Comparison of register allocation strategies for the PLEX programming language

Peter Karlsson, peter.karlsson@mds.mdh.se
Stefan Ohlsson, stefan.ohlsson@mds.mdh.se

January 2000

Master's thesis report for 20 credits in computer sciences

Abstract

PLEX is a real-time language used for programming telephone exchanges at Ericsson. The compiler for the language was written back in the 1980s, and so was the register allocation part. Research has found better ways of allocating registers, why it was considered time for the old algorithm to be replaced. This report describes and studies a number of different register allocation algorithms which could fit that bill, describes how the switch was made, and compares how well the new algorithm performs in comparison to the old.

Contents

Chapter 1

Requirements

This report is for a master's thesis on D level, for 20 credits in the computer sciences area.

1.1 Definition of the thesis

This is the definition of the master's thesis that we have carried out:

> PLEX is a real time language used for programming telecommunication systems at Ericsson. The language is based on a signaling paradigm, and is very efficient in handling thousands of ongoing "processes." Even so efficiency is always an issue and continuously improvements to the compiler to produce more efficient machine code is important. A global register allocation technique called "Sequential Graph Coloring with Graph Reduction" is used in the PLEX compiler. One new algorithm called "fat cover" algorithm is according to research results on average better than the graph coloring algorithm.

1.2 Background

The way compilers are written today they work in several passes; Firstly, the program code that the programmer has written is interpreted, and at the same time it is checked so that the correct syntax is used. This interpretation yields an internal representation of the program that is easier to work on in the other parts of the compiler. Several analyses are done on this internal representation, to determine which optimizations should be performed, and what machine language code should be generated for the target system. Just before the code generation is done, a register allocation step is performed, which decides what machine registers (if any) to use for each variable and constant [1].

1.3 Goals

The goal is to replace the algorithm today used for register allocation in the PLEX compiler with a more modern approach. The allocation algorithm used today was written in the early 1980s, and was then considered to be good, but research done since then has shown that it can be implemented in other, more efficient ways [9].

Initially, the plan was for us to examine whether it was possible to implement the algorithm known as "fat cover" [10], but it was found not to be suitable for the programming language as it is today, so we have chosen to research a couple of other algorithms instead.

Chapter 2

What is register allocation?

Register allocation is the step in the analysis that chooses how variables are to be stored during the program execution, if they are to be stored in memory or in CPU registers, and if so, in which registers [2].

2.1 Identification of live ranges

To be able to identify the live ranges in a good way, the compiler needs to figure out when the different variables are used. This is done by splitting the variables into *live ranges*, which consist of the time from when the variable gets a defined value, to the last possibility for this value, or a value derived from it, to be used. By splitting the variables in this manner, you do not have to worry about variables that are used for different reasons in different parts of the programs, for examples as indices in loops [4].

2.1.1 Live ranges in the PLEX compiler

The PLEX compiler's flow analysis section partitions the variables into instances, which consist of the variables' *def*inition and *use* points, and the region between. The extent of an instance is from its first *def* to its last *use* [13], with a granularity at the PLEX statement level. This corresponds to the definition of live ranges, as given above. The reason for the granularity being so low is because of the high level at which the internal representation (SPIL, *"Simple PLEX Intermediate Language"*) works.

2.2 Graph coloring

A graph consisting of all nodes (instances), interconnected by edges (interferences) are to be colored by a limited number of colors (registers) in such a manner that no nodes which are connected directly to each other get the same color.

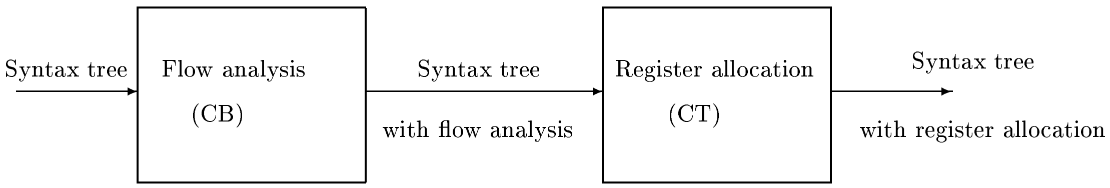

Figure 2.1: Flow in the register allocator

This graph is often not flat, which makes the problem even more complex. A graph is flat if the graph can be redrawn so that no edges cross each other. The well-known four-color problem is defined for flat graphs, and it has been proven that such graphs can always be colored with at most four colors. When the graph is not flat, the worst case requires the same number of colors as there are nodes in the graph (if all nodes are interconnected with all other).

2.2.1 Graph coloring in the PLEX compiler

The global register allocation process in the PLEX compiler manipulates the output of the global flow analyzer. The information about the different live ranges of the instances is used in the register allocation step to determine which instances interfere with each other, and thus must be put in different registers. Two instances are said to interfere if a definition point for one instance is located in the region the other one extends over [12]. See also chapter 7.4 for more information about how the compiler is constructed.

This information is collected in an interference graph, which is used in the algorithm to be able to determine how the registers best can be allocated to instances in a simpler way. If an instance, for some reason, is not possible to allocate to a register, an error message is emitted. More information about how this is done is given in chapter 5.

2.2.2 Global and inter-procedural graph coloring

Usually, two kinds of graph coloring are mentioned, *local* graph coloring, working on the *basic block* level, and *global* graph coloring, which is used on whole procedures. Today, the most common used is the global graph coloring.

Since PLEX lacks functions as defined in conventional programming languages (see chapter 4 for more information), the graph coloring is instead performed on the whole program, something that is called *inter-procedural* graph coloring [3]. This means that instances from parts of the program execution far apart are kept in the same interference graph, something that increases the size tremendously. Not much research has been done on how to simplify inter-procedural graph coloring, since it seems to be generally accepted that global graph coloring is the "best."

Finding another solution to the graph coloring problem than this inter-procedural graph coloring is hard in this case because functions are missing from the language, and it is possible to jump freely between the different parts of the program.

2.3 NP-complete problems

The problem of coloring non-flat graphs is NP-complete, *"non-deterministic polynomial-time complete,"* which means that the number of combination, and thus the time it takes to find the solution, grows exponentially with the number of nodes in the graph. This means that it is very hard to find an optimal solution of the register allocation. Unfortunately, the problem of determining if you are in a certain distance from the optimal solution grows in the same way, and thus finding approximations to the optimal solution is also an NP-complete problem. Because of this, it is always possible to write a program that will render a register allocation that is far from the optimal one, without easily being able to determine that so is the case.

2.4 Register allocation for PLEX

As the PLEX language is defined today, it is the responsibility of the programmer to choose whether a variable is to be stored in a register or in the memory. This means that the register allocation algorithm does not have to move variables out to the memory. Instead, it only has to tell the programmer what variables it thinks should be put in memory if all instances cannot be fit into registers [8].

This simplifies the work of the compiler, but makes it harder for the programmer, among other things because what has to be moved to memory is not a single live range, but instead the whole variable (unless the programmer by himself splits the variable before moving it out).

Something that makes the work harder for the PLEX compiler is hard constraints in PLEX for which the registers certain instances must be placed in, e g when sending signals.

Chapter 3

Notes on how we have worked

The first thing we did was read the documentation on the PLEX compiler and its register allocation, as well as different articles about register allocation in general. We timed most of the reading during vacation time because most people being on vacation we wouldn't get much help with the code anyway. It is also worth mentioning that when we first heard of the name PLEX-C, we thought that it was more like C than it really is. See table A.6 and graph A.7.

3.1 Simulation

We immediately started to use CVS to store code and documentation, because being two on the job, it became much simpler to keep everything up to date and consistent.

After reading documentation on a couple of algorithms, we tested implementations of four different algorithms, Yorktown [6, 5], Optimistic [4], Msk[1] and sequential graph coloring as used in the PLEX compiler, to get an understanding on how much better or worse the new algorithms were.

In these tests, the algorithms simply failed if the register allocation failed. This was done since you always have to manually, in the source code, move those instances which will not fit in registers to the memory when programming PLEX. What we were looking for was the routine that produced the best results without spilling.

Optimistic and Msk were best at coloring the random interference graphs we tried with. They were often much better than both Yorktown and the sequential algorithm. The sequential algorithm, however, was often better than Yorktown. See table A.6 and figure A.7.

3.2 Implementation

After the tests were examined, Yorktown, Optimistic and Msk were implemented in the PLEX compiler. To simplify the work, we split the over 100 Kbyte large file where all the functions were located into smaller parts, one file per allocation algorithm, plus one for shared functions and one for spilling code.

The first compile of the compiler itself it took about half a day, but after that only the register allocator needed to be recompiled, which took significantly less time, but could have been done faster by just doing a re-linking. As it is implemented now, the makefile searches through all directories, something that takes a lot of time. We recompiled the register allocator separately anyway, to check for errors before we ran make on the complete compiler.

First the algorithms were implemented without consideration for spill, since the test programs we first got access to already were adjusted so they were allocatable with the old algorithm. As

[1] An own variant of Matula-Beck [11]

always, a lot of bugs were encountered on the way, which had to be corrected, but the thing that was hardest was to find our way through the old code to see which data structures and values were used.

Our assumptions that the new algorithms that had shown to be better in the previous test should prove themselves better here as well proved itself to be correct. Optimistic and Msk algorithms were the only algorithms to be implemented fully, Yorktown was only implemented partly since we had found that Yorktown was not as good as the others because of its earlier spilling decisions. Because of that, no care was taken to adjust the spilling calculations so that it also would fit it.

When both algorithms were working, implementation of code for taking care of spilling started. Much of the old spill code had to be remade, since the new algorithms work differently from the sequential. Because of this, the same assumptions about free or busy registers (etc.) cannot be made. Also, an error in the existing compiler, in the calculation of which instance is the best prospect for moving out to the memory, was fixed. It was a calculation that by mistake became an integer division instead of a floating point division, an error that probably occurred when the compiler was converted from Pascal. Many residues of Pascal are showing; in some tests that sees if a number is in a certain range a set with all numbers in that range is created and it is checked if the number is in the set. That might seem a bit unnecessary.

3.3 Extensions

After the spill code began to work, we started experimenting with Marie's Law (see chapter 7.2 for more information).

Marie's Law is used to keep certain key registers free, since the code then will get smaller and better. The strange thing with this is that before we used Marie's Law, the total code in the test programs was shorter than when we implemented it. Some programs were longer without Marie's Law (in one case much longer), but others were shorter, making up for the difference. Only when we did not use Marie's Law at all, the total code size was smaller than with the old compiler.

The register allocation order was changed to that which the old compiler used, which drastically decreased the number of used registers. At first, we did not care about the order the registers were allocated with, which resulted in the somewhat unfortunate circumstance of allocating PR0 early, even when no instance required it. But even before this change, the register usage already was lower than the old compiler.

3.4 Testing

Tests were performed on two collections of programs, one consisting of 29 rather small programs (averaging 400 Kbyte source code per program), and the other consisting of 19 large programs (averaging 1.5 Mbyte source code per program). Compiling all the test programs took long time, about two and a half hours for the small programs, and four hours for the large ones. Many of the tests were done with varying constants for Marie's Law. We also created a lot of small programs and scripts to parse and collect information from the debug files written by the compiler. They retrieved the code size, the number of and which registers were used, and the amount of time it took, etc.

Chapter 4

Introduction to PLEX

The compiler we have been working with is written to compile the PLEX programming language, a real-time programming language, closely related to the hardware it is run on, created in 1970 by Ericsson (with some additions in 1983) for use in their telephone exchanges. PLEX is used in thousands of exchanges all over the world.

4.1 PLEX

PLEX is an abbreviation for *"Programming Language for EXchanges."* The programming language is largely specialized for its area of use, and is in part tied to the underlying hardware, the *AXE* telephone exchange [8].

PLEX is available in two flavors, PLEX-C and PLEX-M. The variant we have worked with is PLEX-C, which is used in *CP*, the central processors of the telephone exchanges.

PLEX-M is an eight-bit variant of PLEX that is used in the *EMRP*, *"extension module regional processor,"* which is the part of the telephone exchange that handles the subscriber stage. We have not worked with this variant.

4.2 Terminology

Terminology used in PLEX:

- A *"code segment"* is code that is executed from a call to an exit, and can be compared to a function.
- A *"block"* is a collection of code segments that shares global variables. Several blocks usually co-operate to handle a complete telephone exchange.
- To communicate, the code segments send *signals* to each other.

4.3 Hardware

PLEX is closely tied to the hardware it is run on, Ericsson's own system APZ, which is used in telephone exchanges. APZ has built-in functionality for message queues, with full support for sending and receiving messages, something that has been inherited by the PLEX language. This makes the translation from program code to machine code quite simple for these instructions. However, this is a disadvantage if you wish to move the programming language to another hardware architecture, but that is beyond the scope of this report.

It is also possible to write small pieces of code written in the CPU's assembler language, *ASA blocks* or *ASA sectors*, directly in the PLEX program.

4.4 Signals

Signals are a central concept in PLEX, which parallel function calls in conventional programming languages. When a program wants to call another code segment, either in the same, or another, block, it sends a signal.

There are two kinds of signals, single signals and combined signals. The single signals are stored in a message queue, from which the operating system will figure out which program code to execute when the current segment exits.

Combined signals resemble function calls in conventional programs, since they consist of two signals, one in each direction. However, the semantics differ somewhat from function calls, since the called signal can chose to send one of many different signals in return, and these are handled by the code segment that originally sent the signal by telling where all the possible resulting signals are to be taken care of.

To avoid it taking too long time from when a signal is sent to when the receiving code segment is executed, it is possible to send signals which are not to be queued. It is also possible to send signals that should wait a certain amount of time before being executed.

Different data can be sent together with the signal. In the case of single signals, these are copied to the signal queue, and retrieved when the signal is executed. For combined signals the data is sent like normal function call arguments in other programming languages.

The signals are mainly *non-preemptive* (but see also the next section concerning the levels of execution), meaning that only one piece of code is executed at a time. This simplifies things as to conventional programming languages' function calls, since the local variables do not need to be stored to memory each time. However, this means that, when it comes to combined signals, one cannot be guaranteed that any of the temporary variables retain their values when you return after the call [7].

4.5 Execution model

The execution of PLEX programs is centered around the signals, since everything that ever executes are signals. When a process sends a single signal, this is saved in a signal queue, usually in the last position, but that can be changed with the signal priorities.

Support for the signal queues is available in the hardware, something which makes it possible for effective switches between the executions of the different signals. When a signal has finished, special instructions are used to find out from the queue, which signal is to be run next. This signal is then removed from the queue and executed. If there were delayed signals whose time has come, these are moved from the waiting queue, to the execution queue, but they are placed last, which makes it possible that it takes a little while longer before they are executed.

There is a kind of limited pre-emption; there are four different execution levels, where a signal that running at a higher level automatically pre-empts those running at lower levels. The four levels that exist are (from the highest to the lowest):

1. TRL *"trace level"* is used to find errors.
2. THL *"traffic handling level,"* which is the level that is used for normal telecommunications.
3. CL *"C Level"* is used for I/O handling of different kinds.
4. DL *"D Level"* is used for self-tests of soft- and hardware.

The division into priority levels also shows that the soft- and hardware specialized just for this purpose, it gives priority to telecommunications before other tasks, and it has direct support for doing so in the hardware.

This, however, does not have any impact on the register allocation, since there are complete sets of registers for each level of execution.

4.6 PLEX variables

Variables in PLEX come in two flavors; temporary variables that are only retained from when a signal is started to when it is finished, and permanent variables that always are accessible, and which are stored in the memory (DS – *data store*).

Since a PLEX program is non-pre-emptive, all temporary variables are stored in registers, without having a memory location to be saved to, and, in the same manner, registers are not used for the permanent variables more than temporary when they are used in various calculations.

4.6.1 Effects on the register allocation

In PLEX, it is the responsibility of the programmer to make sure that the registers used are enough for the temporary variables in use, and if so is not the case, one has to move one or more variables to the permanent memory.

With this arrangement, the register allocation routine does not have to be able to introduce code to spill variables that do not fit, something that has to be done for conventional programming languages. However, the compiler should to hint the programmer about which variables are suitable for spilling, or at least which could simplify the register allocation if they were spilled, so such functionality exists, but as soon as a temporary variable has to be spilled, it will result in the compiler stopping in the end of the compilation phase, and reporting which variable(s) that should be moved to the data store.

Chapter 5

Existing algorithm

When a variable is spilled by the PLEX compiler it must be entirely put in the memory, *"data store,"* unlike in other programming languages where code is inserted to move it to a register temporarily, and then write it back to memory again. When the compiler discovers that an instance must be spilled it is still put in a dummy register so the code generator can run and possibly find more instances that have to be spilled (due to lack of registers for temporary calculations). The user that gets messages about which variable(s) has to be put in memory for the compilation to succeed (if any).

5.1 Description

The register allocation is based upon graph coloring, a sequential algorithm called *"Sequential Graph Coloring with Graph Reduction."*

The graph is traversed sequentially, node by node. Every node is given a color that none of the interfering nodes have. For this coloring an unlimited number of symbolic registers are used. When the coloring is done the number of colors surrounding each node is counted. This is done for each node in the graph. Note that the number of colors that surrounds a node can be less than the actual number of nodes that surrounds the node. That is because nodes that do not directly interfere with each other can get the same color.

Then it is checked that the number of colors around each node does not exceed the allowed number of colors (the number of available machine registers). If it is equal to or exceeded that at any node, the node must be removed from the graph. The node that has most different colors next to it is selected for removal. The process continues in the same way until the graph is colored with a number of colors at most equal to the number of available machine registers.

5.2 Register assignment

Symbolic registers are allocated to the instances so that no node has a neighbor with the same color (register) as itself. An infinite number of symbolic registers are available. If this succeeds (the number of symbolic registers is less or equal to the number of machine registers), the instances are given real registers, otherwise the compiler stops with an error message.

In PLEX it is possible to use assembly code in-line in a so called *ASA sector*. It must be guaranteed that the register variables used in the assembly code is mapped to the register they represent.

5.3 Affinity

Some assembly instructions demand that the parameters are in specific registers. If one or more parameters are in other registers they must be moved to the right registers and then back again. If the registers that the instruction required already was in use, their original contents must be saved before the instruction, and be restored again afterwards. This takes time, and is something that should be avoided. Because of that, a strategy is employed that tries to allocate the instances that contain such parameters to the correct registers in the first place. These registers are called the *affinity* registers of the instance. See chapter 7.1 for more information on affinity.

5.4 Drawbacks

A problem with allocating registers sequentially is that it seldom is the optimal order, which also means that the register allocation will fail unnecessarily often. Because graph coloring of non-flat graphs is an NP-complete problem it is impossible to find the optimal solution, but there are methods to find better solutions than just going through the nodes in order. It is those kind of strategies we have tried to examine closer.

5.5 Pseudo code

The sequential graph coloring looks somewhat like this:

```
" This algorithm goes through the instances sequentially, and tries to
  find a color for each instance in the order they are numbered"

BEGIN Color the Instances ( startinst, maxnrsmbregs).

   PreColor Pseudo Variables( All Instances ).

   "Loop through each instance and try to assign a color to it"
   FOR inst IN instances DO
      "Try to find a color for this instance"
      FOR col IN available_colors DO
         "Is this color busy?"
         busy=FALSE.

         FOR i IN interferands DO
            IF i INTERFERES inst THEN
               "inst interferes with i"
               IF Assigned a Color(i) AND
                  col = Color of (i) THEN
                  "Color is busy! Try again."
                  busy=TRUE.
               END IF
            END IF
         END FOR

         IF NOT busy THEN
            "The color col is available"
            Assign Color(inst,col).
         END IF
```

```
      END FOR

      IF busy THEN
         "Failure! No color was available."
      END IF
      IF ERROR THEN
         EXIT
      END IF
   END FOR
END.
```

Chapter 6

New algorithm

We have implemented a couple algorithms, which all by themselves have their pros and cons.

6.1 What algorithms exist?

There are a lot of different algorithms to allocate registers in functions, brief descriptions of a few of them follows:

6.1.1 Graph coloring

Graph coloring sees the problem with assigning registers to variables as a coloring problem. The live ranges of the variables are nodes in a graph, and they are tied together where the corresponding live ranges interfere.

6.1.2 Fat cover

Fat cover tries to do a better work on loop variables, since other algorithms often underestimates how much can be earned by putting them in registers. By working inside-out, allocating registers to loop variables first, faster code is hopefully created. To be able to perform a good allocation, the programming language has to be well-structured, the more the better, but PLEX is not very well structured, often containing `goto`, something that allows arbitrary jumps in and out of loops, and between signal handling routines. This makes the implementation a lot harder, and reduces the gain. Furthermore, implementation of Fat Cover would require substantial modifications of both the flow analyzer and register allocation routines, something that there simply was not enough time to do.

6.1.3 Priority based

Priority based graph coloring was introduced by Chow and Hennessy. Instead of making the graph simpler by spilling, they divide instances in smaller parts with use of *live range splitting*. To keep the costs down, the interference graph must be made more crude and not as exact as with Chaitin's algorithm. Instead of looking at interferences on an instruction level, this algorithm looks at the basic block level, where a basic block is a sequence of instructions that are always executed linearly [1]. Thus, it assumes that all instances that exist in the same basic block are interfering with each other. Priority based algorithms can however give better results than spilling algorithms (Yorktown, Optimistic, etc) in some cases. Another difference, except the way they see interferences, is that Chaitin's algorithm assumes that all instances are in registers and spills when needed, but the

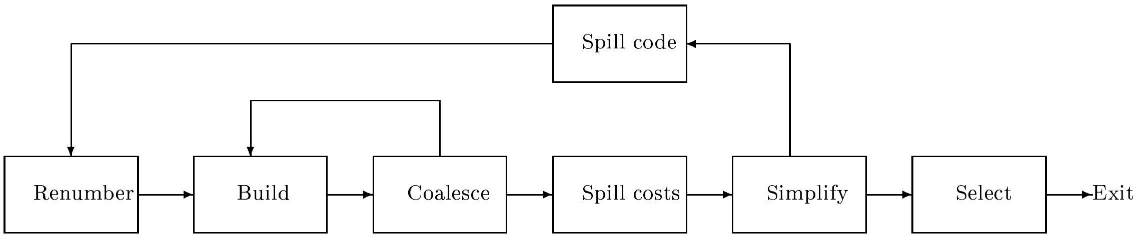

Figure 6.1: Chaitin's register allocator

priority based algorithm assumes that all instances are in memory and tries to put the ones that produce the best results (execution time wise) in registers.

6.2 Why did we choose Optimistic?

Optimistic was sufficiently similar to the sequential algorithm to make the implementation simple, but at the same time so much better that it was worth implementing.

6.3 General description of graph coloring

The algorithms below all build upon Chaitin's register allocator and consist of 7 fundamental parts [4]. See figure 6.1.

Renumber enumerates all live ranges uniquely.

Build then builds an interference graph showing where the different live ranges interfere with each other.

Coalesce removes unnecessary copies by combining two or more live ranges that in reality are the same value, but under different names, by removing the copy instruction and combining them. Since such an operation can change the interference graph, *Build* and *Coalesce* are repeated until no further coalescing is possible.

Spill costs computes the cost in terms of longer execution time that is caused by spilling a live range. This is done for every live range in the graph.

Simplify together with *Select* are the parts that color the graph. Simplify dismantles the graph node by node and pushes them onto a stack according to specific criteria (which vary from algorithm to algorithm) for coloring in a later step. The stack is a normal last-in-first-out stack and is used as intermediate storage of the nodes while it is being dismantled.

Select takes each node from the graph and colors them (if possible), in the opposite order to that which it was pushed on the stack.

Spill code inserts code to load/save registers for the live ranges that were marked for spilling earlier, and if code needs to be inserted everything is redone from *Renumber* because the graph is no longer valid because of it.

Because of the manner that the register allocation is implemented in the PLEX compiler, we have not been able to do it exactly the way it is written. The differences that have come up are detailed in chapter 7.

6.4 Yorktown (Chaitin)

Description

The Yorktown allocator was the first implementation of global register allocation based on graph coloring [6]. For the sake of efficiency, the interference graph is stored in two ways simultaneously, a triangular bit matrix and vectors that show which live ranges interfere with which (adjacency vectors). Simplify always removes nodes that has degree less than k in arbitrary order. If no node with degree less than k exists, an instance is spilled immediately and the whole graph coloring is restarted.

Advantages and disadvantages

Because of the Simplify-algorithm's pessimistic assumption that a graph cannot be colored if only nodes of degree $\geq k$ are left it will spill often where it is not necessary, but at the same time, when the Simplify phase succeeds, it will automatically be possible to color the graph.

Pseudo code

```
BEGIN Allocate Symbolic Registers to the Instances
        ( startinst , maxnrsmbregs )

  LEFTTOCOLOR := { All Instances } - { Spilled Instances } -
                 { Pseudo Registers }.
  REMOVESTACK := ().

  WHILE LEFTTOCOLOR is non-empty DO
    current := Find Instance That Can Be Colored ( maxnrsmbregs , LEFTTOCOLOR ).
    IF no instance was found THEN
      " Ignore it, which leads to spilling "
      current := Find Instance With Highest Degree ( LEFTTOCOLOR ).
    ELSE
      Push Instance ( REMOVESTACK , current ).
    END IF no instance was found.
    LEFTTOCOLOR LESS := current.
  END WHILE LEFTTOCOLOR.

  WHILE REMOVESTACK is non-empty DO
    current := Pop Instance ( REMOVESTACK ).
    Color Instance ( current ).
  END WHILE REMOVESTACK.

END Allocate Symbolic Registers to the Instances
```

Note: The rest of the pseudo code can be found in appendix B.

6.5 Optimistic (Briggs)

Description

The largest difference between Yorktown and Optimistic is the way it dismantles the graph, see figure 6.2. Instead of giving up and spill if only nodes of degree $\geq k$ remains, it hopes for the best and removes an arbitrary node; in our implementation, it takes the first node in the list of

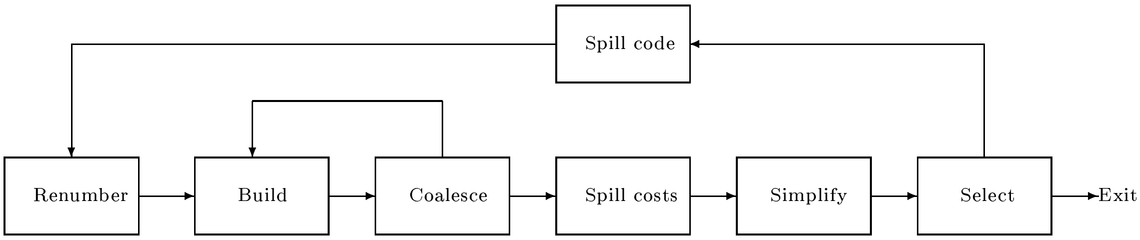

Figure 6.2: Briggs' register allocator

non-colored graph. If it is clear in the later coloring step that the graph is not colorable with k colors this algorithm also has to spill. This, however, occurs much more seldom.

Advantages and disadvantages

Can color all graphs that the Yorktown allocator can color and many more, but it does not show up until in the Select step whether spilling has to be done or not.

Pseudo code

```
BEGIN Allocate Symbolic Registers to the Instances
        ( startinst , maxnrsmbregs )

  LEFTTOCOLOR := { All Instances } - { Spilled Instances } -
                 { Pseudo Registers }.
  REMOVESTACK := {}.

  WHILE LEFTTOCOLOR is non-empty DO
    current := Find Instance That Can Be Colored ( maxnrsmbregs , LEFTTOCOLOR ).
    If no instance was found THEN
      current := First ( LEFTTCOLOR ).
    END IF non-empty.
    Push Instance ( REMOVESTACK , current ).
    LEFTTCOLOR LESS := current.
  END WHILE LEFTTCOLOR.

  WHILE REMOVESTACK is non-empty DO
    current := Pop Instance ( REMOVESTACK ).
    Color Instance ( current ).
  END WHILE REMOVESTACK.

END Allocate Symbolic Registers to the Instances
```

Note: The rest of the pseudo code can be found in appendix B.

6.6 Msk

Description

This method tries to color the graph in two ways, first by removing the nodes with most neighbors first, and then, if this first approach failed, by removing the nodes with least neighbors first. This has shown to be almost as effective as Optimistic.

This algorithm is a variant developed by us of the algorithm described by Matula-Beck [11].

Advantages and disadvantages

Colors more graphs than Yorktown without spill, but takes longer time to do it because it retries once if it fails.

Pseudo code

```
" Variant of Matula-Beck (dubbed ‘msk’) "
BEGIN Allocate Symbolic Registers to the Instances
        ( startinst , maxnrsmbregs )

  LEFTTOCOLOR := { All Instances } - { Spilled Instances } -
                 { Pseudo Registers }.
  REMOVESTACK := ().

  WHILE LEFTTOCOLOR is non-empty DO
    current := Find Instance With Highest Degree ( LEFTTOCOLOR ).
    Push Instance ( REMOVESTACK , current ).
    LEFTTOCOLOR LESS := current.
  END WHILE LEFTTOCOLOR.

  REVERSESTACK := Reverse ( REMOVESTACK ).

  allocatable := true.
  WHILE REMOVESTACK is non-empty DO
    current := Pop Instance ( REMOVESTACK ).
    Color Instance ( current ).
    IF could not color THEN
      " We could not color, try the other direction "
      allocatebale := false.
      QUIT WHILE REMOVESTACK.
    END IF no color.
  END WHILE REMOVESTACK.

  IF allocatable THEN
    REVERSESTACK := ().
    RETURN.
  END IF allocatable.

  REMOVESTACK := ().

  " This direction is the ‘real’ Matula-Beck "
  WHILE REVERSESTACK is non-empty DO
    current := Pop Instance ( REMOVESTACK ).
    Color Instance ( current ).
  END WHILE REVERSESTACK.

END Allocate Symbolic Registers to the Instances
```

Note: The rest of the pseudo code can be found in appendix B.

Chapter 7

Technical details

During our implementation of register allocation in the PLEX compiler, we have struck certain implementation related problems, to which the solutions are not to be directly found in the descriptions of the algorithms we have implemented.

7.1 Affinity degrees

The first problem is that the registers in the architecture that the PLEX compiler works on are used in different ways, something that leads to that certain calculations can only be done in certain registers. This register specialization is a distinguishing property of CISC architectures, such as the one the system is built on.

This is the background to the before mentioned "affinity registers," and each instance of a variable that is involved in a calculation that has to (or should) be placed in a certain register is given an "affinity degree" for that register. This number is then used to determine which register is best suited for containing the variable, to lessen the number of unnecessary copies.

The algorithms we have evaluated do not, in their pure forms, have any support for putting certain instances in specific registers, so it is more or less "glued on" on the algorithm to make it work in an acceptable manner. We have solved the problem by modifying the way the instances are removed from the graph, and by modifying the coloring step to take notice of which registers can be assigned.

We have made as much as we could to ensure that these additions would not affect the mathematical foundations on which the algorithms lie.

7.2 "Marie's Law"

In the original compiler, something called "Marie's Law" [1] was invented to avoid problems with instances that end up in registers they should not end up in. Marie's Law basically says that the compiler should remove certain registers from the register allocation if there are more instances that at any time also have the need to be used in that register.

It is mainly two registers that are affected, one is the pointer register used to find records in data files, PR0, and the other is the index register, IR, which is used for indexing arrays.

Since these operations occur frequently in the program code, Marie's Law in its original form does not allow any instance to use such register directly unless they have many more uses of the register than any other instance. The exception is when the instance is the sole user of the register during its live range, in which case it is given the register right away.

[1] Named after an earlier head of the development department

Register	Description	Use
IR	Index register	Index for arrays
PR0	Pointer register	Pointing to record individuals
PR1	Pointer register	Pointing to buffer variables
CR	Compare register	Store one value in comparisons
SR0, SR1	Signal registers	Store signal information
DR0–DR23	Data registers	Store signal data
AR0–AR3	Arithmetic/logic	Calculations
WR0–WR29	Working registers	Temporary variables

Table 7.1: Register usage

A later addition added the data registers DR0–DR23 to the scope of the law. These registers are important when sending signals, since the data that is sent is stored in the registers, ordered from DR0 and upwards.

The various special registers in the processors are listed in table 7.1 [8].

Since Marie's Law, as implemented, was written for the existing register allocation algorithm, we have tried in our tests to find out which parameters best fit the new algorithms.

7.3 Local register allocation

Another problem we have found is that the register allocation is done at a higher level than the machine code, a limitation that was introduced because of the IBM mainframe system the compiler initially was designed for. The computer was not able to handle more than 16 Mbytes of memory, and an analysis showed that if the register allocation would have been done directly on the machine code, it would take up more memory than was available, why it instead was running on a higher level.

This abstraction leads to the problem that extra registers are sometimes needed, in addition to those that were allocated by the global *(inter-procedural)* register allocation routine, something that is handled in an extra *local* register allocation phase, which is called from the code generation step. This extra register allocation phase means that it is not always a very good idea to generate a "perfect" register allocation in the global phase.

This limitation has not been lifted afterwards, since it would had meant a major rewrite of the compiler. Because of this, we have chosen to "ignore" the problem and targeted making the register allocation step as good as possible, something that sometimes lead to errors occurring afterwards, because the code generation needs to use registers temporarily for calculations, etc.

7.4 Interaction between parts of the compiler

The PLEX compiler is built around a number of different exchangeable modules, something that has meant that it has been quite easy to change just the register allocation part. Two modules are involved with the part we have looked at, one is CB, the global flow analyzer [13], and the other is CT, which is the register allocation itself [12].

The flow analyzer (CB) is the part that partitions the various variables into instances, and which calls the register allocation step (CT). After CB, the code generation module (*"Code Selector"* — CS) is called, which, in turn, calls the before mentioned local register allocator (CL). CT uses some of the library functions common with the CB step to read instance information and handle some set variables, why we had to study the function of this part as well.

We have added some extra library functions that we needed and that were not already included, as well as adding some definitions of extra data sets that are used by the algorithms we have implemented.

We have not changed any other parts of the compiler than CB and CT, and this has not posed any problems, the compiler as a whole works quite well even with a new register allocation algorithm.

Chapter 8

Results of comparisons

To determine which algorithm works best, we have compared the number of registers allocated by the different algorithms, how many instances are allocated to their affinity registers, and the size of the resulting machine code.

8.1 Methods

We have compiled a number of different programs, real programs that are used/have been used in Ericsson's telephone exchanges.

8.1.1 What we have done

These areas are those that we have taken a closer look at:

- The number of unique registers allocated to the different instances, a measurement of how "greedy" the algorithm is in allocating registers. The fewer registers used, the more instances it should be able to allocate before being forced to move variables out to the data store.

  This number does not, however, tell everything about how well the algorithm performs at allocation registers. This is because a single segment with a high register pressure will throw the statistics off course. But all in all, it can probably be seen as a rather decent measurement on how well the algorithm is at saving registers.

- The number of instances that are allocated to their affinity registers, a measurement on how "good" the algorithm is at placing the different instances in the correct registers.

  This number is greatly influenced by Marie's Law, since it prohibits certain instances from being allocated to their affinity registers. It is thus not very strange that our new algorithms, when used without Marie's Law, prove themselves to be much better at allocating instances to their affinity registers, so those results are only included for reference. The results for the new algorithms when using Marie's Law are the ones that are interesting, although we have had some trouble implementing it correctly (see below).

- The size of the generated code, which, when considered in combination with the affinity registers mentioned above, is a measurement on how well the algorithm places the different instances in registers, since code that is generated to move values between registers increase the code size unnecessarily.

8.1.2 What we have not done

We have *not* tried to compare the execution times, or how many instances we have can have in registers before spilling is needed.

The first one is simply because that, on one hand, the programs do not run linearly and on the other hand, on the telephone exchanges they work together, something that makes it hard to compare the execution times. Furthermore, the problem with calculating execution times is beyond the scope of this master's thesis, it is a problem that others have been working on at the same time period as ours.

We did not measure the number of instances that can be put into registers without spilling, partly because the already existing programs we have tested the algorithms on already had been adapted so that they were compilable with the existing algorithm, assigning registers to all instances, and partly because it is hard to generate realistic test cases without having experience of how PLEX programs for telephone exchanges are written.

We did, however, check that the spilling algorithm actually worked by writing a simple program with a ridiculous amount of simultaneous instances, simply to force spilling. The test we did is limited by the number of possible registers, and that limit is the same whether we are running the original algorithm, or any of our modified algorithms, so this test does not say anything about which one is the better, only that it actually works.

8.2 Results

Generally, our new algorithms have proven themselves better than the original algorithm in most areas, even though we have had some problems with adapting Marie's Law (see chapter 7.2) to the new algorithms. We have tested our register allocation algorithms on 48 different PLEX programs, and the results are presented in appendix A.

We have measured two sets of programs — one set of mostly larger programs, and one set with mostly smaller programs (see chapter 3.4), in two different situations — with Marie's Law (tables A.2 and A.4) and without Marie's Law (tables A.3 and A.5) (see chapter 8.1). Please note that the original algorithm is always used *with* Marie's Law, since it is always used in the original compiler.

Diagrams over the results are also found in appendix A.

8.2.1 First comparison: Larger programs

See table A.2 and A.3, as well as figure A.1, A.2 and A.3.

Regarding the larger programs, we can see that the programs increase about 5% in size when compiled with the new algorithms, compared to the old, something that holds true both when Marie's Law is in use, and when it is not. The decrease in the number of used registers is significant (about 20%) for both algorithms, and also quite constant through the tests. Also the increase in the number of registers receiving their affinity registers is significant (about 30% for Msk and about 40% for Optimistic). The improvement is somewhat smaller when Marie's Law is implemented, compared to when it is not, and in the same spirit, the program size decreases somewhat when Marie's Law is not in use.

The improvement calculated as the difference in the number of allocated registers divided by the difference in size (both measured compared to the original algorithm) is almost the same for the two algorithms, Optimistic is 1.11 times better without Marie's Law compared to with it, and Msk 1.13 times.

8.2.2 Second comparison: Smaller programs

See table A.4 and A.5, as well as figure A.4, A.5 and A.6.

Regarding the smaller programs, we can see that the programs are about the same size when compiled with the new algorithms as with the old one, and in one case it is even somewhat smaller. The decrease in the number of used registers is significant (about 10%), while not as large as in the other set of programs, and also quite constant through the tests. Also the increase in the number of registers receiving their affinity registers is about 10–15%, that too somewhat smaller than in the other test set, and the code size increases in the same way as Marie's Law is implemented.

The comparison number calculated as in the previous section shows that Msk is 1.02 times better without Marie's Law than with it for this set of programs, but, in contrast to the other cases, Optimistic generates *smaller* code than the original algorithm, but it is still better without Marie's Law than with.

8.2.3 Compilation times

In table A.1 the total compilation times (excluding the code analysis) for the two program sets are shown for the various algorithms. Please note that the compilation has been done at the same time as other work has been performed on the computers, so the numbers are not entirely accurate. The compilations times have been measured on SUN SparcStation 5 computers with SPARC CPUs at 110 MHz and 128 Mbyte of memory.

Chapter 9

Possible improvements

During our time working on this project, we have thought about a couple details that might be improved, but which has been beyond the scope of this project, or which would have taken too much time to implement. Most of these improvements are described in other literature in the field.

9.1 Affinity

As an alternative and/or complement to affinity, you could let the compiler add new instances with pre-colored registers right before signal calls and other operations that require parameters located in certain registers [3].

Say that a function call requires that its parameters are in order starting with register r_0. Then the compiler could add code to move parameters like this:

$$x \leftarrow f(a, b, c, d);$$

becomes:

$$\begin{array}{l} r_0 \leftarrow a; r_1 \leftarrow b; r_2 \leftarrow c; r_3 \leftarrow d; \\ x \leftarrow f(r_0, r_1, r_2, r_3); \end{array}$$

Adding code for moving parameters to the correct registers before the register allocation code is executed would make it possible to do a more effective allocation and code generation. This is because the code generator does not need to do the moves afterwards, and the register allocation routine does not have to consider the moves, nor keeping registers free for them.

Furthermore, this would not produce extra code, since the coalescing step will combine the pre-colored register instance with the instance that it used first in the function call. If coalescing is not done, the code will still be there, but it would had been added anyway by the code generator at a later stage.

The PLEX compiler register allocator is working on the SPIL level, where SPIL is an internal representation of the PLEX program's syntax tree, something that can make a possible implementation of this in the compiler harder.

9.2 Spill

9.2.1 Calculate on entire variables

When the spill cost is calculated, it is only done on the interfering instances, but when you spill, the whole variable is spilled, including all of its instances. Because of that, it would be more effective to calculate the value on the basis of all instances when the spill cost is calculated. Say that an instance of P and an instance of Q interfere with each other, and one of them must be spilled. As it is now, if Q's instance has a lower spill cost than P's instance, Q is spilled. It would probably pay off to examine whether it is better to calculate the spill cost on both P's and Q's all instances, and spill the one that has the lowest total spill cost.

9.2.2 Best of three

As an effect by it not being possible to tell how far from the optimal solution the found one is located, it follows that it most of the time is impossible to tell if an algorithm is better than another at allocating the registers, which makes the results oscillate, sometimes in favor of one algorithm, sometimes in favor of another.

This also applies to the calculation of the spill costs, why it may be interesting to calculate it using several different methods and take the average, to smoothen any errors for certain variables. This is also suggested in [4] but of course costs time, which is why we did not implement it.

9.3 Live range splitting

The longer the live range for an instance is, the greater the risk is for it interfering with others. To circumvent this problem, the instance can be split into several smaller live ranges, the exact opposite to the coalescing step, where each live range can be allocated to a different register, with move instructions between them. This method is suggested in [3].

However, this splitting increases the complexity of the register allocator, since the number of instances to be allocated increases. There are also some drawbacks in doing this, as you have to take into consideration that the newly split instances are not combined back together again in the coalescing step, something that will result in an increase in the time used by the register allocation.

Chapter 10

Conclusion

We saw rather in the beginning that the sequential graph coloring, as originally implemented, was not very effective, something that proved itself to be correct in the tests we made. However, something that we were not as clear on was that it actually did prove itself to be better than a "clean" implementation of the Yorktown allocator, as written by Chaitin. However, the optimistic allocation routine, as written by Briggs, has exceeded all our expectations, and was able to allocate the variables to less registers, while at the same time give more instances their affinity registers.

Unfortunately, it has been hard to draw any clear conclusions on whether the new allocation routines actually increases the efficiency of the programs in reality, something which mainly comes because:

- The programs we have performed our tests on has already been adapted to compile with the old compiler.
- The execution times for the programs are hard to calculate, since the programs must be run together as a whole to work.

Because of this, we have only conducted closer studies on how many (and which) registers the different variables get allocated to, and the size of the compiled program.

10.1 Register usage

Since the programs we compiled were already adapted in such a way that all variables were allocatable by the old compiler, it has been hard to draw any conclusions on whether the compiler actually facilitates the compilation of more programs without the need to for the programmer to put variables into the data store. In the simple tests we have performed, that seems, however, to be the case.

What we have compared is instead the number of different registers used in the programs. That number shows that the new algorithms work better than the old one did in most cases. However, in some cases the old algorithm is better, but that is only to be expected, since all algorithms try to approximate a "perfect" solution, a solution that is not to be found in polynomical time.

With the algorithms that we have implemented in our tests, the optimistic algorithm has proven itself to be the best one.

10.2 Code size

In one of the test suites the total code size decreased, but in the other one it decreased (the I programs). Our experiments with Marie's Law were not very successful, and in the cases where we used it, the code size was larger than what the original compiler created. In one of the two test suites, the code size decreased if we completely removed Marie's Law. The code size increased as we increased the constants we fed to Marie's Law, while the register usage decreased.

From this, we can draw the conclusion that for Marie's Law, the constants seem to be related to the register allocation routine, and that one has to find new constants for it, perhaps even change the way it works. The constants were empirically determined for the old compiler, and to get the best results, one will have to do the same thing now.

Appendix A

Tables and diagrams

This appendix contains the tables and diagrams discussed in chapter 8. Table A.1 contains the compilation times, and the tables A.2, A.3, A.4 and A.5 contain the following information:

Registers The number of registers used by the program.

Reg-Diff The difference compared to the original algorithm.

Affinity The amount of instances that want affinity registers and actually got one of those that it wants.

Programs	Opt	Msk	Org
Large	3:39	3:32	3:53
Small	1:55	2:31	1:58

Table A.1: Compilation times (hours and minutes)

Data from table A.2 and A.3 is shown in three diagrams; figure A.1 shows the number of registers used in the various programs, A.2 the number of registers receiving their affinity registers and A.3 the size of the generated code.

Similarly, the data from table A.2 and A.3 is shown in the three figures A.4 (number of registers), A.5 (affinity registers) and A.6 (code size).

Results from our simulations, where we took a random interference graph and checked whether it was allocatable without spilling, are shown in table A.6 and figure A.7. All the combinations of the number of instances and registers was run for about an hour each (on a Pentium Classic 120 MHz equipped computer). Since the tests take different amounts of time to finish, the number of tests has varied from 500 (for the combination 1000/30) to 8000 (200/7), but most combinations are tested about 3500 times.

Program	Registers			Reg-Diff		Affinity				Code size		
	Opt	Msk	Org	Opt	Msk	Opt	Msk	Org	Max	Opt	Msk	Org
iucach	32	32	49	-3	-17	417	349	264	576	9696	9436	9281
iucbca	29	30	31	-2	-1	656	585	401	947	21167	21003	21132
iuccs	36	43	52	-6	-9	777	738	510	1367	25387	26844	23357
iuchdb	34	33	46	-12	-13	416	428	275	549	7681	7707	7698
iucsca	29	30	30	-1		654	571	387	963	20636	20573	20711
iucspa	32	33	42	-10	-9	487	474	324	606	8949	8875	8974
iucw	30	31	34	-4	-3	647	586	356	873	21385	21307	21534
iudb	32	32	44	-12	-12	1096	924	689	2173	35680	35481	34532
iuinte	30	30	37	-7	-7	644	631	508	694	8786	8807	9088
iuisph	31	32	36	-5	-4	919	904	679	1044	14938	15333	14962
iumcid	32	32	37	-5	-5	719	679	527	918	20661	20461	20378
iusccs	36	43	52	-16	-9	777	738	510	1367	25387	26844	23357
iussc	32	39	46	-14	-7	625	614	340	812	38747	35318	32149
iusscm	32	32	47	-15	-15	385	369	251	481	9704	9676	9659
iusscr	35	39	45	-10	-6	1485	1368	1142	1699	25403	24577	21630
msdh2	38	38	46	-8	-8	155	155	119	183	14775	14865	14018
mtv	30	30	37	-7	-7	860	862	674	906	9104	9103	9287
reu	30	31	34	-4	-3	2184	2161	1833	2252	18508	18245	18572
sufda	27	27	31	-4	-4	386	381	340	494	16295	16294	16175
TOTAL	607	637	776	-169	-139	14289	13457	10129	18904	352889	350753	336494
%	78.2	82.1	100			75.6	71.2	53.6	100	104.9	104.2	100
						141.1	132.9	100				

Table A.2: Register allocation without Marie's Law, large programs

Program	Registers			Reg-Diff		Affinity				Code size		
	Opt	Msk	Org	Opt	Msk	Opt	Msk	Org	Max	Opt	Msk	Org
iucach	32	32	49	-3	-17	404	336	264	576	9689	9482	9281
iucbca	29	31	31	-2		665	542	401	947	21239	21319	21132
iuccs	35	43	52	-17	-9	753	715	510	1367	26250	26907	23357
iuchdb	33	33	46	-13	-7	403	398	275	549	7711	7782	7698
iucsca	29	31	30	-1	+1	655	526	387	963	20728	20919	20711
iucspa	32	33	42	-10	-9	474	427	324	606	8967	9054	8974
iucw	30	31	34	-4	-3	648	571	356	873	21451	21369	21534
iudb	32	32	44	-12	-12	1238	881	689	2173	35449	35715	34532
iuinte	30	30	37	-7	-7	640	625	508	694	8819	8841	9088
iuisph	31	32	36	-5	-4	905	895	679	1044	14996	15343	14962
iumcid	32	32	37	-5	-5	703	670	527	918	20714	20465	20378
iusccs	35	43	52	-17	-9	753	715	510	1367	26250	26907	23357
iussc	32	39	46	-14	-7	587	584	340	812	38918	35352	32149
iusscm	32	32	47	-15	-5	350	336	251	481	9779	9749	9659
iusscr	35	39	45	-10	-6	1466	1359	1142	1699	25245	24581	21630
msdh2	38	38	46	-8	-8	153	153	119	183	14776	14866	14018
mtv	30	30	37	-7	-7	860	859	674	906	9179	9115	9287
reu	30	31	34	-4	-3	2179	2155	1833	2252	18556	18259	18572
sufda	27	27	31	-4	-4	393	381	340	494	16267	16294	16175
TOTAL	604	639	776	-172	-137	14229	13128	10129	18904	354983	352319	336494
%	77.8	82.3	100			75.3	69.4	53.6	100	105.5	104.7	100
						140.5	129.6	100				

Table A.3: Register allocation with Marie's Law, large programs

Program	Registers			Reg-Diff		Affinity				Code size		
	Opt	Msk	Org	Opt	Msk	Opt	Msk	Org	Max	Opt	Msk	Org
bada	12	13	12		+1	153	108	70	182	7353	7436	7439
baeou	12	12	13	-1	-1	100	100	70	106	1042	1042	1087
bch	16	16	16			159	158	88	229	4924	4930	4996
bt7	23	23	22	+1	+1	375	375	276	388	6194	6194	6443
cawe	22	22	22			319	312	189	336	4971	4970	5299
cawemt	22	22	22			293	276	174	302	4233	4249	4504
cju	24	24	28	-4	-4	272	255	159	292	7918	7935	8165
conman1	38	39	41	-3	-2	73	56	16	179	9373	11490	8734
csu	22	22	23	-1	-1	130	130	119	151	6462	6464	6467
dau	20	20	22	-2	-2	87	86	34	120	4340	4309	4021
dblogh	31	31	45	-14	-14	153	152	101	210	6169	6182	6018
dipst6	21	21	24	-3	-3	150	150	88	175	5351	5351	5503
iucda	30	31	33	-3	-2	277	242	163	335	4845	4784	4831
iuiniw	29	29	32	-3	-3	253	256	206	302	4388	4390	4486
iutcph	30	30	36	-6	-6	205	199	145	306	2796	2800	2898
loas	13	13	13			140	139	101	156	3803	3804	3848
maptc	46	46	42	+4	+4	85	86	56	95	2178	2171	2233
mash	22	23	22		+1	40	50	27	61	2087	2058	2078
milo	18	19	19	-1		25	25	17	32	901	901	890
pestu	12	12	12			91	91	77	105	6271	6271	6271
prmcu	12	14	13	-1	+1	76	48	23	82	2768	2798	2836
rceau	25	25	26	-1	-1	211	185	103	292	7201	7044	6832
re	29	29	37	-8	-8	284	251	165	337	8334	8377	8580
rssa	13	13	13			88	86	52	173	6967	6957	6973
scapr26	23	23	27	-4	-4	258	247	144	347	9447	9423	9422
ssint	23	23	22	+1	+1	125	118	70	151	3483	3488	3466
suscfsb	29	29	39	-10	-10	308	275	163	335	10582	10626	10744
teaman3	31	31	32	-1	-1	88	87	26	107	2465	2459	2512
xpreaau	13	13	13			428	428	156	521	13547	13547	13838
TOTAL	661	668	721	-60	-53	5246	5031	3078	6407	160388	162446	161414
%	91.7	92.6	100			81.9	78.5	48.0	100	99.4	100.6	100
						170.4	163.5	100				

Table A.4: Register allocation without Marie's Law, small programs

Program	Registers			Reg-Diff		Affinity				Code size		
	Opt	Msk	Org	Opt	Msk	Opt	Msk	Org	Max	Opt	Msk	Org
Program	Registers			Reg-Diff		Affinity				Code size		
	Opt	Msk	Org	Opt	Msk	Opt	Msk	Org	Max	Opt	Msk	Org
bada	12	13	12		+1	153	108	70	182	7352	7436	7439
baeou	12	12	13	-1	-1	100	100	70	106	1053	1053	1087
bch	16	16	16			155	157	88	229	4934	4930	4996
bt7	23	23	22	+1	+1	378	378	276	388	6203	6201	6443
cawe	21	21	22	-1	-1	318	309	189	336	4974	4978	5299
cawemt	21	21	22	-1	-1	292	275	174	302	4241	4255	4504
cju	24	24	28	-4	-4	267	250	159	292	7950	7963	8165
conman1	38	39	41	-3	-2	67	52	16	179	9451	11547	8734
csu	22	22	23	-1	-1	130	130	119	151	6464	6464	6467
dau	20	20	22	-2	-2	86	83	34	120	4252	4250	4021
dblogh	31	31	45	-14	-14	153	152	101	210	6172	6182	6018
dipst6	21	21	24	-3	-3	152	152	88	175	5380	5351	5503
iucda	30	31	33	-3	-2	272	235	163	335	4872	4813	4831
iuiniw	29	29	32	-3	-3	256	252	206	302	4345	4416	4486
iutcph	30	30	36	-6	-6	200	190	145	306	2957	2845	2898
loas	13	13	13			140	139	101	156	3813	3814	3848
maptc	46	46	42	+4	+4	86	86	56	95	2177	2171	2233
mash	23	23	22	+1	+1	49	49	27	61	2056	2059	2078
milo	18	19	19	-1		25	25	17	32	901	901	890
pestu	12	12	12			91	91	77	105	6271	6271	6271
prmcu	12	14	13	-1	+1	76	48	23	82	2768	2798	2836
rceau	25	25	26	-1	-1	226	183	103	292	7094	7063	6832
re	29	29	37	-8	-8	284	248	165	337	8371	8420	8580
rssa	13	14	13		+1	92	84	52	173	6997	6968	6973
scapr26	23	23	27	-4	-4	262	232	144	347	9589	9601	9422
ssint	23	23	22	+1	+1	126	118	70	151	3493	3530	3466
suscfsb	29	29	39	-10	-10	309	276	163	335	10610	10638	10744
teaman3	31	31	32	-1	-1	88	87	26	107	2472	2461	2512
xpreaau	13	13	13			428	428	156	521	13547	13547	13838
TOTAL	660	667	721	-61	-54	5264	4917	3078	6407	160528	162926	161414
%	91.5	92.5	100			82.2	76.7	48.0	100	99.5	100.9	100
						171.0	159.7	100				

Table A.5: Register allocation with Marie's Law, small programs

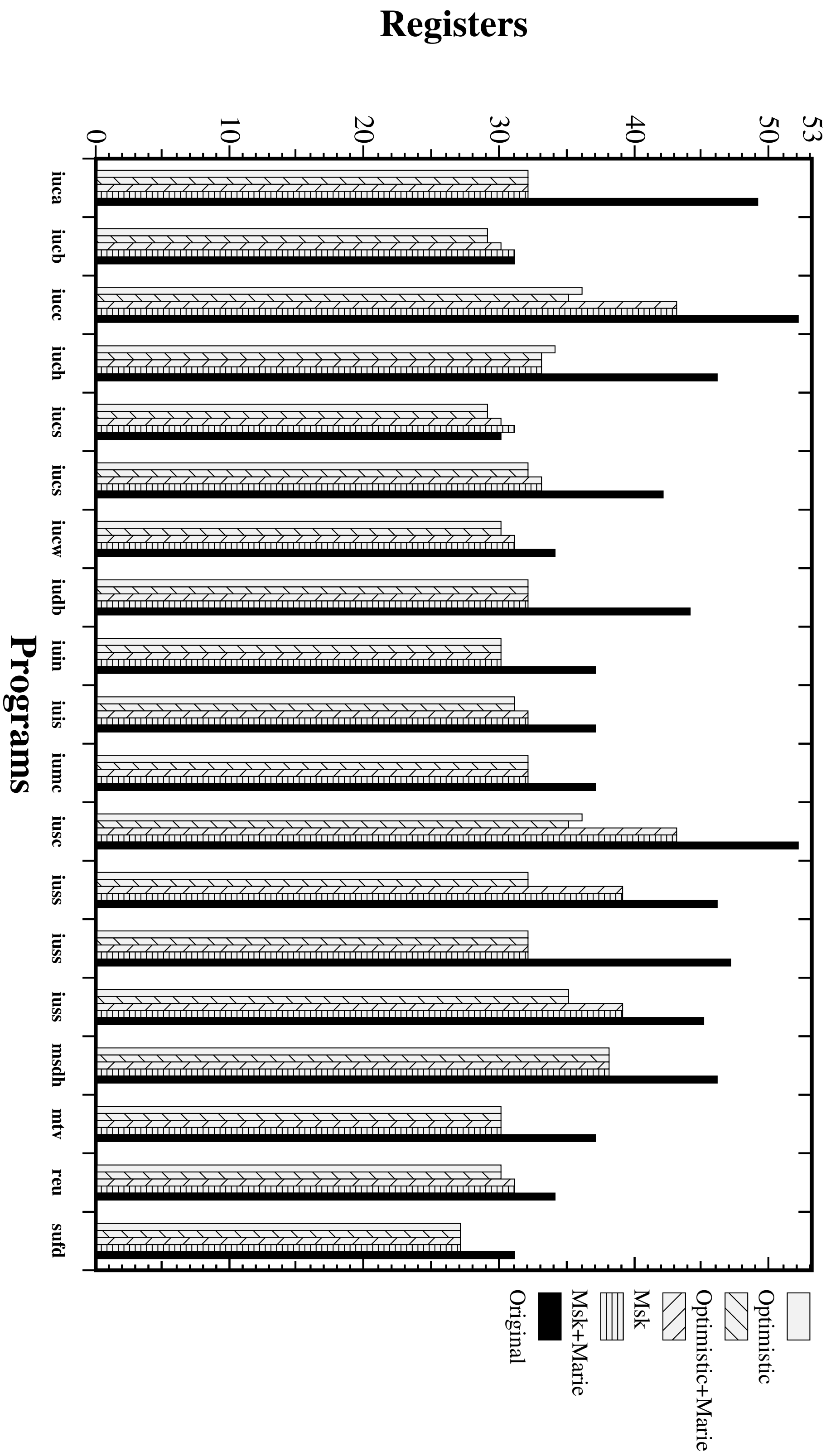

Figure A.1: Register allocation for first set of programs

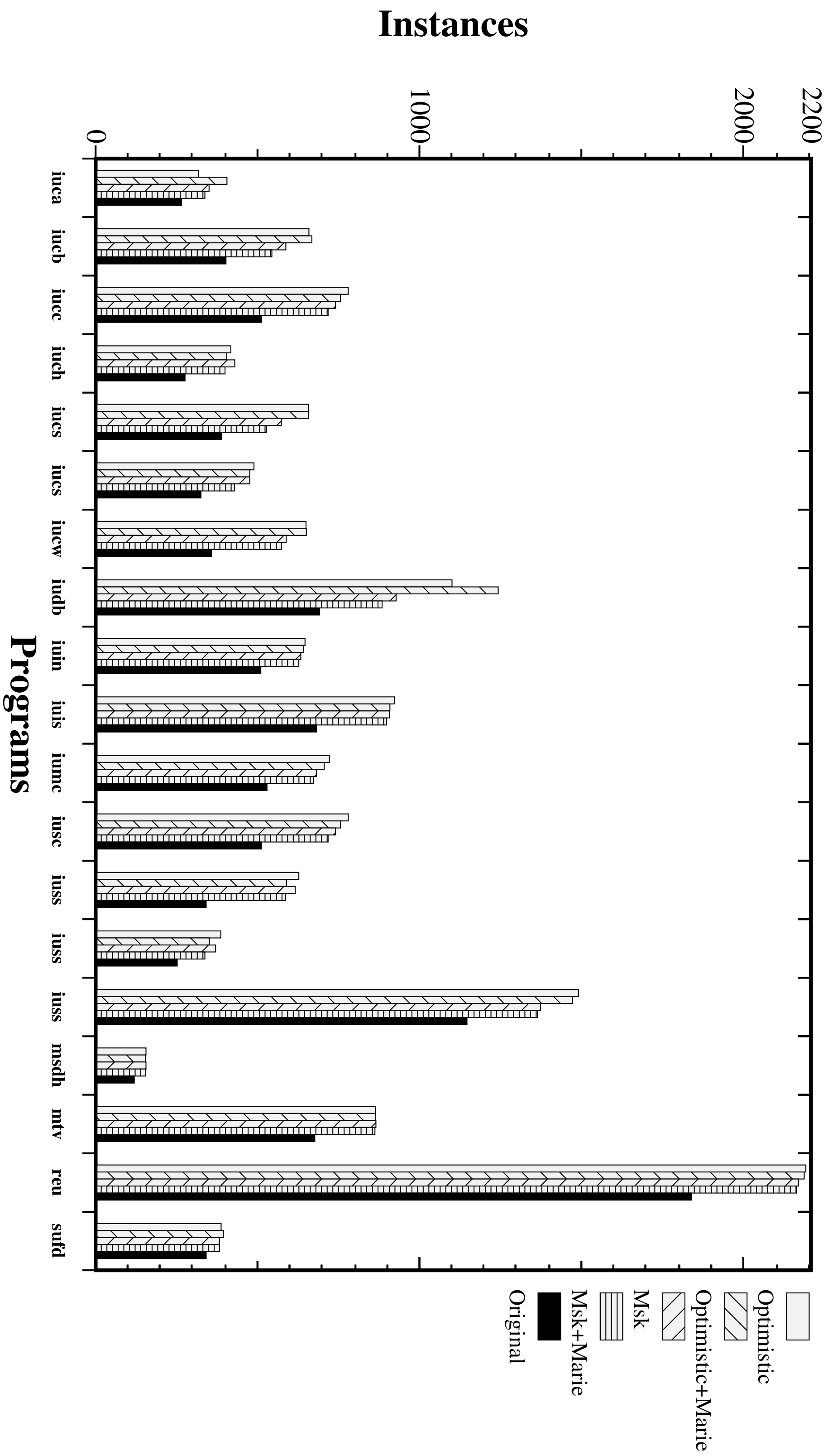

Figure A.2: Number of affinity registers assigned for first set of programs

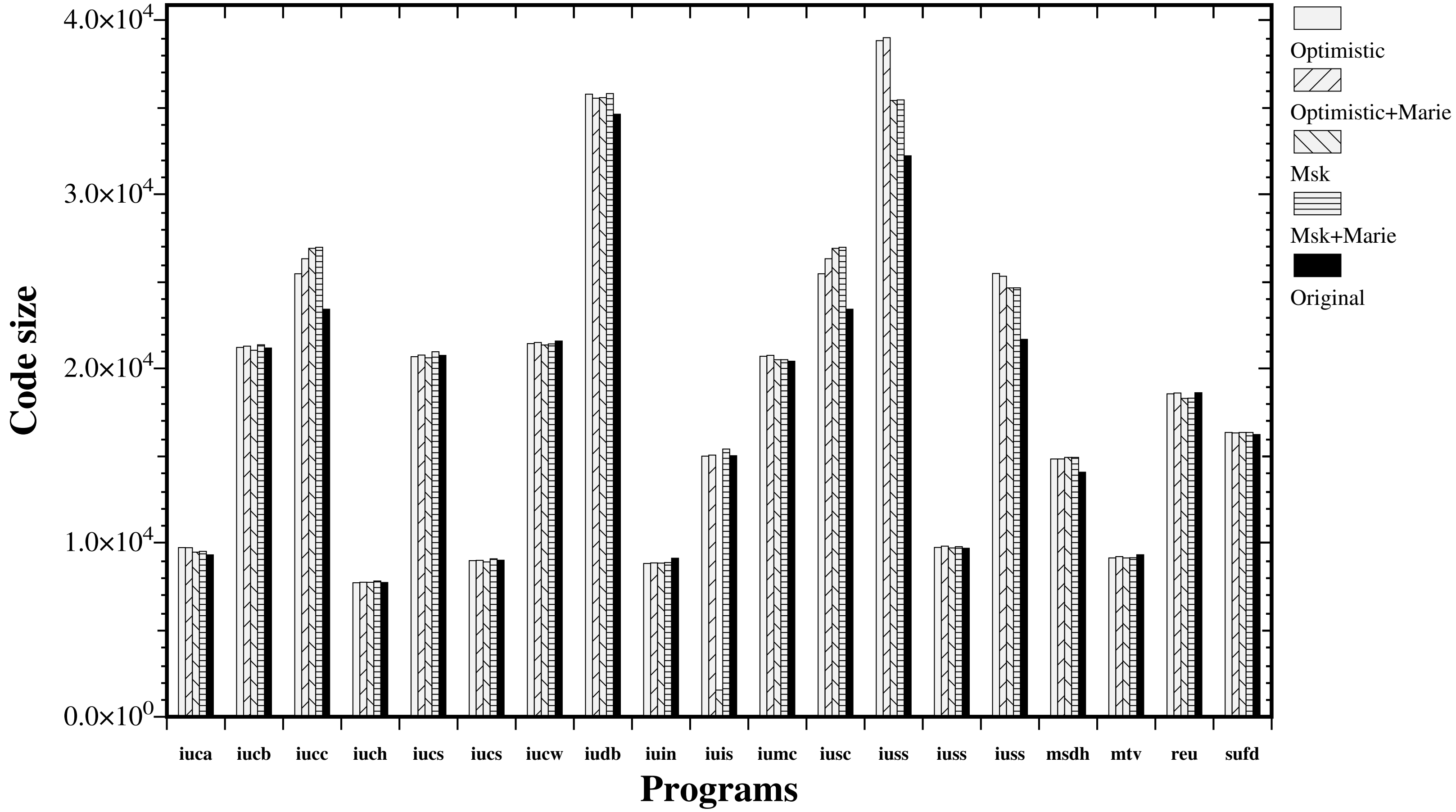

Figure A.3: Code size for the first set of programs

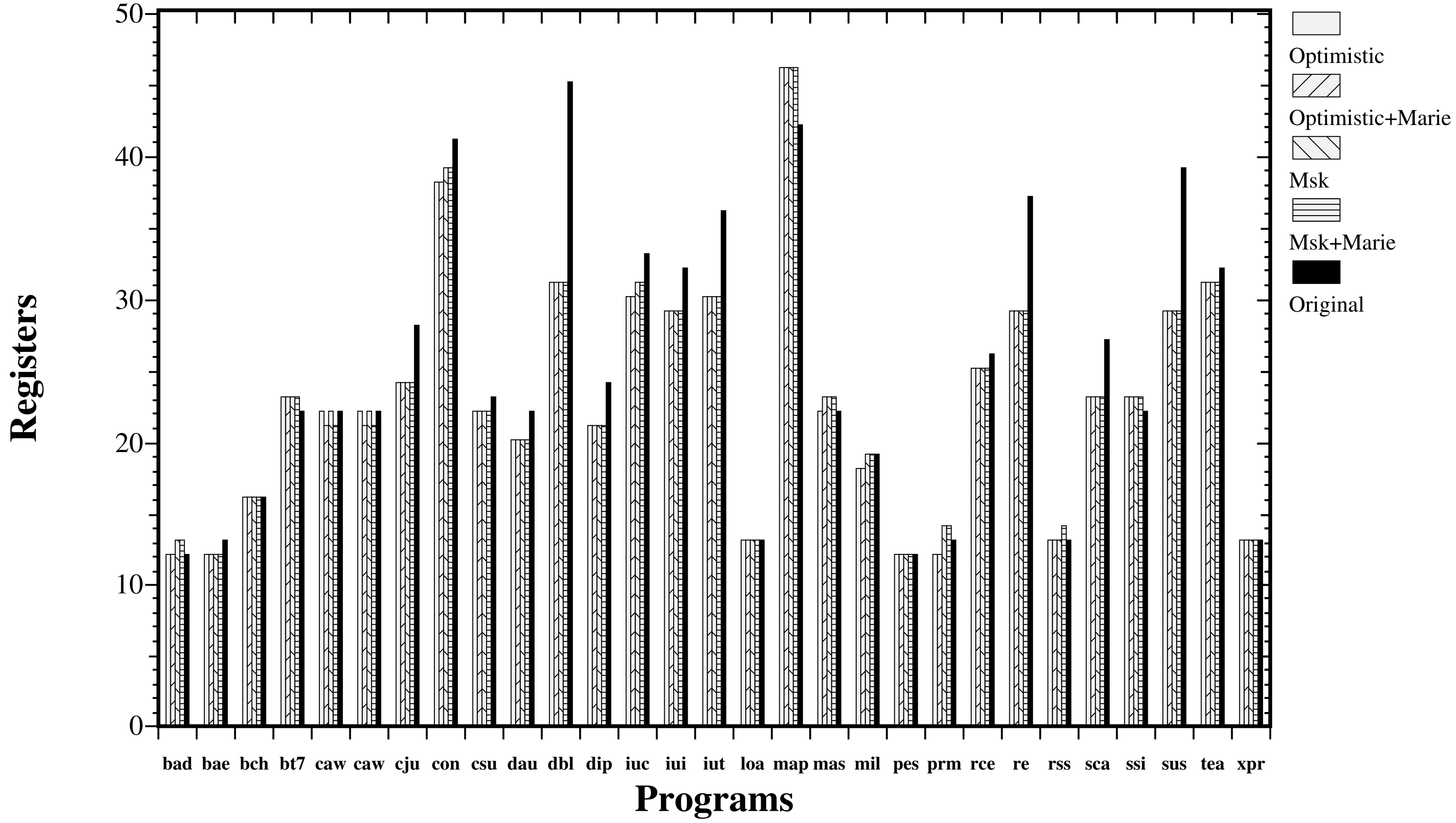

Figure A.4: Register allocation for second set of programs

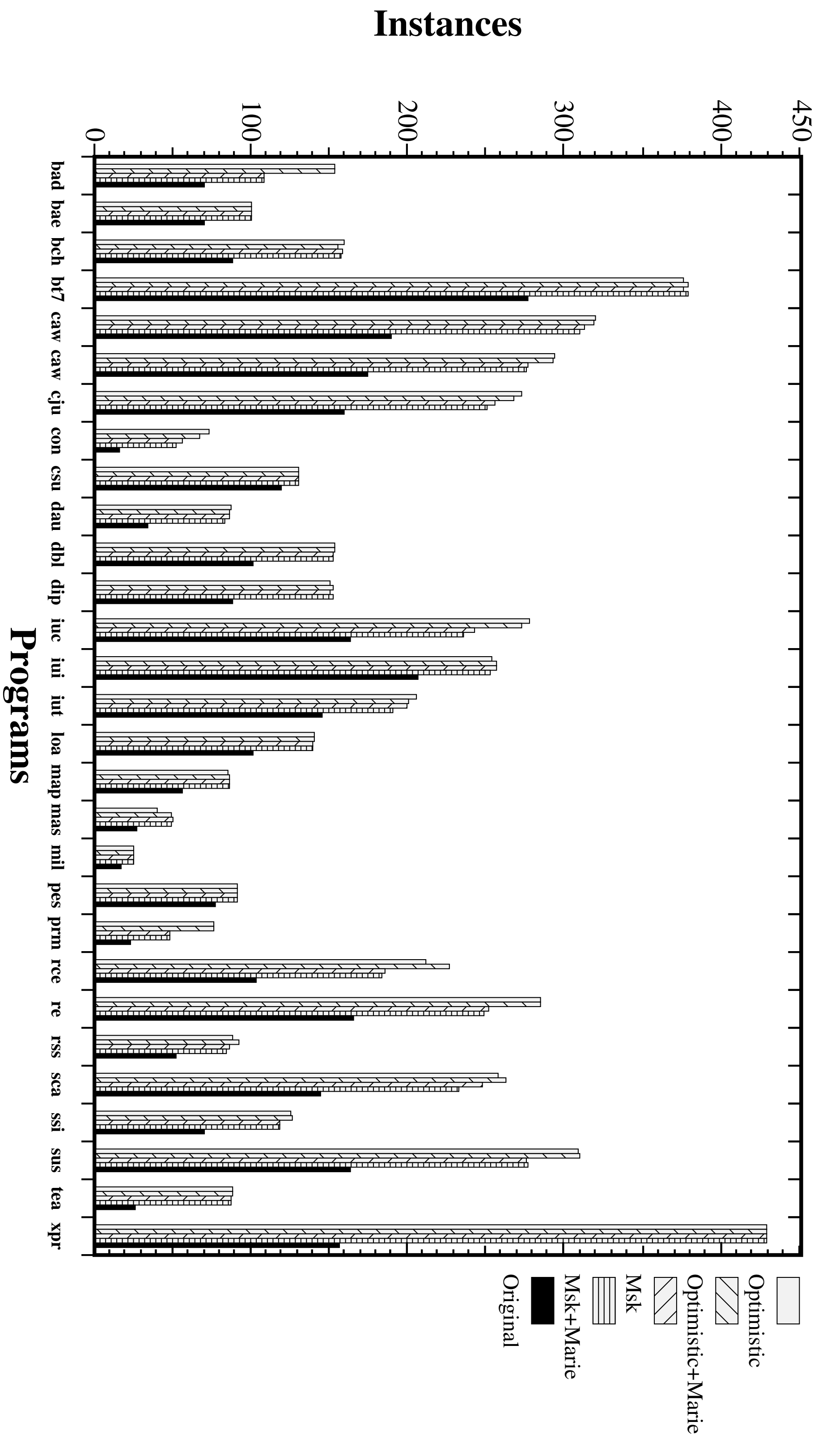

Figure A.5: Number of affinity registers assigned for second set of programs

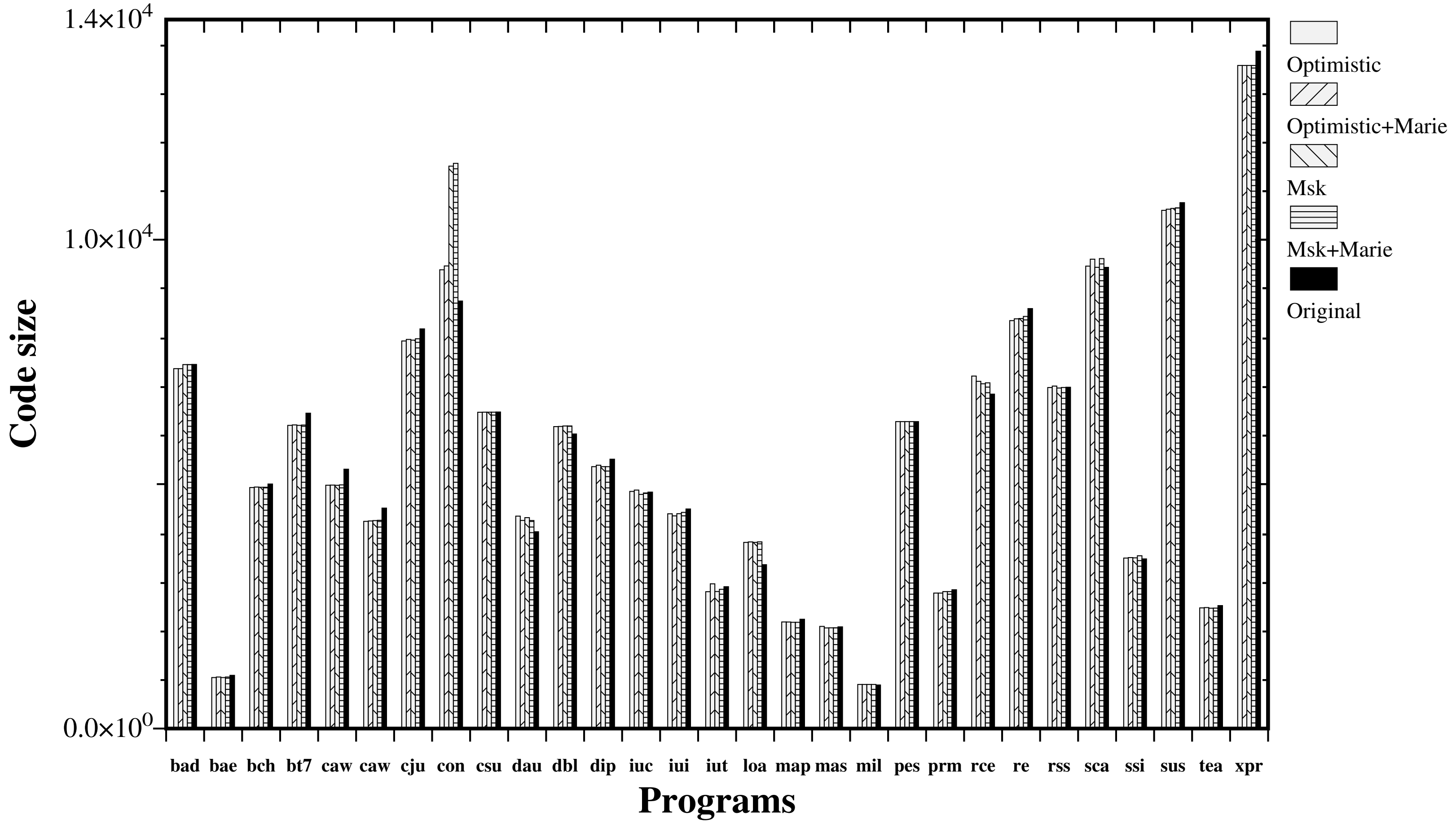

Figure A.6: Code size for the second set of programs

Instances/registers	Yrk	Msk	Opt	Seq
100/20	54.8%	100%	100%	95.4%
200/7	7.7%	10.2%	9.2%	7.0%
250/10	8.9%	14.6%	12.8%	10.4%
250/12	12.0%	20.5%	18.0%	15.0%
250/17	16.5%	32.8%	30.3%	24.8%
250/25	24.2%	60.5%	57.4%	46.2%
400/7	3.7%	4.7%	4.2%	3.0%
500/30	13.4%	36.1%	35.2%	27.7%
1000/30	7.8%	21.2%	20.0%	15.8%

Table A.6: Simulations with random interference graphs

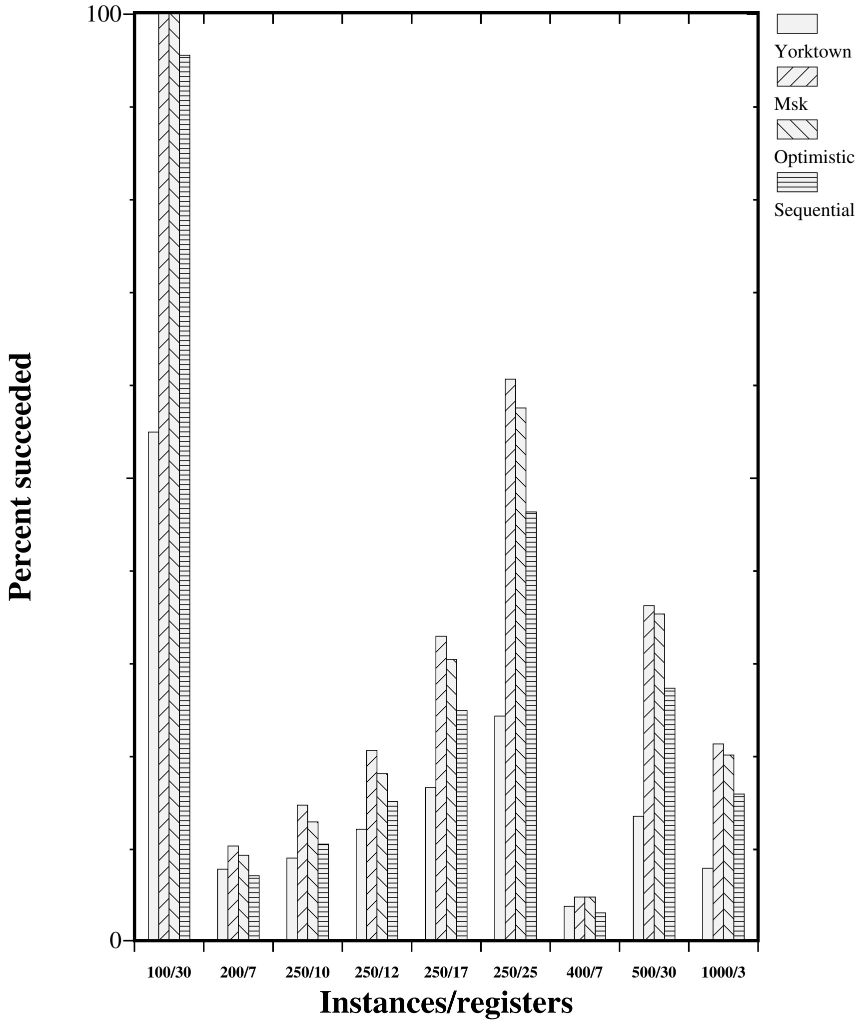

Figure A.7: Simulations with random interference graphs

Appendix B

Other pseudo code

This appendix contains the pseudo code that is shared between the different register allocation routines we have examined, both that which is called by the allocation routines, and that which itself calls them.

B.1 Shared code

```
" This routine finds instance that can be colored, and gives priority
  to those that have affinity defined[1] "
BEGIN Find Instance That Can Be Colored ( available_colors , nodedomain ).

  " Note: ‘Nodes With[out] Affinity’ are not implemented as functions "
  FOR node IN Nodes Without Affinity ( nodedomain ) DO
    IF Number of Neighbors ( node , nodedomain ) < available_colors THEN
      RETURN node.
    END IF Number of Neighbors.
  END FOR node.

  FOR node IN Nodes With Affinity ( nodedomain ) DO
    IF Number of Neighbors ( node , nodedomain ) < available_colors THEN
      RETURN node.
    END IF Number of Neighbors.
  END FOR node.

  " No valid node was found "
  Exit With ERROR.

END Find Instance That Can Be Colored.

" [1] (so that they will be read back early on, and thus hopefully can be
  allocated to a register they have affinity towards "

" This routine finds the instance with the highest number of neighbors.
  Used by ‘mysko’. "
BEGIN Find Instance With Highest Degree ( nodedomain )
  best = -1.
  register = undef_reg.
```

```
  FOR node IN nodedomain DO
    IF Number of Neighbors ( node , nodedomain ) > best THEN
      best = Number of Neighbors ( node , nodedomain).
      register = node.
    END IF node.
  END FOR node.

  RETURN node.
END Find Instance With Highest Degree

" Color the instance ‘node’ with an available register "
BEGIN Color Instance ( node , smbregs ).

  freesmbregs := { symbolic representations of physical registers }.

  " Traverse all interferands "
  FOR interferand IN inst_interferands DO
    IF Allocated ( interferand ) THEN
      freemsbregs LESS := interferand_smbreg.
    END IF interferand.

    IF freesmbregs = {} THEN
      " All colors taken "
      Exit With ERROR.
    END IF freemsbregs.
  END FOR interferand.

  " Tries to allocate an affinity register first, and if none was found,
    then allocates the first available register. "
  node_smbreg := Determine Affinity Register ( node, freesmbregs ).
  IF no affinity register was found THEN
    node_smbreg :=
      Find Register Without Neighbor Affinity ( node , freesmbregs ).
    IF no affinity register was found THEN
      node_smbreg := FirstInAllocOrder ( freesmbregs ).
    END IF no affinity.
  END IF no affinity.

  RETURN node_smbreg.

END Color Instance.


" Find the number of neighbors to ‘node’ in the interference graph that is
  contained in the set ‘nodedomain’"
BEGIN Number of Neighbors ( node , nodedomain )

  count := 0.

  FOR interferand IN node_interferands DO
    IF interferand IN nodedomain THEN
      count + := 1.
```

```
    END IF interferand.
  END FOR interferand.

  RETURN count.

END Number of Neighbors


" Find the best matching affinity register for the instance ‘node’ in
  the set of free registers ‘regs’ "
" Maybe the old code is better? "
BEGIN Determine Affinity Register ( node, regs )

  " Create a set of all free affinity registers "
  freeaffregs = {}.
  FOR register IN regs DO
    IF inst_affregs(register) > 0 THEN
      freeaffregs WITH := register.
    END IF register.
  END FOR register.

  " Remove registers unallocated interferands have a higher affinity to "
  FOR register IN freeaffregs DO
    total_used := 0.
    removed := FALSE.

    FOR interferand IN inst_interferands DO
      IF NOT Is Allocated ( interferand ) THEN
        IF interferand_affregs[register] > inst_affregs[register] THEN
          freeaffregs LESS := register.
          removed := TRUE.
          QUIT FOR interferand.
        ELSE
          total_used + := interferand_affregs[register].
        END IF interferand_affreg.
      END IF Is Allocated.
    END FOR interferand.

    " Our adaptation of Marie’s Law "
    IF NOT removed THEN
      " PR0 is only given to pointers with _strong_ affinity, or
        undisturbed temporaries "
      IF register = PR0 THEN
        IF NOT ((var = ptr AND inst_affregs[PR0] >= MARIE * total_used)
                OR
                (var = temp AND total_used = 0)) THEN
          freeaffregs LESS := PR0.
        END IF NOT.
      END IF register.

      " IR is only given to temporaries with _strong_ affinity, or
        undisturbed pointers "
```

```
      IF register = IR THEN
        IF NOT ((var = temp AND inst_affregs[IR] >= MARIE_IR * total_used)
                OR
                (var = ptr AND total_used = 0)) THEN
          freeaffregs LESS := IR.
        END IF NOT.
      END IF register.

      IF register IN [DR0...DR23] THEN
        IF inst_affregs[register] >= MARIE_DR*totalUsed THEN
          freeaffregs LESS := register.
        END IF inst_affregs
      END IF register

    END IF NOT removed.
  END FOR register.

  " Select the best register from those which are left "
  IF freeaffregs is empty THEN
    RETURN undef_reg.
  END IF freeaffregs.

  affinity_reg := undef_reg.
  max_used     := 0.
  FOR register IN freeaffregs DO
    IF inst_affregs(. register .) > max_used THEN
      affinity_reg := register.
      max_used := inst_affregs ( register ).
    END IF inst_affregs.
  END FOR register.

  RETURN affinity_reg.

END Determine Affinity Registers

" See if we can find a free register that no yet unallocated neighbor
  has affinity to "
BEGIN Find Register Without Neighbor Affinity ( node , regs )

  FOR register IN regs DO
    affinity := FALSE.

    FOR interferand IN node_interferands DO
      IF NOT Is Allocated ( interferand ) AND
         interferand_affreg[register] > 0 THEN
        affinity := TRUE.
        QUIT FOR interferand.
      END IF interferand_affreg.
    END FOR interferand.

    IF NOT affinity THEN
      RETURN register.
```

```
    END IF NOT affinity.
  END FOR register.

  RETURN undef_reg.

END Find Register Without Neighbor Affinity.
```

B.2 Main loop

```
" This is the routine that calls the register allocation routine itself "
BEGIN Determine the Graph's Allocatability
        ( graphstatus , startinst-input , blockedinst , maxnrsmbregs ).

  IF startinst NOT INITIALIZED THEN
    " static startinst "
    startinst = startinst-input.
  END IF startinst.

  FOR inst IN Global Instance Information DO

    " skip spilled instance "
    IF inst_status = spilled THEN
      CONTINUE FOR inst.
    END IF inst_status.

    inst_interfersmbregs := {}.
    inst_smbreg := undef_reg.   " Remove all allocations "
  END FOR inst.

  Allocate Symbolic Registers to the Instances ( startinst , maxnrsmbregs ).
  Search for Blocked Instance ( graphstatus , blockedinst ).

END Determine the Graph's Allocatability

" This rouine searches the graph for the first instance that could not
  be allocated, and spills it "
BEGIN Search for Blocked Instance ( startinst , graphstatus , blockedinst ).

  FOR inst IN Global Instance Information FROM startinst DO

    " skip spilled instances "
    IF inst_status = spilled THEN
      CONTINE FOR inst.
    END IF inst_status.

    IF NOT Allocated ( inst ) THEN
      blockedinst := inst.
      graphstatus := unallocatable.
      QUIT FOR inst.
    END IF NOT Allocated.
  END FOR inst.
```

```
END Search for Blocked Instance.
```

B.3 Spill code

```
" Searches for an instance that was not coloured, and that can be spilled "
BEGIN Search for an Instance to Spill ( blockedinst , spillinst , spillvar )

  largest := 0.

  " Collect a set of the unallocated instances and their interferands "
  SPILLCANDIDATES := {}.
  FOR inst IN Global Instance Information FROM blockedinst DO

    " skip spilled instances "
    IF inst_status = spilled THEN
      CONTINUE FOR inst.
    END IF inst_status.

    " skip allocated instances "
    IF Allocated ( inst ) THEN
      CONTINUE FOR inst.
    END IF Allocated.

    SPILLCANDIDATES WITH := inst.

    FOR interferand IN inst_interferands DO
      " skip spilled instances "
      IF inst_status /= spilled AND interferand > nroccregs THEN
          SPILLCANDIDATES WITH := interferand.
      END IF inst_status.
    END FOR inteferand.

  END FOR inst.

  " Traverse the set and see which one is best to spill "
  FOR inst IN SPILLCANDIDATES DO
    spillvar := inst_var.

    " skip unspillable instances "
    IF spillvar_type = pointer THEN
      CONTINUE FOR inst.
    END IF.

    IF spillvar_syref ( 1 ) = '%' THEN
      CONTINUE FOR inst.
    END IF.

    " Find out how many interfering symbolic registers this instance has "
    FOR interferand IN inst_interferands DO
      inst_interfersmbregs WITH := interferand_smbreg.
```

```
    END FOR interferand.

    spillcriterion := inst_interfersmbregs / spillvar_spillcost.

    IF largest <= spillcriterion THEN
      largest   := spillcriterion.
      spillinst := inst.
    END IF largest.

    inst_interfersmbregs := {}.

  END FOR inst.

  spillvar := spillinst_var.

END Search for an Instance to Spill
```

Acknowledgements

The authors wish to thank Anton Massoud of Ericsson Utvecklings AB for his invaluable support during the time we worked on our master's thesis there.

We also wish to thank our supervisor at Mälardalen University, Kjell Post, who always is available for questions.

Also a small thank you to our friends Andreas A, Andreas G and Camilla for their input and nice company during the time they did their, coinciding with ours, master's thesis at Ericsson.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*, chapter 9. Addison-Wesley Publishing Company, 1986.

[2] Andrew W. Appel. *Modern Compiler Implementation In Java: Basic Techniques*, chapter 11. Cambridge University Press, 1997.

[3] Urban Boquist. *Code Optimisation Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, Göteborg University, 1999.

[4] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, 1992.

[5] G[regory] J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, volume 17, pages 98–105, 1982.

[6] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[7] Ericsson Telecom AB. *PLEX-C Language Description*, 1994.

[8] Ericsson Telecom AB. *PLEX-C 1*, 1997.

[9] Patric Hedlin. A prestudy survey on register allocation. Ericsson internal memo, June 1999.

[10] L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. Technical Memo 33, McGill Universtity, February 1993.

[11] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM*, 30(3):417–427, July 1983.

[12] SoftLab. *Code Generator Temporary Variable Allocator Function*, June 1993. Ericsson No. 1/1551 - CNA 377 09.

[13] SoftLab. *Code Generator Basic Block Analyser Function*, July 1997. Ericsson No. 1/1551 - CNA 377 08.