connection topologies supported by channels in Insense. 9

Figure 3.6 Insense application with a temperature sensor and two channels. 10

Figure 3.7 Declaration of a 3-dimensional array in Insense. 10

Figure 3.8 Creation of a structure and an instance of it. 11

Figure 3.9 Function declaration in Insense. 11

Figure 4.1 The evolutionary development model. 15

Figure 7.1 Send and Receive algorithm proposed by (Sharma et al. 2009). 19

Figure 7.2 Header prepended to allocated memory (in the shared heap). 20

Figure 7.3 Dynamically allocated memory for three components inside of the shared heap. 21

Figure 7.4 Dynamically allocated memory for three components inside of private heaps. 22

Figure 7.5 Retrieving data from directly mapped cache. 24

Figure 7.6 Cache misses with the "shared heap" scheme. 25

Figure 7.7 Cache misses with the "private heaps" scheme. 26

Figure 7.8 40 threads linked to Insense components dynamically placed on 4 cores. 27

Figure 7.9 40 threads linked to Insense components statically placed with the "Round-Robin" algorithm. 28

Figure 7.10 40 threads linked to Insense components statically placed on a single core. 28

Figure 7.11 Component connectivity patterns. 30

Figure 7.12 11 threads linked to Insense components statically placed on cores based on the nature of communication with other threads. 31

Figure 8.1 The "Hello" program written in Insense. 32

Figure 8.2 The supertype structure for Insense components. 33

Figure 8.3 A part of `component_create` where initialisation of the structure for storing metadata describing the component and creation of a private heap take place. 34

Figure 8.4 The wrapper function used to pass three parameters to the function run inside of a POSIX thread. 34

Figure 8.5 The function for stopping Insense components. 35

Figure 8.6 (a): "Hello" program that illustrates management of components. **(b)**: timeline of calls of functions dealing with management of Insense components during execution of the "Hello" program. 35

Figure 8.7 Adapted to multi-core architecture `array_loc` function. 36

Figure 8.8 Structure used to describe Insense channels on Unix. 37

Figure 8.9 Function responsible for creation of half-channels. 38

Figure 8.10 Function responsible for binding two half-channels into one complete channel. 39

Figure 8.11 Function responsible for destroying channels. 40

Figure 8.12 (a): "Hello" program that illustrates channels. **(b)**: timeline of calls of functions dealing with channels during execution of the "Hello" program. 41

Figure 8.13 The `main` function in Insense runtime. 42

Figure 8.14 The `primodial_main` function generated for the "Hello" Insense runtime. 43

Figure 8.15 Insense memory header structure adjusted for the 64-bit architecture. 44

Figure 8.16 (a): "Hello" program that illustrates memory management and garbage collection. **(b)**: descriptions of behaviour of the garbage collector 44

Figure 8.17 Structure that describes the thread-safe linked list. 45

Figure 8.18 Thread-safe "add" operation in the linked list. 46

Figure 8.19 Command to "make" and run the "Hello" Insense program on Linux. 47

Figure 8.20 Output of the "Hello" program after successful compilation and execution. 47

Figure 8.21 (a): "Hello" program that illustrates allocation of memory with the shared heap. **(b)**: timeline of allocation of data 48

Figure 8.22 Compiled code of the "Hello" example program dealing with allocation of memory for a channel and a local variable. 49

Figure 8.23 Modified and extended function for dynamically allocating memory in Insense. 50

Figure 8.24 Allocation of memory required by the "Hello" program in the shared heap. 51

Figure 8.25 Allocation of memory required by the “Hello” program in two private heaps. 52

Figure 8.26 Function for creation of a new private heap. 53

Figure 8.27 A structure that stores meta information for a new private heap. 54

Figure 8.28 Initialisation of a structure that stores meta information for a new private heap. 55

Figure 8.29 Allocation of memory into a private heap assigned to the current POSIX thread. 56

Figure 8.30 Allocation of memory with `SH_alloc_at_base`. 57

Figure 8.31 (a): “Hello” program that illustrates dynamic thread affinity. **(b)**: setting affinity to instances of Insense components on a machine with 4 cores. 59

Figure 8.32 Functions used for setting affinity of threads. 60

Figure 8.33 Part of `component_create()` function responsible for changing affinity of the newly created POSIX thread. 61

Figure 9.1 Results of running an experiment with utilising multiple heaps for (de-)allocation of memory. One component. 64

Figure 9.2 Results of running an experiment with utilising multiple heaps for (de-)allocation of memory. Four component instances and size of array: 1000x5 elements. 65

Figure 9.3 Running out of memory with an array of size 850x5 elements. 66

Figure 9.4 Amount of cache misses with different memory and thread placement schemes. 68

Figure 9.5 Execution time of `test_cache` program with an array of size 850 by 5 and 500,000 `behaviour` cycles. 69

Figure 9.6 Execution time of `test_cache` program with an array of size 220 by 5 and 500,000 `behaviour` cycles. 69

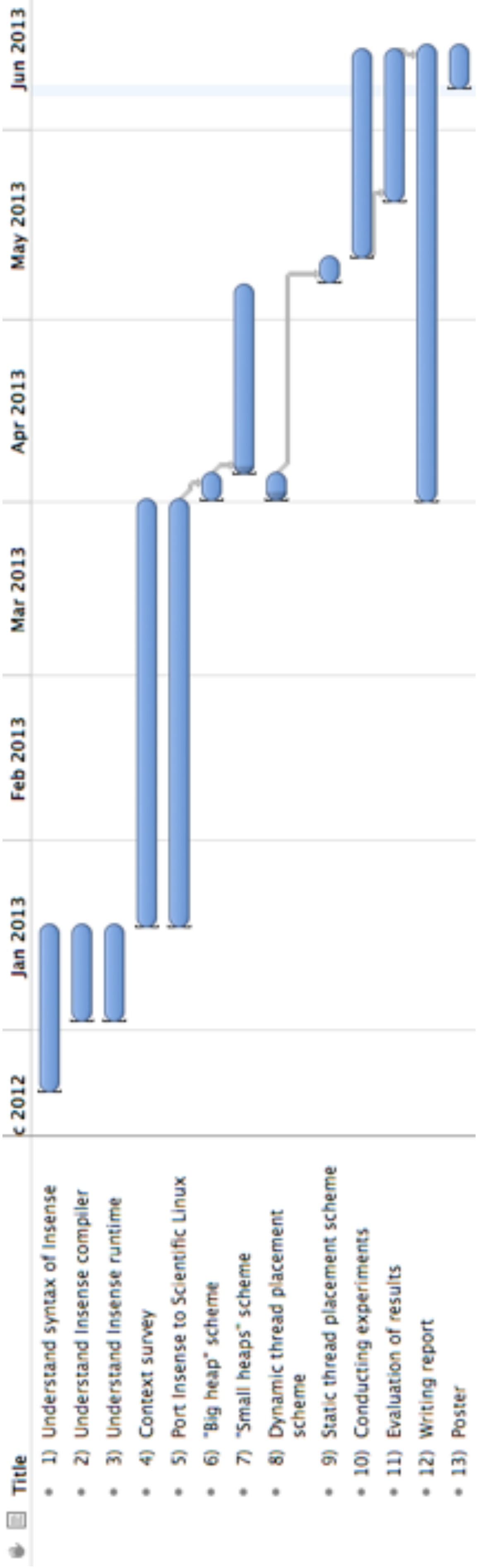
Figure 9.7 Results from “*Sender - Receiver*” example with two senders and two receivers. 73

Figure 9.8 Results from “*Client - Server*” example with 12 clients and 3 servers. 74

Figure 9.9 Results from “*Dispatch - Worker*” example with 3 dispatch instance and 12 workers. 75

Figure 9.10 Results from "*No communication*" example with 4 workers and two sizes of the array. 76

APPENDIX 2: Gantt Chart for the Project

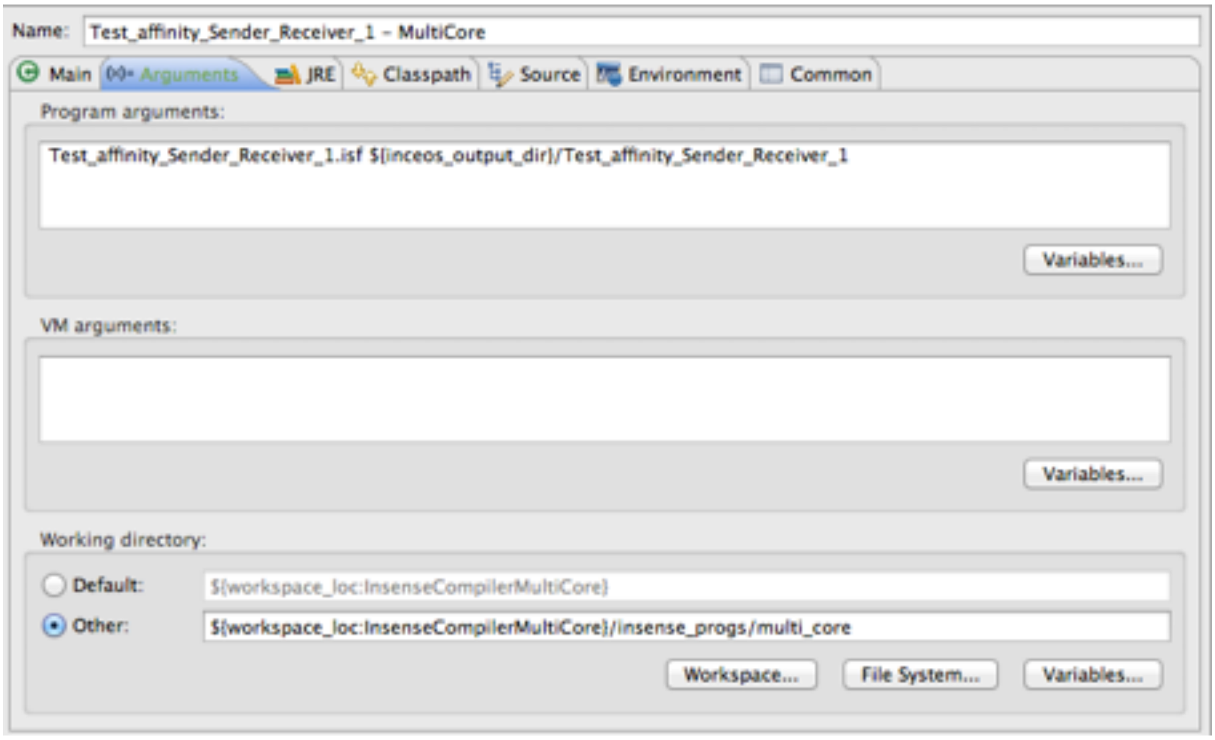


APPENDIX 3: Instruction on Executing Insense Programs

In order to run Insense program one needs to follow the following steps:

1. Obtain multi-core Insense compiler and runtime for a Unix-based machine (e.g. Scientific Linux). The runtime may be downloaded from https://github.com/Hollgam/Multi-Core-Insense-Runtime. And, the compiler may be found at https://github.com/Hollgam/Multi-Core-Insense-Compiler.

2. Compile the Insense program using the aforementioned multi-core Insense compiler. It can be achieved with help of Eclipse IDE:



Name: Test_affinity_Sender_Receiver_1 – MultiCore

Program arguments:

Test_affinity_Sender_Receiver_1.isf ${inceos_output_dir}/Test_affinity_Sender_Receiver_1

VM arguments:

Working directory:

○ Default: ${workspace_loc:InsenseCompilerMultiCore}

● Other: ${workspace_loc:InsenseCompilerMultiCore}/insense_progs/multi_core

Figure: Example of arguments used for compiling an Insense program with Eclipse IDE.

3. Make the compiled program into an executable with the `make` command in Terminal.

Note: a **configuration** of memory and thread placement schemes is defined in the file `GlobalVars.h` in runtime. To use a different scheme one needs to change values of `HEAPS` and `AFFINITY_ALGO` constants.

APPENDIX 4: Creation of a New Component

```
void *component_create(behaviour_ft behaviour, int struct_size, int
stack_size, int argc, void *argv[], int core) {
    struct IComponent_data *this_ptr; // Define thread
    // Allocate space for the struct, create a new private heap (if required)
#if HEAPS // Private Heaps
```

```
    // Passing 2 to indicate that a new heap needs to be created
    void * newHeap = ((struct IComponent_data *) DAL_alloc(0, 2));
    // Create a new element to be put to the map
    struct shMapType * heapElement = SH_init(newHeap, NULL );
    // Allocate space for this_ptr in the newly created private heap
    this_ptr = SH_alloc_at_base(struct_size, heapElement);
    if (this_ptr == NULL ) {
        return NULL ;
    } else {
        memset(this_ptr, 0, struct_size);
    }
#else // Shared Heap
    if ((this_ptr = ((struct IComponent_data *) DAL_alloc(struct_size, true))) == NULL ) {
        return NULL;
    } else {
        memset(this_ptr, 0, struct_size);
    }
#endif
    // Initialize this->comp_create_sem sem_init call.
    sem_init(&(this_ptr->component_create_sem), 0, 0);

    // Setup the stopped condition
    if (struct_size) {
        struct IComponent_data *t = (struct IComponent_data*) this_ptr;
        t->stopped = 0;
    }

    // Define new structure for arguments for the wrapper function
    // Whenever dealing with garbage collection, first define as NULL
    struct argStructType * argStruct = malloc(sizeof(struct argStructType));
    argStruct->behaviour = behaviour;
    argStruct->argc = argc;
    argStruct->argv = argv;
    argStruct->this_ptr = this_ptr;

    // Create thread
#if HEAPS // Private Heaps
    // Lock mutex to avoid problems with scheduling
    pthread_mutex_lock(&thread_lock); #endif
    // Create a POSIX thread
    pthread_create(&this_ptr->behav_thread, NULL, startRoutine, argStruct);
    //Set affinity
#if !(AFFINITY_ALGO == 0)
    if (core != -1) { // Manually passed Core ID
        setAffinityToCore(this_ptr->behav_thread, core);
    } else { // Core ID was not passed to the component_create function
        // Use an algorithm defined in GlobalVars.h
        setAffinity(this_ptr->behav_thread);
    }
    // Check if setting affinity worked.
    getAffinityThread(this_ptr->behav_thread);
#endif

    // If private heaps are used new entry in the map with a pointer to a newly created pthread.
#if HEAPS // Private Heaps
    heapElement->thread_id = this_ptr->behav_thread; // Put a thread id to the element to be to the map
    listAdd(SHList, heapElement);
    pthread_mutex_unlock(&thread_lock); // Unlock mutex
```

```
#endif
    // Insert thread into the list of threads
    listAdd(threadList, this_ptr->behav_thread);
    // Wait for creation of the component
    sem_wait(&this_ptr->component_create_sem);
    return this_ptr;
}
```

APPENDIX 5: Deallocation of Memory for Private Heaps

```
void SH_free(void *ptr) {
    //First check if memory was allocated using malloc()
    void * adr = listGetMallocedMemoryAdr(mallocList, ptr);
    if (adr != NULL ) {
        free(adr); // Memory was allocated using malloc(), use free instead
        listRemove(mallocList, adr); // Remove address from the list
        return;
    }

    // Else, find which private heap it was put into.
    struct shMapType * shMapEntry = listGetMemoryLocation(SHList, pthread_self());
    unsigned int toFree;  // Pointer to block that needs to be freed
    unsigned int cur, prev;

    toFree = ((unsigned int *) ptr - (shMapEntry->memArea + 1));

    // If block, that is being freed is before the first free block
    if (toFree < shMapEntry->available) {
    // If next free block is immediately after block that is being freed
       if (((refToNextBlock(toFree, shMapEntry->memArea) + 1) == shMapEntry->available) && shMapEntry->available < shMapEntry->memAreaSize)
            shMapEntry->memArea[toFree] += (shMapEntry->memArea[shMapEntry->available] + 1);  // Defragmentation of free space

        else
              shMapEntry->memArea[refToNextBlock(toFree, shMapEntry->memArea)] = shMapEntry->available;

        shMapEntry->available = toFree;
    }

    // If block, that is being freed isn't before the first free block
    else {
        prev = cur = shMapEntry->available;

        while (cur < toFree) {
            prev = cur;
            cur = nextBlock(cur, shMapEntry->memArea);
        }
    // If previous free block is immediately before block that is being freed
        if ((refToNextBlock(prev, shMapEntry->memArea) + 1) == toFree) {

            shMapEntry->memArea[prev] += (shMapEntry->memArea[toFree] + 1); // Defragmentation of free space

    // If next free block is immediately after block that is being freed
            if (((refToNextBlock(toFree, shMapEntry->memArea) + 1) == cur) && cur < shMapEntry->memAreaSize)
                shMapEntry->memArea[prev] += (shMapEntry->memArea[cur] + 1); // Defragmentation of free space
            else
                shMapEntry->memArea[refToNextBlock(toFree, shMapEntry->memArea)] = cur;
        } else {
            shMapEntry->memArea[refToNextBlock(prev, shMapEntry->memArea)] = toFree;
            shMapEntry->memArea[refToNextBlock(toFree, shMapEntry->memArea)] = cur;
        }
    }
}
```

APPENDIX 6: Test_alloc Program Used to Evaluate Effects of Using Multiple Heaps for (De-)allocation of Memory

```
type IComp is interface () // Interface for Comp component

component Comp presents IComp {
    runTime = 5000   // Amount of times behaviour is to be executed.
    count = 0        // Used to track number of times behaviour has executed.

    constructor() {}

    behaviour {
        a = new integer[1000][5] of 0  // 5002 allocation take place

        count := count + 1 // Increment a counter of executions of behaviour
        // Show percentage done: 50% - 100%
        if (count % (runTime / 2) == 0) then {
            printInt(100 / (runTime / count))
            printString("% done.\n")
        }
        // If a limit of execution has been exceeded - stop
        if (count >= runTime) then {
            stop
        }
    }
}

// Create and test components
comp1 = new Comp()
comp2 = new Comp()
comp3 = new Comp()
comp4 = new Comp()
```

APPENDIX 7: Test_cache Program Used to Evaluate Effects of Memory Management Schemes on Cache Usage

```
type CompCache is interface()

component Comp presents CompCache {
    // 250*28+250*8+8 = 9008 Bytes of memory, < 32KB size of L1 cache
    a = new integer[250][5] of 0
    runTime = 50000      // Amount of times behaviour is to be executed.
    count = 0     // Used to track number of times behaviour has executed.

    proc incrementArray(){
        for i = 0 .. a.length-1 do {
            for j = 0 .. a [i].length-1 do {
                a[i][j] := a[i][j] + 1
            }
        }
    }

    constructor() { }

    behaviour {
        incrementArray()

        count := count + 1 // Increment a counter of executions of behaviour
        // Show percentage done: 50% - 100%
        if (count % (runTime / 2) == 0) then {
            printInt(100 / (runTime / count))
            printString("% done.\n")
        }
        // If a limit of execution has been exceeded - stop
        if (count >= runTime) then {
            stop
        }
    }
}

// Insense main
comp1 = new Comp()
comp2 = new Comp()
comp3 = new Comp()
comp4 = new Comp()
```

APPENDIX 8: Programs Used to Evaluate Effects of Thread Placement and Communication Between Components on Performance

Appendix 8.1: “Sender - Receiver” Scheme

```
type IReceiver is interface( in integer input )
type ISender is interface( out integer output )

component Sender presents ISender {
    runTime = 100    // Amount of times behaviour is to be executed.
    count = 0        // Used to track number of times behaviour has executed.

    constructor(){ }

    behaviour {
        send 0 on output // First send an int to a server

        count := count + 1 // Increment a counter of executions of behaviour
        // Show percentage done: 50% - 100%
        if (count % (runTime / 2) == 0) then {
            printInt(100 / (runTime / count))
            printString("% done.\n")
        }
        // If a limit of execution has been exceeded - stop
        if (count >= runTime) then {
            stop
        }
    }
}

component Receiver presents IReceiver {
    runTime = 100    // Amount of times behaviour is to be executed.
    count = 0        // Used to track number of times behaviour has executed.

    constructor(){ }

    behaviour {
        receive blank from input // First receive an int from a client

        count := count + 1 // Increment a counter of executions of behaviour
        // Show percentage done: 50% - 100%
        if (count % (runTime / 2) == 0) then {
            printInt(100 / (runTime / count))
            printString("% done.\n")
        }
        // If a limit of execution has been exceeded - stop
        if (count >= runTime) then {
            stop
        }
    }
}

//Insense main
sender1 = new Sender()
sender2 = new Sender()
receiver1 = new Receiver()
receiver2 = new Receiver()

// Connect senders and receiver with channels.
connect sender1.output to receiver1.input
connect sender2.output to receiver2.input
```

Appendix 8.2: “Client - Server” Scheme

```
type IServer is interface( in integer input )
type IClient is interface( out integer output )

component Client presents IClient {
    runTime = 100    // Amount of times behaviour is to be executed.
    count = 0        // Used to track number of times behaviour has executed.
    calcTime = 100   // Times a server is requested to perform computation.

    constructor(){ }

    behaviour {
        send calcTime on output // First send an int to a server

        count := count + 1 // Increment a counter of executions of behaviour
        // Show percentage done: 50% - 100%
        if (count % (runTime / 2) == 0) then {
            printInt(100 / (runTime / count))
            printString("% done.\n")
        }
        // If a limit of execution has been exceeded - stop
        if (count >= runTime) then {
            stop
        }
    }
}
```

```

component Server presents IServer {
    a = new integer[1000][5] of 0 // Array used for computations.
    runTime = 400    // Amount of times behaviour is to be executed.
    count = 0        // Used to track number of times behaviour has executed.

    // Increment elements of an array "num" times
    proc incrementArray(integer num) {
        for i = 0 .. num do {
            for j = 0 .. a.length-1 do {
                for k = 0 .. a[j].length-1 do {
                    a[j][k] := a[j][k] + 1
                }
            }
        }
    }

    constructor(){ }

    behaviour {
        receive calcTime from input // First receive an int from a client

        incrementArray(calcTime) // Perform calculations

        count := count + 1 // Increment a counter of executions of behaviour
        // Show percentage done: 50% - 100%
        if (count % (runTime / 2) == 0) then {
            printInt(100 / (runTime / count))
            printString("% done.\n")
        }
        // If a limit of execution has been exceeded - stop
        if (count >= runTime) then {
            stop
        }
    }
}

//Insense main
server1 = new Server()
server2 = new Server()
server3 = new Server()
client1 = new Client()
client2 = new Client()
client3 = new Client()
client4 = new Client()
client5 = new Client()
client6 = new Client()
client7 = new Client()
client8 = new Client()
client9 = new Client()
client10 = new Client()
client11 = new Client()
client12 = new Client()

// Connect clients and servers with channels.
connect client1.output to server1.input
connect client2.output to server1.input
connect client3.output to server1.input
connect client4.output to server1.input
connect client5.output to server2.input
connect client6.output to server2.input
connect client7.output to server2.input
connect client8.output to server2.input
connect client9.output to server3.input
connect client10.output to server3.input
connect client11.output to server3.input
connect client12.output to server3.input
```

Appendix 8.3: “Dispatch - Worker” Scheme

```
type IWorker is interface( in integer input )
type IDispatch is interface( out integer output )

component Dispatch presents IDispatch {
    runTime = 300    // Amount of times behaviour is to be executed.
    count = 0        // Used to track number of times behaviour has executed.
    calcTime = 100   // Times a server is requested to perform computation.

    constructor(){ }

    behaviour {
        send calcTime on output // First send an int to a server

        count := count + 1 // Increment a counter of executions of behaviour
        // Show percentage done: 50% - 100%
        if (count % (runTime / 2) == 0) then {
            printInt(100 / (runTime / count))
            printString("% done.\n")
        }
        // If a limit of execution has been exceeded - stop
        if (count >= runTime) then {
            stop
        }
    }
}

component Worker presents IWorker {
    a = new integer[1000][5] of 0 // Array used for computations.
    runTime = 100    // Amount of times behaviour is to be executed.
    count = 0        // Used to track number of times behaviour has executed.

    // Increment elements of an array "num" times
    proc incrementArray(integer num) {
        for i = 0 .. num do {
            for j = 0 .. a.length-1 do {
                for k = 0 .. a[j].length-1 do {
                    a[j][k] := a[j][k] + 1
                }
            }
        }
    }

    constructor(){ }

    behaviour {
        receive calcTime from input // First receive an int from a client

        incrementArray(calcTime) // Perform calculations

        count := count + 1 // Increment a counter of executions of behaviour
        // Show percentage done: 50% - 100%
        if (count % (runTime / 2) == 0) then {
            printInt(100 / (runTime / count))
            printString("% done.\n")
        }
        // If a limit of execution has been exceeded - stop
        if (count >= runTime) then {
            stop
        }
    }
}
//Insense main
dispatch1 = new Dispatch()
dispatch2 = new Dispatch()
dispatch3 = new Dispatch()
worker1 = new Worker()
worker2 = new Worker()
worker3 = new Worker()
worker4 = new Worker()
worker5 = new Worker()
worker6 = new Worker()
worker7 = new Worker()
worker8 = new Worker()
worker9 = new Worker()
worker10 = new Worker()
worker11 = new Worker()
worker12 = new Worker()

// Connect dispatch instances and workers with channels.
connect dispatch1.output to worker1.input
connect dispatch1.output to worker2.input
connect dispatch1.output to worker3.input
connect dispatch1.output to worker4.input
connect dispatch2.output to worker5.input
connect dispatch2.output to worker6.input
connect dispatch2.output to worker7.input
connect dispatch2.output to worker8.input
connect dispatch3.output to worker9.input
connect dispatch3.output to worker10.input
connect dispatch3.output to worker11.input
connect dispatch3.output to worker12.input
```

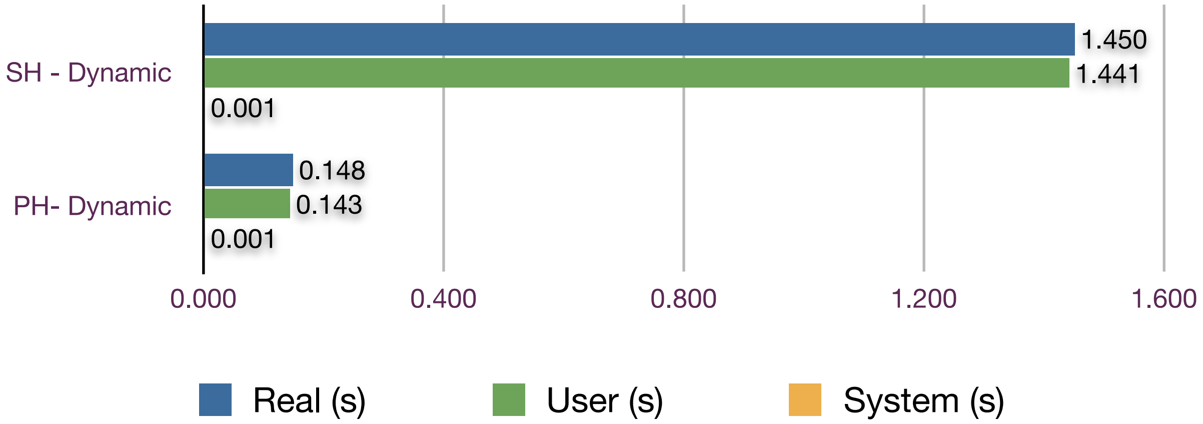
Appendix 8.4: “No communication” Scheme

```
type IWorker is interface( )

component Worker presents IServer {
    a = new integer[1000][5] of 0 // Array used for computations.
    runTime = 100    // Amount of times behaviour is to be executed.
    count = 0        // Used to track number of times behaviour has executed.

    // Increment elements of an array "num" times
    proc incrementArray(integer num) {
        b = 0
        for i = 0 .. num do {
            for j = 0 .. a.length-1 do {
                for k = 0 .. a[j].length-1 do {
                    a[j][k] := a[j][k] + 1
                }
            }
        }
    }

    constructor(){ }

    behaviour {
        incrementArray(1000) // Perform calculations

        count := count + 1 // Increment a counter of executions of behaviour
        // Show percentage done: 50% - 100%
        if (count % (runTime / 2) == 0) then {
            printInt(100 / (runTime / count))
            printString("% done.\n")
        }
        // If a limit of execution has been exceeded - stop
        if (count >= runTime) then {
            stop
        }
    }
}

//Insense main
worker1 = new Worker()
worker2 = new Worker()
worker3 = new Worker()
worker4 = new Worker()
```

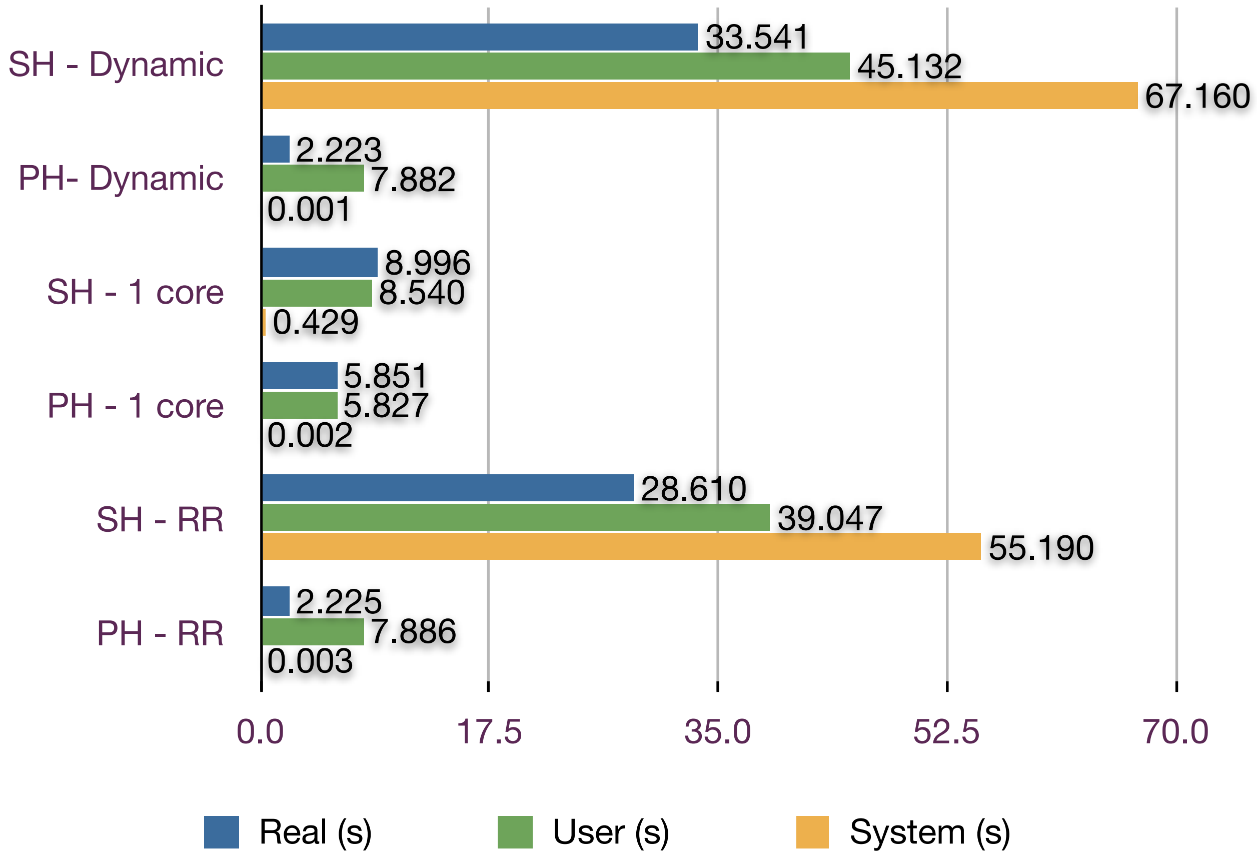
APPENDIX 9: Real, User, and System Time of Running Experiments

Notation used in this appendix is described in Section 9.1.3.

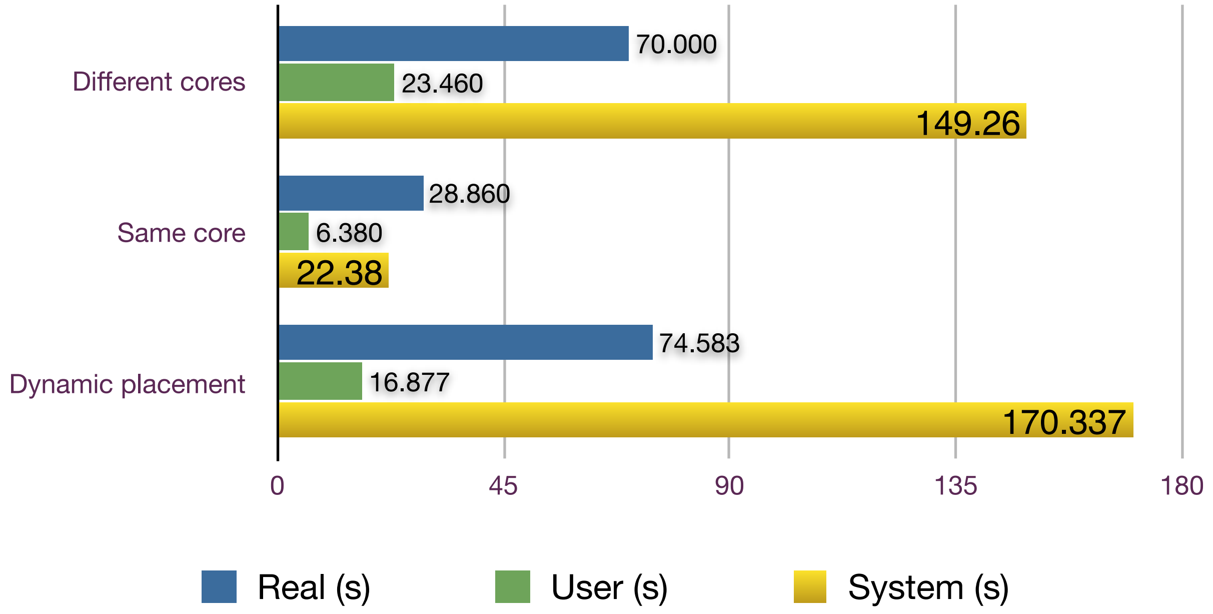
Appendix 9.1: Experiment with (De-)allocation of Memory, 1 Component



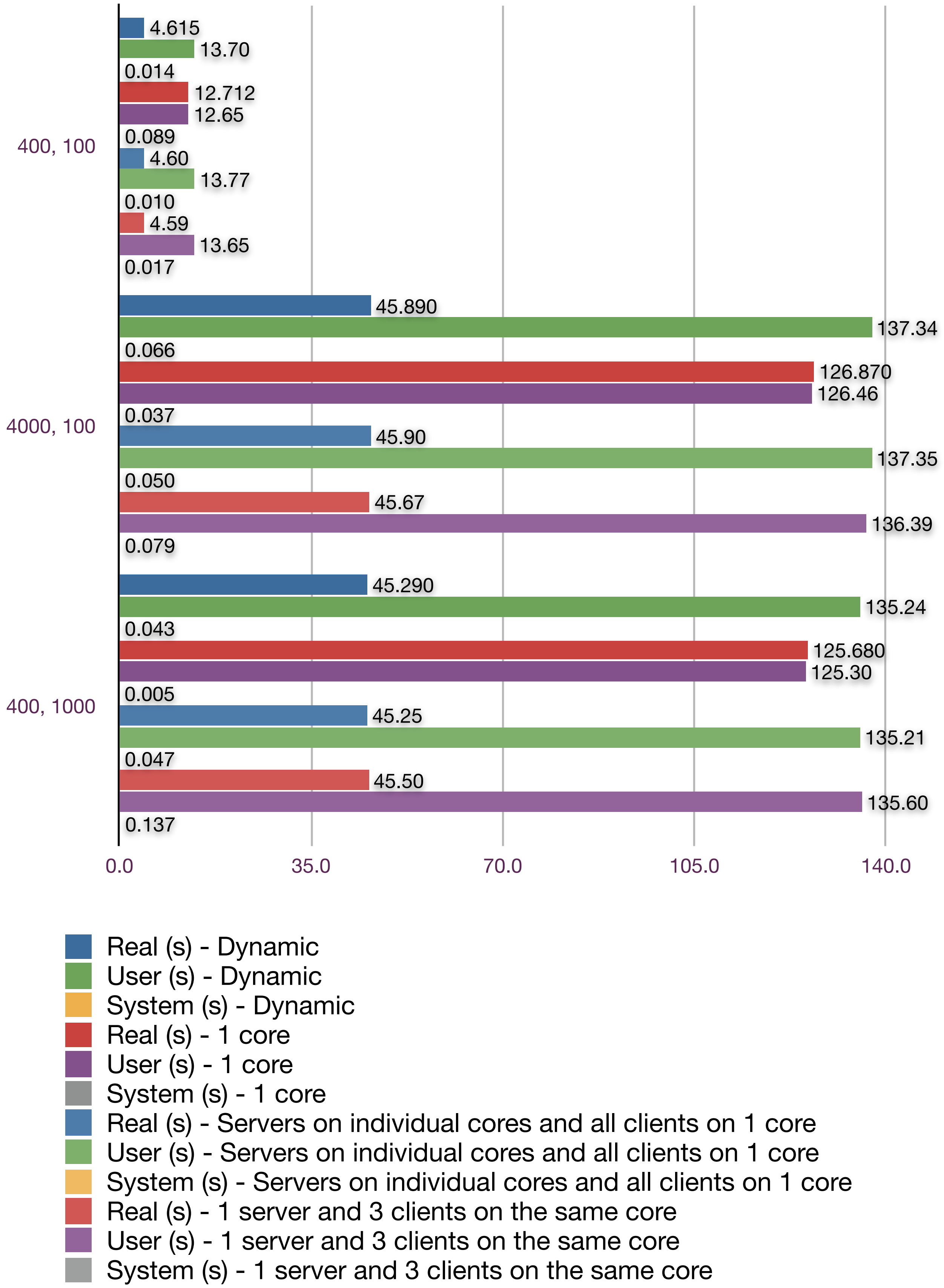
Appendix 9.2: Experiment with (De-)allocation of Memory, 4 components



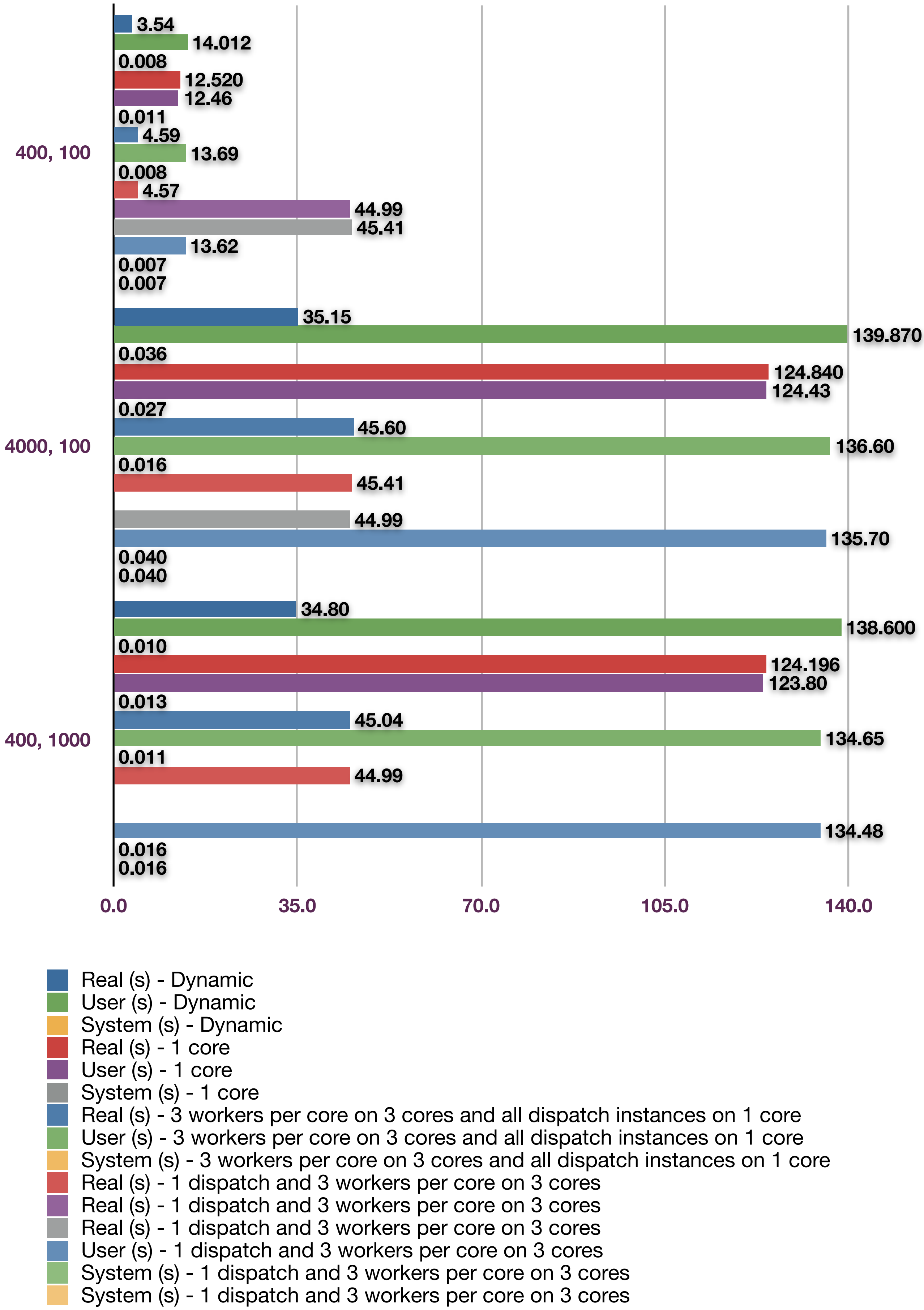
Appendix 9.3: Results from “Sender - Receiver” example with 2 senders and 2 receivers



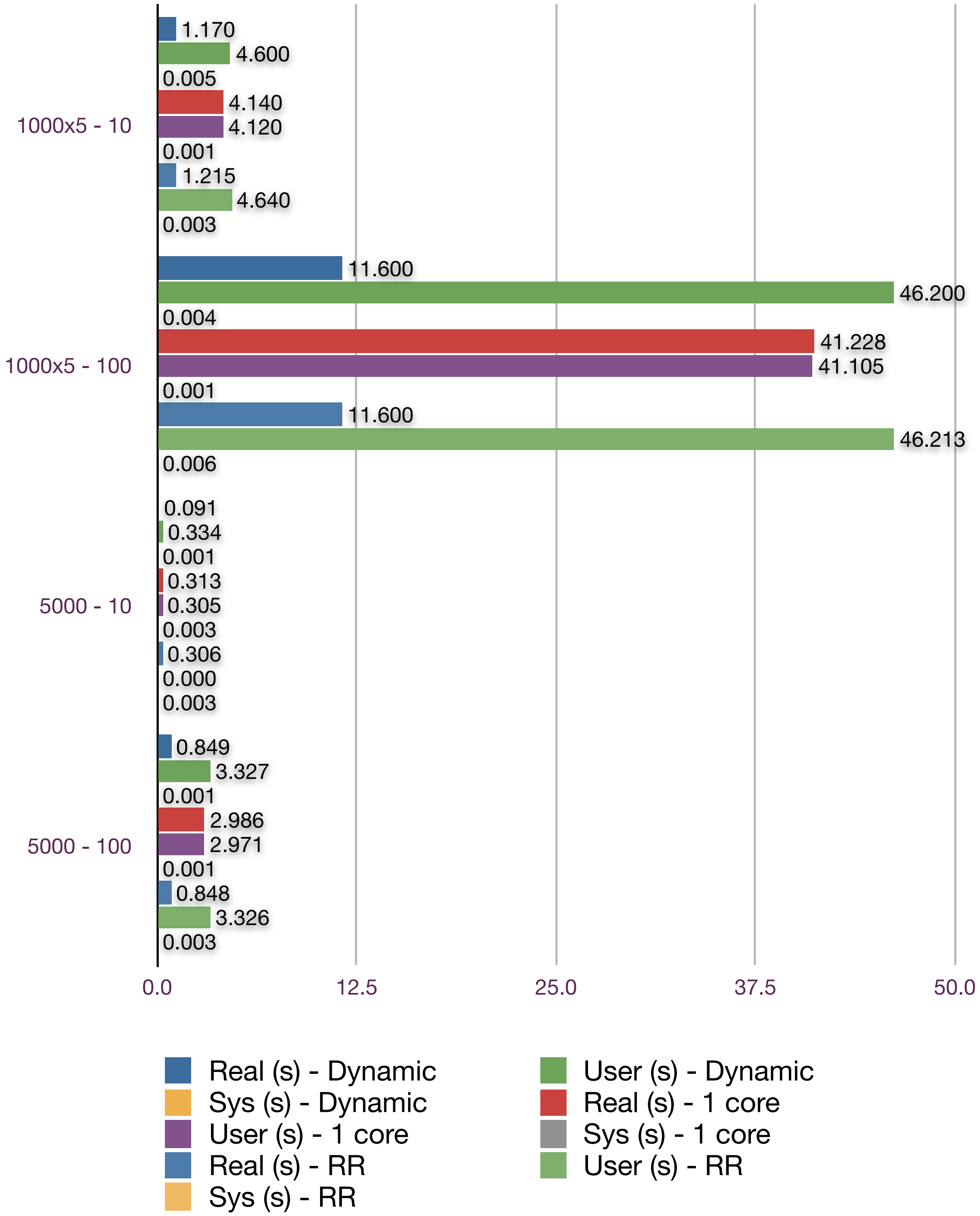
Appendix 9.4: Results from “Client - Server” example



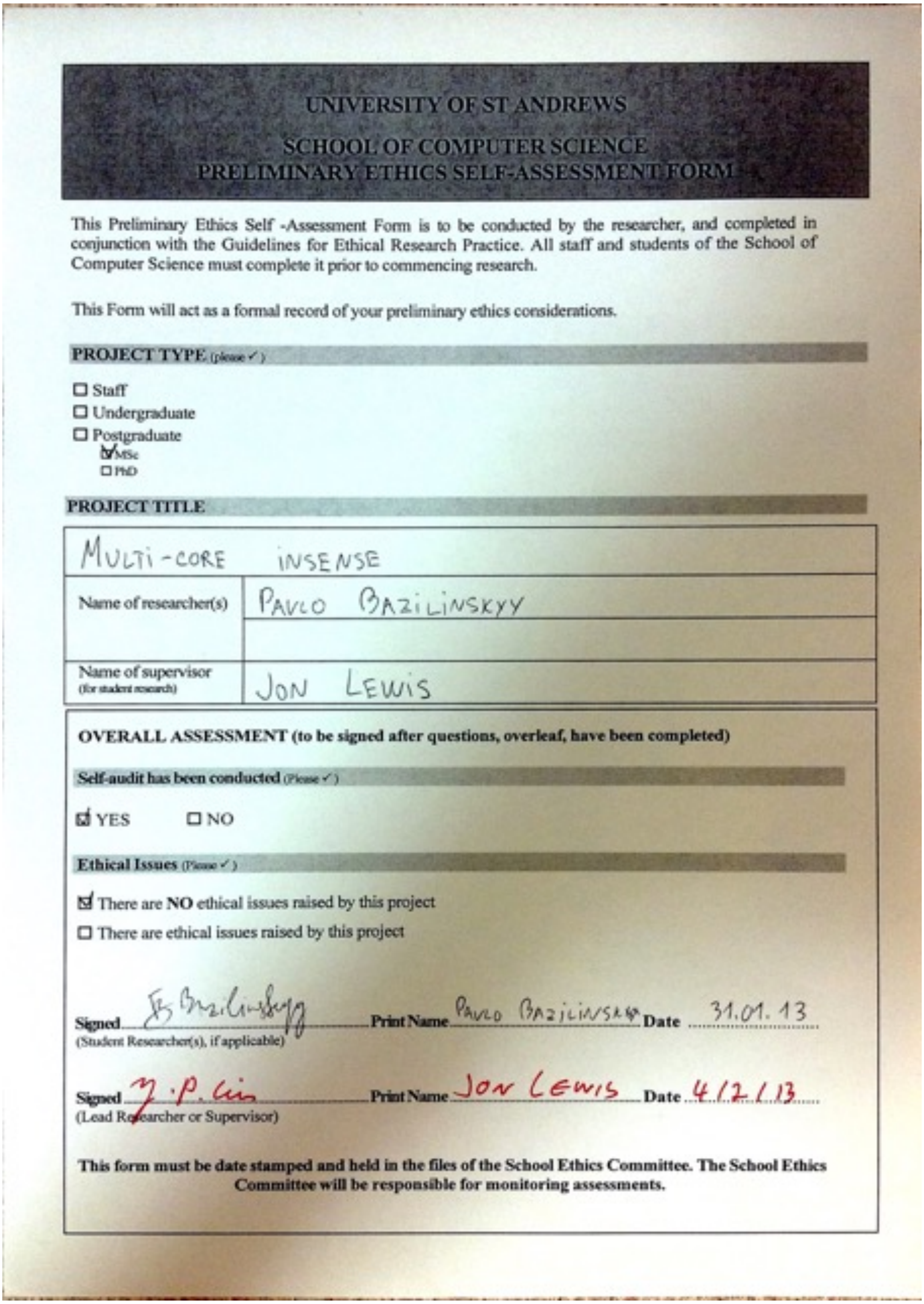
Appendix 9.5: Results from “Dispatch - Worker” example



Appendix 9.6 Results from "No communication" example with 4 workers and two sizes of the array



APPENDIX 10: Ethics approval form



UNIVERSITY OF ST ANDREWS

SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self -Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your preliminary ethics considerations.

PROJECT TYPE (please ✓)

□ Staff
□ Undergraduate
□ Postgraduate
  ☑ MSc
  □ PhD

PROJECT TITLE

MULTI-CORE INSENSE	
Name of researcher(s)	PAVLO BAZILINSKYY
Name of supervisor (for student research)	JON LEWIS

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self-audit has been conducted (Please ✓)

☑ YES □ NO

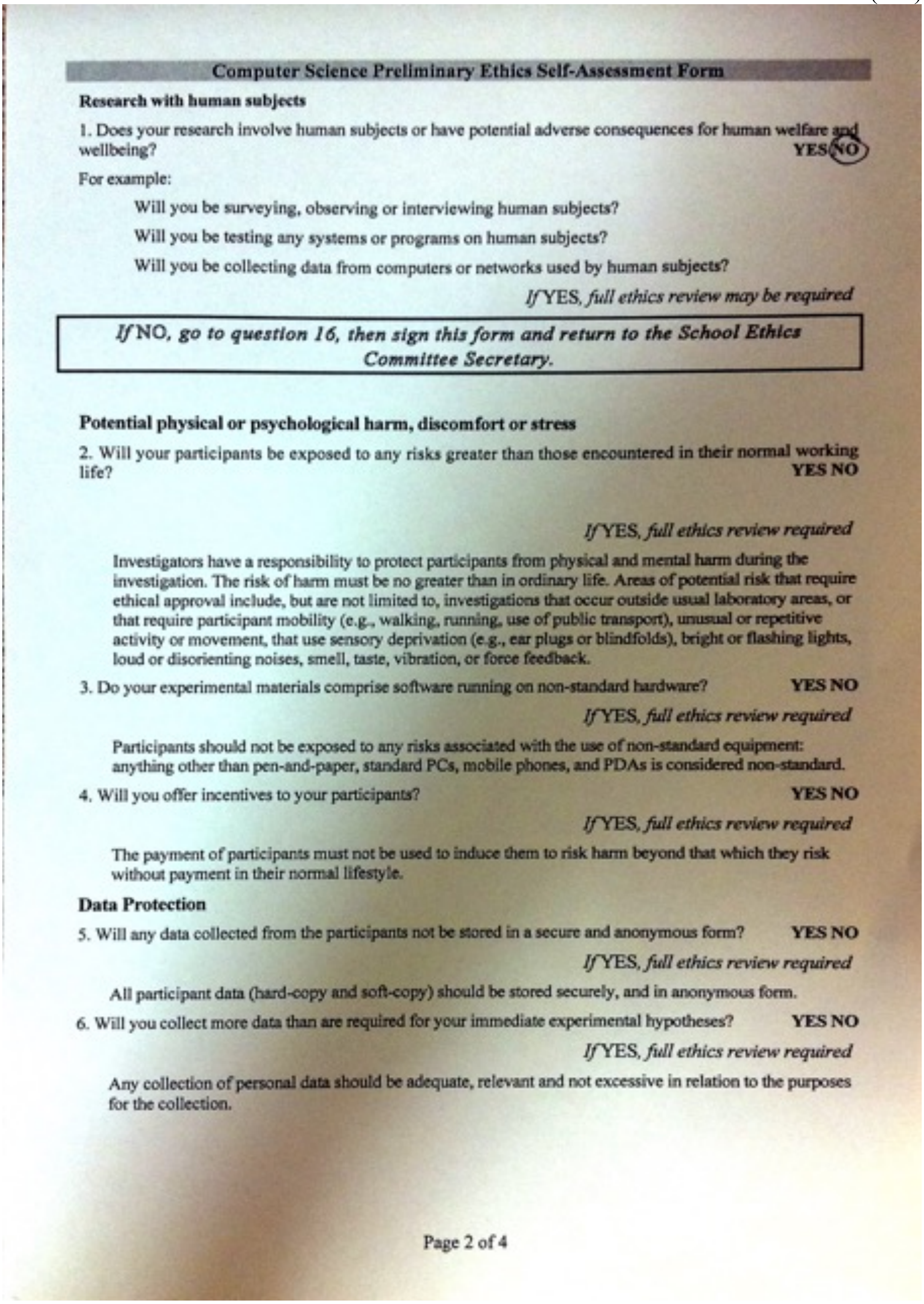
Ethical Issues (Please ✓)

☑ There are **NO** ethical issues raised by this project
□ There are ethical issues raised by this project

Signed .... Print Name PAVLO BAZILINSKYY Date 31.01.13
(Student Researcher(s), if applicable)

Signed .... Print Name JON LEWIS Date 4/2/13
(Lead Researcher or Supervisor)

This form must be date stamped and held in the files of the School Ethics Committee. The School Ethics Committee will be responsible for monitoring assessments.



Computer Science Preliminary Ethics Self-Assessment Form

Research with human subjects

1. Does your research involve human subjects or have potential adverse consequences for human welfare and wellbeing? **YES [✓ NO]**

For example:

Will you be surveying, observing or interviewing human subjects?

Will you be testing any systems or programs on human subjects?

Will you be collecting data from computers or networks used by human subjects?

If YES, *full ethics review may be required*

> *If* NO, *go to question 16, then sign this form and return to the School Ethics Committee Secretary.*

Potential physical or psychological harm, discomfort or stress

2. Will your participants be exposed to any risks greater than those encountered in their normal working life? **YES NO**

If YES, *full ethics review required*

Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life. Areas of potential risk that require ethical approval include, but are not limited to, investigations that occur outside usual laboratory areas, or that require participant mobility (e.g., walking, running, use of public transport), unusual or repetitive activity or movement, that use sensory deprivation (e.g., ear plugs or blindfolds), bright or flashing lights, loud or disorienting noises, smell, taste, vibration, or force feedback.

3. Do your experimental materials comprise software running on non-standard hardware? **YES NO**

If YES, *full ethics review required*

Participants should not be exposed to any risks associated with the use of non-standard equipment: anything other than pen-and-paper, standard PCs, mobile phones, and PDAs is considered non-standard.

4. Will you offer incentives to your participants? **YES NO**

If YES, *full ethics review required*

The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

Data Protection

5. Will any data collected from the participants not be stored in a secure and anonymous form? **YES NO**

If YES, *full ethics review required*

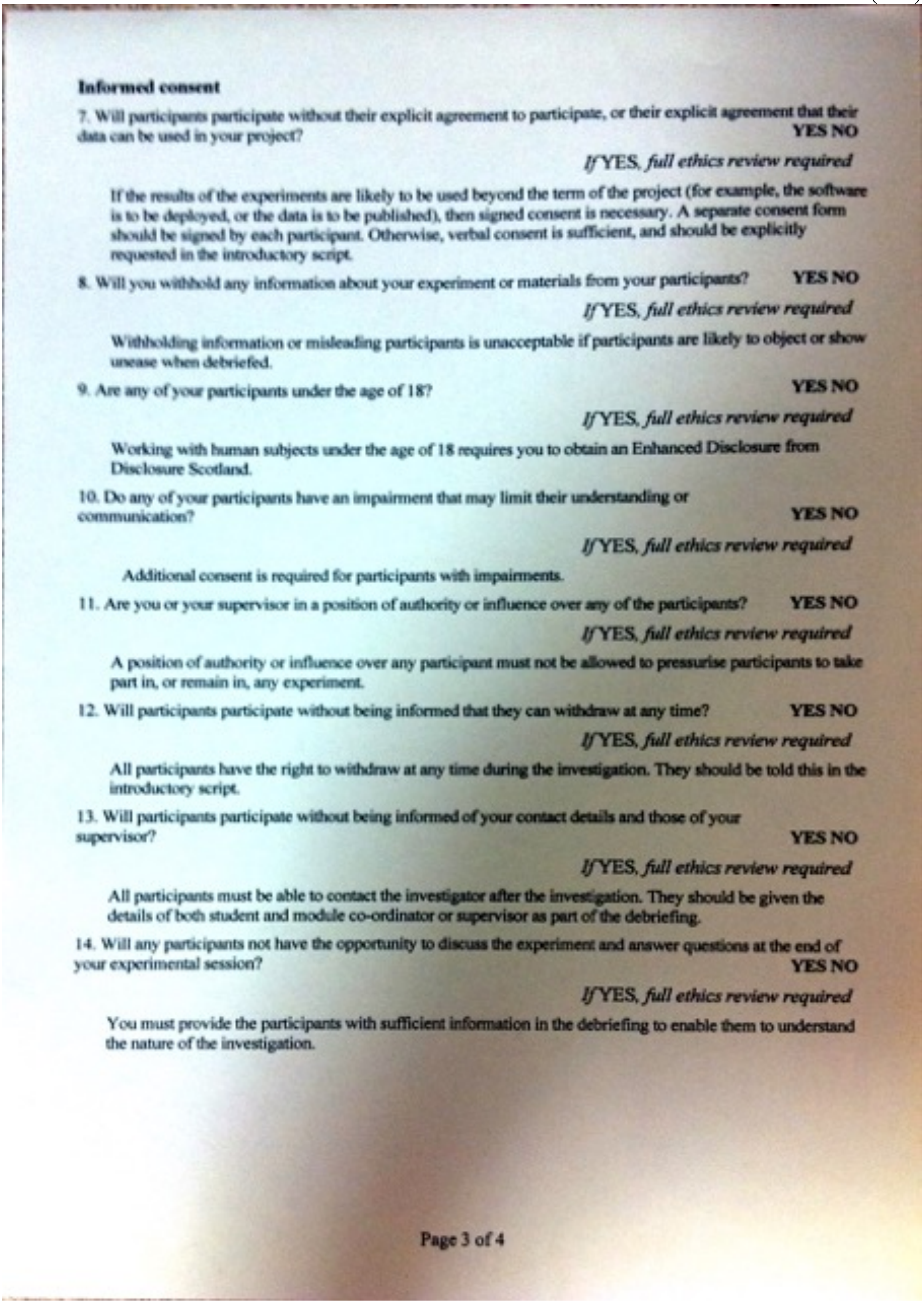
All participant data (hard-copy and soft-copy) should be stored securely, and in anonymous form.

6. Will you collect more data than are required for your immediate experimental hypotheses? **YES NO**

If YES, *full ethics review required*

Any collection of personal data should be adequate, relevant and not excessive in relation to the purposes for the collection.

Page 2 of 4



Informed consent

7. Will participants participate without their explicit agreement to participate, or their explicit agreement that their data can be used in your project? **YES NO**

If YES, full ethics review required

If the results of the experiments are likely to be used beyond the term of the project (for example, the software is to be deployed, or the data is to be published), then signed consent is necessary. A separate consent form should be signed by each participant. Otherwise, verbal consent is sufficient, and should be explicitly requested in the introductory script.

8. Will you withhold any information about your experiment or materials from your participants? **YES NO**

If YES, full ethics review required

Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

9. Are any of your participants under the age of 18? **YES NO**

If YES, full ethics review required

Working with human subjects under the age of 18 requires you to obtain an Enhanced Disclosure from Disclosure Scotland.

10. Do any of your participants have an impairment that may limit their understanding or communication? **YES NO**

If YES, full ethics review required

Additional consent is required for participants with impairments.

11. Are you or your supervisor in a position of authority or influence over any of the participants? **YES NO**

If YES, full ethics review required

A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.

12. Will participants participate without being informed that they can withdraw at any time? **YES NO**

If YES, full ethics review required

All participants have the right to withdraw at any time during the investigation. They should be told this in the introductory script.

13. Will participants participate without being informed of your contact details and those of your supervisor? **YES NO**

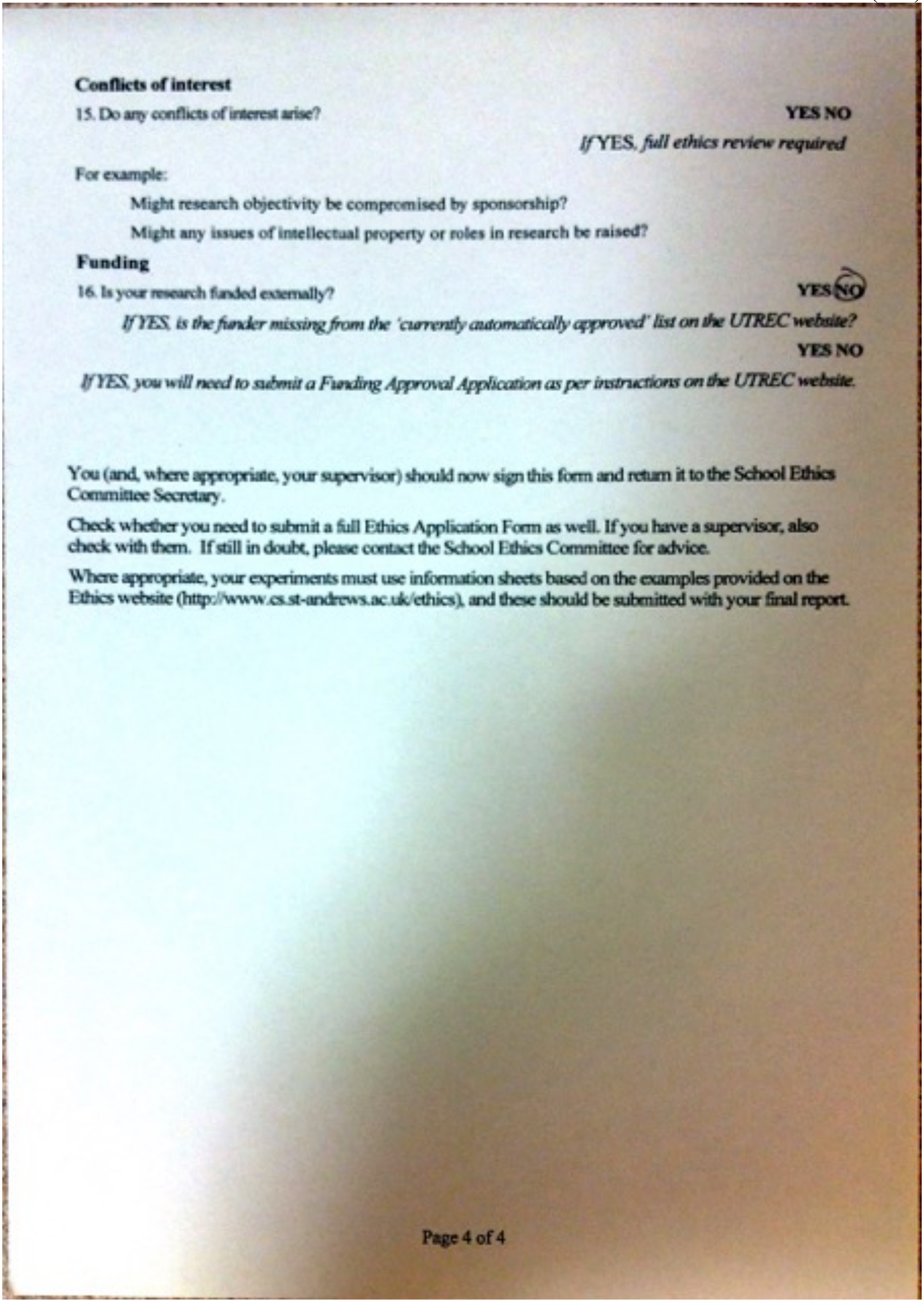
If YES, full ethics review required

All participants must be able to contact the investigator after the investigation. They should be given the details of both student and module co-ordinator or supervisor as part of the debriefing.

14. Will any participants not have the opportunity to discuss the experiment and answer questions at the end of your experimental session? **YES NO**

If YES, full ethics review required

You must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation.



Conflicts of interest

15. Do any conflicts of interest arise? **YES NO**

If YES, full ethics review required

For example:

Might research objectivity be compromised by sponsorship?

Might any issues of intellectual property or roles in research be raised?

Funding

16. Is your research funded externally? **YES [✓ NO]**

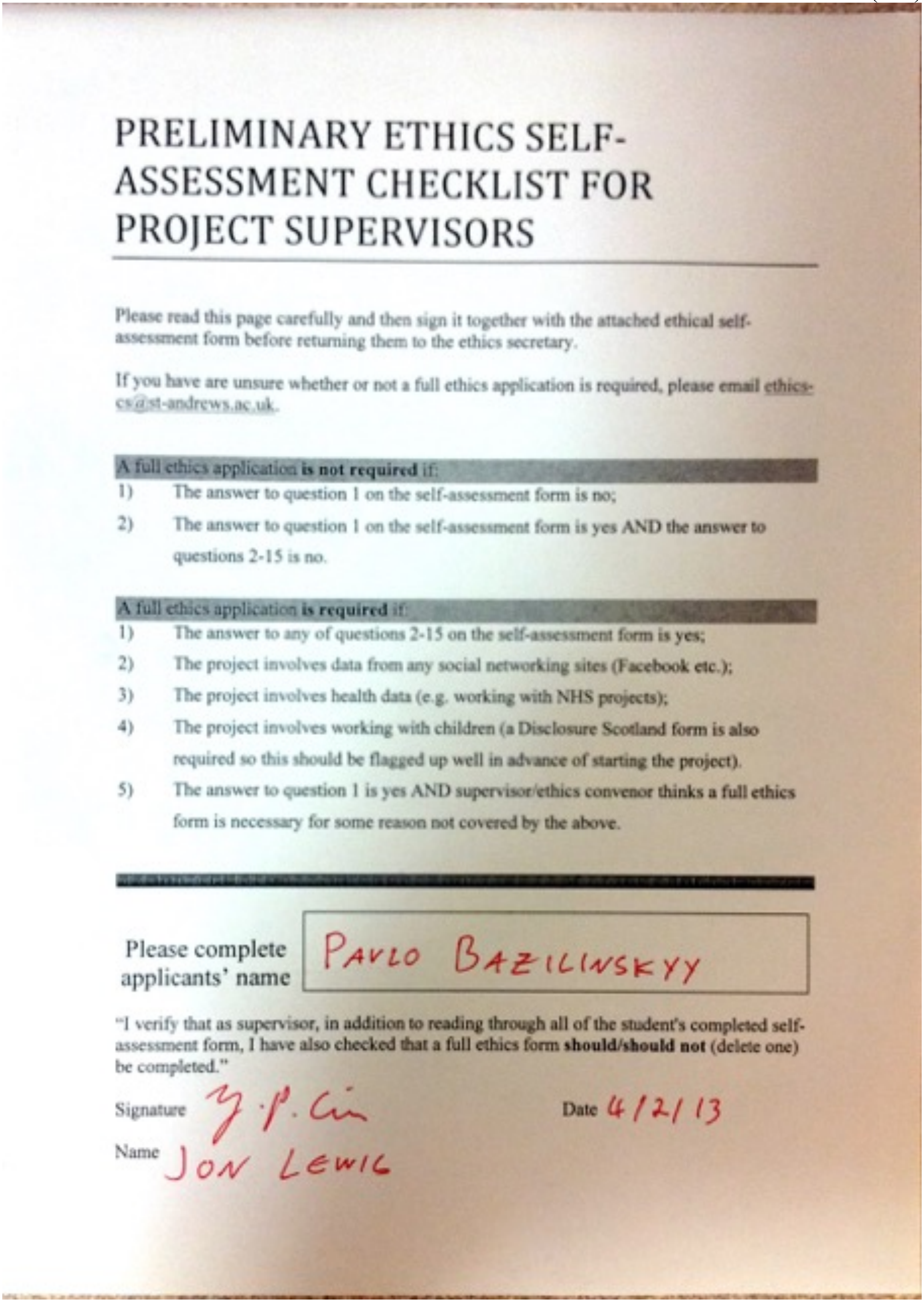
If YES, is the funder missing from the 'currently automatically approved' list on the UTREC website? **YES NO**

If YES, you will need to submit a Funding Approval Application as per instructions on the UTREC website.

You (and, where appropriate, your supervisor) should now sign this form and return it to the School Ethics Committee Secretary.

Check whether you need to submit a full Ethics Application Form as well. If you have a supervisor, also check with them. If still in doubt, please contact the School Ethics Committee for advice.

Where appropriate, your experiments must use information sheets based on the examples provided on the Ethics website (http://www.cs.st-andrews.ac.uk/ethics), and these should be submitted with your final report.



PRELIMINARY ETHICS SELF-ASSESSMENT CHECKLIST FOR PROJECT SUPERVISORS

Please read this page carefully and then sign it together with the attached ethical self-assessment form before returning them to the ethics secretary.

If you have are unsure whether or not a full ethics application is required, please email ethics-cs@st-andrews.ac.uk.

A full ethics application **is not required** if:

1) The answer to question 1 on the self-assessment form is no;
2) The answer to question 1 on the self-assessment form is yes AND the answer to questions 2-15 is no.

A full ethics application **is required** if:

1) The answer to any of questions 2-15 on the self-assessment form is yes;
2) The project involves data from any social networking sites (Facebook etc.);
3) The project involves health data (e.g. working with NHS projects);
4) The project involves working with children (a Disclosure Scotland form is also required so this should be flagged up well in advance of starting the project).
5) The answer to question 1 is yes AND supervisor/ethics convenor thinks a full ethics form is necessary for some reason not covered by the above.

Please complete applicants' name PAVLO BAZILINSKYY

"I verify that as supervisor, in addition to reading through all of the student's completed self-assessment form, I have also checked that a full ethics form **should/should not** (delete one) be completed."

Signature

Date 4/2/13

Name JON LEWIS

APPENDIX 11: Output of Cachegrind for the test_cache program.

Appendix 11.1: Shared Heap - Dynamic

```
==13397==
==13397== I   refs:      19,258,620,065
==13397== I1  misses:             1,028
==13397== LLi misses:             1,027
==13397== I1  miss rate:           0.00%
==13397== LLi miss rate:           0.00%
==13397==
==13397== D   refs:      11,779,768,219  (9,028,425,520 rd   + 2,751,342,699 wr)
==13397== D1  misses:             4,891  (        1,636 rd   +         3,255 wr)
==13397== LLd misses:             4,880  (        1,625 rd   +         3,255 wr)
==13397== D1  miss rate:            0.0% (          0.0%     +           0.0%  )
==13397== LLd miss rate:            0.0% (          0.0%     +           0.0%  )
==13397==
==13397== LL refs:                5,919  (        2,664 rd   +         3,255 wr)
==13397== LL misses:              5,907  (        2,652 rd   +         3,255 wr)
==13397== LL miss rate:             0.0% (          0.0%     +           0.0%  )
```

Appendix 11.2: Private Heaps - Dynamic

```
==13110==
==13110== I   refs:      19,257,838,664
==13110== I1  misses:             1,007
==13110== LLi misses:             1,006
==13110== I1  miss rate:           0.00%
==13110== LLi miss rate:           0.00%
==13110==
==13110== D   refs:      11,779,609,775  (9,028,349,805 rd   + 2,751,259,970 wr)
==13110== D1  misses:             3,366  (        1,626 rd   +         1,740 wr)
==13110== LLd misses:             3,365  (        1,625 rd   +         1,740 wr)
==13110== D1  miss rate:            0.0% (          0.0%     +           0.0%  )
==13110== LLd miss rate:            0.0% (          0.0%     +           0.0%  )
==13110==
==13110== LL refs:                4,373  (        2,633 rd   +         1,740 wr)
==13110== LL misses:              4,371  (        2,631 rd   +         1,740 wr)
==13110== LL miss rate:             0.0% (          0.0%     +           0.0%  )
```

Appendix 11.3: Shared Heap - 1 core

```
==13904==
==13904== I   refs:      19,258,659,144
==13904== I1  misses:             1,059
==13904== LLi misses:             1,058
==13904== I1  miss rate:           0.00%
==13904== LLi miss rate:           0.00%
==13904==
==13904== D   refs:      11,779,788,309  (9,028,440,406 rd   + 2,751,347,903 wr)
==13904== D1  misses:             4,930  (        1,673 rd   +         3,257 wr)
==13904== LLd misses:             4,911  (        1,654 rd   +         3,257 wr)
==13904== D1  miss rate:            0.0% (          0.0%     +           0.0%  )
```

```
==13904== LLd miss rate:            0.0% (          0.0%     +           0.0%  )
==13904==
==13904== LL refs:                5,989  (        2,732 rd   +         3,257 wr)
==13904== LL misses:              5,969  (        2,712 rd   +         3,257 wr)
==13904== LL miss rate:             0.0% (          0.0%     +           0.0%  )
```

Appendix 11.4: Private Heaps - 1 core

```
==14394==
==14394== I   refs:      19,257,877,876
==14394== I1  misses:             1,038
==14394== LLi misses:             1,037
==14394== I1  miss rate:           0.00%
==14394== LLi miss rate:           0.00%
==14394==
==14394== D   refs:      11,779,629,921  (9,028,364,725 rd   + 2,751,265,196 wr)
==14394== D1  misses:             3,399  (        1,656 rd   +         1,743 wr)
==14394== LLd misses:             3,398  (        1,655 rd   +         1,743 wr)
==14394== D1  miss rate:            0.0% (          0.0%     +           0.0%  )
==14394== LLd miss rate:            0.0% (          0.0%     +           0.0%  )
==14394==
==14394== LL refs:                4,437  (        2,694 rd   +         1,743 wr)
==14394== LL misses:              4,435  (        2,692 rd   +         1,743 wr)
==14394== LL miss rate:             0.0% (          0.0%     +           0.0%  )
```

Appendix 11.5: Shared Heap - Round-Robin

```
==14666==
==14666== I   refs:      19,258,663,865
==14666== I1  misses:             1,062
==14666== LLi misses:             1,061
==14666== I1  miss rate:           0.00%
==14666== LLi miss rate:           0.00%
==14666==
==14666== D   refs:      11,779,789,660  (9,028,441,543 rd   + 2,751,348,117 wr)
==14666== D1  misses:             4,879  (        1,655 rd   +         3,224 wr)
==14666== LLd misses:             4,878  (        1,654 rd   +         3,224 wr)
==14666== D1  miss rate:            0.0% (          0.0%     +           0.0%  )
==14666== LLd miss rate:            0.0% (          0.0%     +           0.0%  )
==14666==
==14666== LL refs:                5,941  (        2,717 rd   +         3,224 wr)
==14666== LL misses:              5,939  (        2,715 rd   +         3,224 wr)
==14666== LL miss rate:             0.0% (          0.0%     +           0.0%  )
```

Appendix 11.6: Private Heaps - Round-Robin

```
==14937==
==14937== I   refs:      19,257,882,642
==14937== I1  misses:             1,038
==14937== LLi misses:             1,037
==14937== I1  miss rate:           0.00%
==14937== LLi miss rate:           0.00%
==14937==
==14937== D   refs:      11,779,631,231  (9,028,365,644 rd   + 2,751,265,587 wr)
==14937== D1  misses:             3,400  (        1,655 rd   +         1,745 wr)
==14937== LLd misses:             3,399  (        1,654 rd   +         1,745 wr)
==14937== D1  miss rate:            0.0% (          0.0%     +           0.0%  )
==14937== LLd miss rate:            0.0% (          0.0%     +           0.0%  )
```

```
==14937==
==14937== LL refs:                 4,438  (        2,693 rd   +         1,745 wr)
==14937== LL misses:               4,436  (        2,691 rd   +         1,745 wr)
==14937== LL miss rate:              0.0% (          0.0%     +           0.0%  )
```