REGISTER STACK AND OPTIMAL ALLOCATION INSTRUCTION PLACEMENT

by

Alban Douillet

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Computer Sciences

Spring 2001

REGISTER STACK
AND OPTIMAL ALLOCATION INSTRUCTION PLACEMENT

by

Alban Douillet

Approved: _____

Guang R. Gao, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____

Jose Nelson Amaral, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____

Sandra M. Carberry, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved: _____

Conrado M. Gempesaw II, Ph.D.
Vice Provost for Academic and International Programs

ACKNOWLEDGMENTS

First I thank my advisor, professor Guang R. Gao for his unconditional support. Since my first day in his lab, he helped me in every aspect of my student life and enlightened me in wide variety of areas. His high expectations always forced to get the best of myself.

My co-advisor, professor Jose Nelson Amaral, was always there when I needed him. His patience and his quick and very clear answers to my questions were of great help during my studies. I feel lucky to have been able to work with professor Amaral, who showed dedication and passion for his work.

I thank Gerolf Hoflehner and Jim Pierce, from Intel Corporation, who provided me with the subject of the thesis. They and the entire Intel IA-64 compiler team were always full of insight advice. I greatly appreciated the discussions with everyone of them. Their kindness and patience will always be remembered.

The thesis could not have been written without the right environment. I sincerely thank all my labmates Thomas Geiger, Mark Butala, Chris Morrone, Rishi Kumar, Hongbo Yang, Mark Legutko, Andres Marquez, Adalberto Castelo, Chuan Sheng, Shreedhar Sampath, Juan Cuvillo, Fransisco Useche, Rishi Kahn, Kevin Theobald. I will always remember the intense discussions we had at lunch and our passionate games at night.

I would like to thank Andrea Michels for her patience and her kindness during the entire writing of the thesis.

Finally I wish to thank my parents Marysette and Jean-Patrick for their unwavering support and love, despite the distance. They always approved any of

my choices, even when it was hard for them. Thanks for being the best parents I could have ever dreamed of.

DEDICATION

To my parents and my brother.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Power consumption and execution speed are two of the most studied characteristics of modern processors. Optimizing the use of registers in a processor, a task known as register allocation, can increase he speed of execution of a program while reducing the power consumption. A new technique for register allocation is the use of a register stack managed by a register stack engine as implemented in the IA-64 architecture. After the compiler assigns registers to the variables of a procedure, the number of registers used must be allocated through a specific instruction: the allocation instruction.

We consider the optimal allocation instruction placement (OAIP) problem: *Given a control-flow graph G for a procedure P and a register assignment for the variables of P, insert the minimum number of allocation instructions in the procedure P in such a way that, for every path in G, the number of registers allocated is minimum.*

This thesis makes the following contributions for the solution of the OAIP problem:

- we present a formulation for the OAIP problem considering a register stack architecture with an idealized allocation instruction;
- we demonstrate that inefficiencies in the allocation of registers in the stack inducing an unnecessary growth of the register stack are caused by (1) registers allocated in a caller function that are unused while a callee function is being executed, and by (2) the allocation of extra registers for a control path that is not taken at runtime.

- we develop a linear time algorithm, MAIA (Minimum Allocation Instruction Algorithm), that addresses the OAIP problem. MAIA minimizes the number of registers allocated in each control path of the control-flow path. We conjecture that MAIA uses the minimum number of allocation instructions.

- we present different optimizations to apply to MAIA in order to accelerate the execution speed of the code generated. The optimizations consider the cost of the allocation instruction, the redundant calls to the same allocation instruction and the profiling information.

- we adapt MAIA to take into consideration the architectural features of a machine with stack registers, the Intel IA-64, including the interference between the allocation instructions, the rotating registers, and predicated execution of code.

Chapter 1

INTRODUCTION

1.1 Background

The two main features of modern processors are power consumption and execution speed. Most of the time, these characteristics are closely related. Execution speed requires power, and a decrease of the power consumption induces a decrease of the execution speed.

However there are some compiler optimizations that can improve both features at the same time. The register allocation, *i.e.* the process of assigning physical registers to variables in the source code, is a compiler optimization that can reduce power consumption and increase the execution speed of programs. Registers use a large area of the processor chip and therefore significantly contribute to the power consumption of the entire processor.

A smart register allocation uses less registers and consequently requires less power. At the same time, only the most accessed variables of programs are stored in registers, reducing memory traffic and increasing the execution speed of the programs.

Most modern architectures use a small bank of static registers visible to the user and a large bank of hidden registers dedicated to register renaming. For instance, the Intel Pentium Pro processors provides the user with 8 visible static registers while 40 hidden registers are used for register renaming. The static registers are few and need to be saved or restored at every function call. The registers used

by the register renaming are very expensive and are known to contribute for a significant part of the power consumption and the area of processor chips.

A newly defined architecture, the IA-64 architecture from Intel, uses a different mechanism to manage the registers. For the Itanium processor, besides the 32 static registers, the new architecture provides the user with a stack of 96 registers. The stacked registers are automatically spilled and restored by the hardware when the program needs more registers than currently available on the stack. The stack uses a smaller area of the processor chip while being more flexible than the registers used for register renaming.

The compiler does not need to spill or restore the stacked registers. These registers can be directly accessed with only one condition: the registers have to be allocated on the stack first. Because there is no need for spilling and restoring instructions, the source code is shorter and faster to execute. Since the area of the register stack on the die chip is smaller than for an equivalent set of hidden registers, the power consumption is also expected to be reduced.

1.2 Problem Statement

Even though the use of a register stack simplifies the compiler task, there is one challenge left: how to decide when and where to allocate the stacked registers. We assume that the register assignment already occurred. We now need to insert a specific instruction, the allocation instruction, in the source code to make sure that the stacked registers are allocated when accessed by the program. At the same time, we want to insert the minimum number of allocation instructions. Inserting too many allocation instructions would significantly decrease the execution speed of the program. Also we do not want to allocate more stacked registers than the number effectively needed.

The problem, named as the optimal allocation instruction placement (OAIP) problem, can be formulated in the following way: *Given a control-flow graph G for*

a procedure P and a register assignment for the variables of P, insert the minimum number of allocation instructions in the procedure P in such a way that, for every path in G, the number of registers allocated is minimum.

A method to solve the OAIP problem is said to be optimal if it inserts the minimum number of allocation instructions while allocating the exact number of stacked registers needed for every control path in the control-flow graph. The optimality of a method only concerns the static number of allocation instructions inserted in the code. We do not try to minimize the number of calls to the allocation instructions at run-time. We consider the static optimization, not the dynamic optimization.

1.3 Contributions

In this dissertation, we show the relationship between the global size of the register stack and the OAIP problem. Solving the OAIP problem consists in reducing the size of the stack. We identify the two components responsible for the useless growth of the register stack: function calls and overestimation of the stack register requirement.

Then we propose three different non-optimal straightforward methods to solve the OAIP problems. The methods only ensure either that the register allocation is respected, or that the minimum number of allocation instructions is used, but not both.

A linear time algorithm is presented, commented and conjectured to provide an optimal solution to the OAIP problem. Optimizations are proposed to take into account practical problems such as dynamic flows of execution, the cost of the allocation instruction of frequency of execution information. The influence of specific features of the IA-64 architecture, such as the use of predication and rotating registers, in our algorithm are studied in Chapter ??. Specific problems related to the IA-64 architecture like predication and rotating registers are also studied.

1.4 Related Work

To the best of our knowledge, the register stack concept is only used in the SPARC architecture [?]. A stack of registers is available to the user. However the number of registers allocated in a register window for each function is fix and the compiler has no control over the size of the windows. Each function has access to only one window.

The IA-64 architecture, while using the idea of overlapping register windows, chose a variable size register window where the size can be reconfigured on the fly. As a consequence, a specific instruction is needed to allocate registers in the register window: the allocation instruction.

The insertion of allocation instructions to allocate registers before they are used has not been studied before. David Wall studied the use of register windows but did not windows [?][?] but did not introduce an algorithm to allocate registers in the windows [?, ?]. He windows. He focused on the study of miss ratios, not on the efficiency of the allocation insertion algorithm.

1.5 Synopsis

The next chapter presents the register stack and the allocation instruction. The Chapter ?? introduces the three non-optimal straightforward algorithms, while Chapter ?? introduces MAIA, Minimum Allocation Instruction Algorithm, a linear supposedly optimal algorithm. Chapter ?? deals with practical optimizations for MAIA. Problems specifically related to the IA-64 architecture are considered in Chapter ??. Chapter ?? exposes some open problems that need to be solved or studied.

Chapter 2

DESCRIPTION OF THE REGISTER STACK

2.1 The Register Stack

The register stack is a pool of registers managed as a stack. The number of registers in the pool (R_T) is fixed and assumed to be implemented in hardware. Therefore the size of the stack is bounded.

The register stack is managed as a standard system stack, with frames and frame pointers. Each function has its own register stack frame and a set of frame pointers. We assume that the hardware transparently takes care of managing frames and frame pointers, since both are of no direct interest for the OAIP problem.

The current active function can only access the registers in its own frame. The registers from other frames, and consequently from other functions, are not visible. Within a frame, the registers are partitioned in three groups: the incoming registers, the local registers and the outgoing registers. When the function is called, the incoming registers contain the value of the parameters passed to the function. By symmetry, the outgoing registers are used to pass values to the functions that are going to be called. Therefore the outgoing registers of the caller function are the incoming registers of the callee function. The frames of the caller and the callee overlap over the incoming/outgoing registers of the callee/caller. The registers in the frame that are not incoming registers or outgoing registers are called local registers.

When a function is called, a new register stack frame is created. The size of the new frame is equal to the number of incoming registers. To simplify our

presentation, we assume that the hardware transparently allocate the size of the new frame according to the number of incoming registers. If the program needs to use any stack register, other than the incoming registers, it must explicitly allocate these registers before their use.

On Figure ??, the function $foo1$ uses 30 stacked registers. Four of the registers are outgoing registers when calling $foo2$. $foo2$ can access incoming register values thanks to the overlap between the two stack frames. $foo2$ uses 40 registers total.

When the register stack is full and there is not enough available stacked registers for the application, spilling of previous frames into memory occurs. Thus each function can have access to at most R_T stacked registers. When the function exits, previous frames are restored if possible and visible again. The operations of frame spill and restore are transparently executed by the hardware. They require no explicit software intervention.

The creation of a new register stack frame is a register renaming operation. Two functions referencing the same stacked register number are not necessarily using the same physical register in the stack. The hardware maps the stacked registers as seen by the function to the corresponding physical registers in the stack. Assume that the first available register in the stack when the function issues an allocation instruction is register R_S. Then the register R_1 of the procedure is mapped to the physical register R_S. The register R_k of the procedure will be mapped to $R_{(S+k) mod R_T}$. The wrapping at the end of the physical stack works because functions cannot allocate more than R_T registers.

The two functions $foo1$ and $foo2$ on Figure ?? use respectively 30 and 40 stacked registers. $foo1$ has no incoming registers, 4 outgoing registers and 26 local registers. $foo2$ has no outgoing registers, 4 incoming registers and 36 local registers. The 4 outgoing registers of $foo1$ overlaps with the 4 incoming registers of $foo2$.

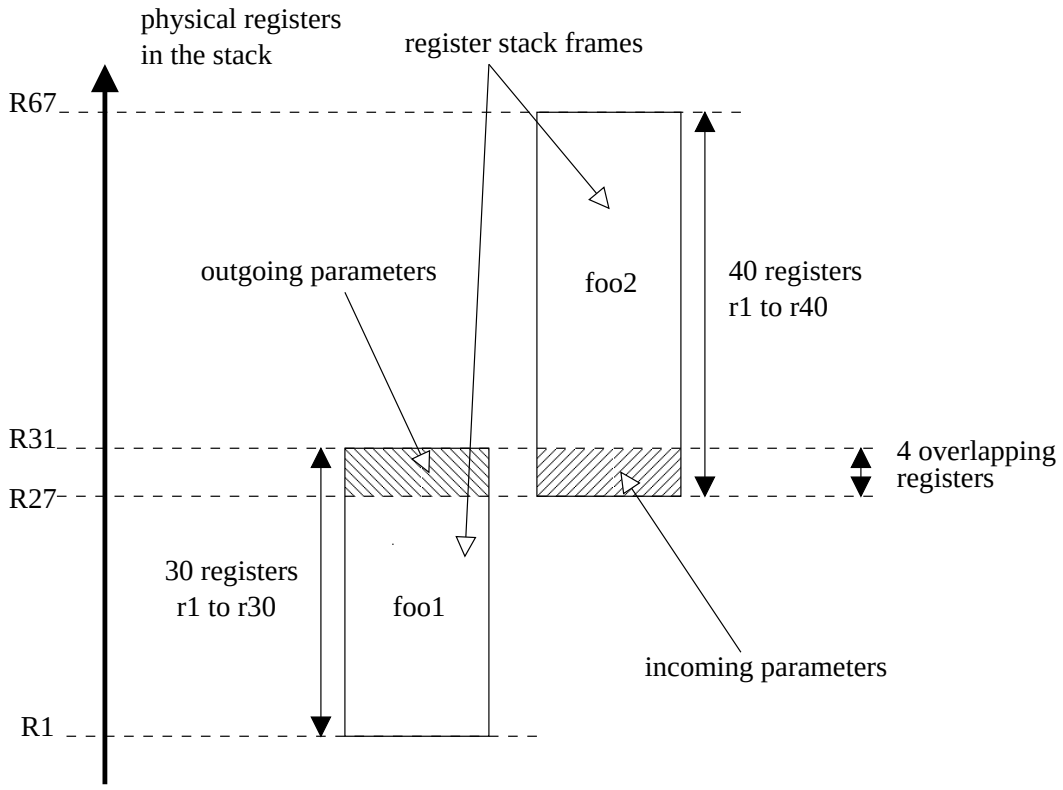


Figure 2.1: Register Stack and Register Stack Frames.

Although $foo1$ and $foo2$ access the registers on the stack using the same stacked register numbers, the mapping is different for the two functions. For instance, when $foo1$ wants to read the value in $r10$, the physical register accessed is $R10$. But, when $foo2$ wants to read the value in $r10$, the physical register accessed is $R36$. Note that on Figure ?? R_T has to be greater than 66.

The stacked registers are complementary to the usual static registers. Stacked registers are only visible to the current function, need to be allocated and are restored and spilled transparently by the hardware. Whereas static registers are fewer, are shared by all the functions, are always available, need to be manually spilled and restored following specific software convention rules (callee/caller saved for instance).

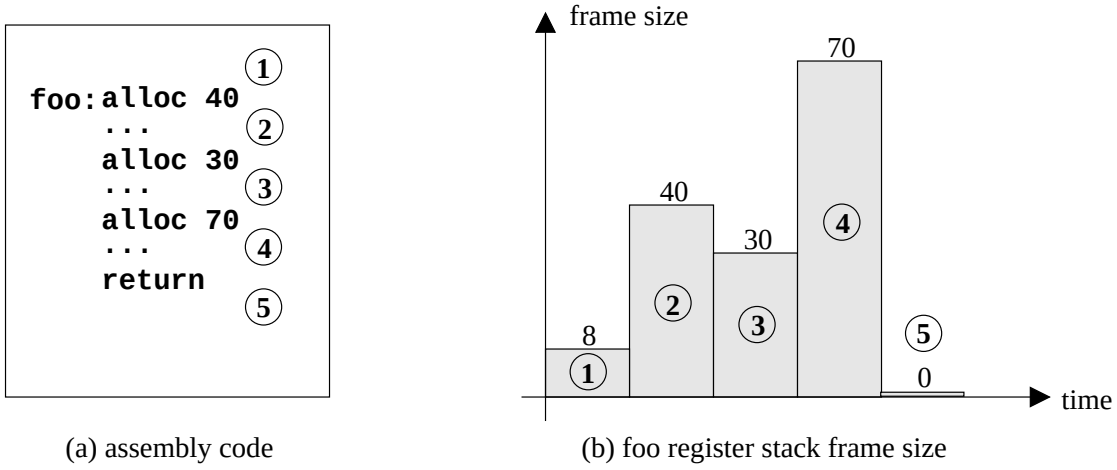
2.2 The Allocation Instruction

Since most of the operations on the register stack are hardware-controlled, the software interface is reduced to only one instruction: the *allocation instruction*. The instruction is used to specify the size of the current register stack frame.

alloc *stack_frame_size*

Figure 2.2: Syntax of the allocation instruction.

For the development of the base algorithm we will assume that the allocation instruction, called alloc, has a single parameter, stack_frame_size, that specifies the number of registers to be allocated in the current register stack frame. When multiple allocations are executed in the same procedure, every allocation overrides the previous ones. Therefore an *alloc* 40 followed by an *alloc* 30 allocates 30 registers, not 70, to the current stack frame (cf. Figure ??). The allocation instruction value must be between 0 and R_T.



(a) assembly code (b) foo register stack frame size

Figure 2.3: Allocation instruction effects on the register stack.

In this paper, we assume that the allocation instruction is ideal, *i.e.* the instruction can be used at any time and does not interfere with other instructions.

The effects of the allocation instruction are instantaneous. We present some algorithms using this ideal allocation instruction in Chapters ?? and ??. Then we will consider a real allocation instruction as implemented in the IA-64 architecture (Chapter ??).

2.3 The Cost of the Allocation Instruction

The register stack is managed by the hardware. Anytime there is not enough registers to be allocated for the current function, registers already allocated by previous functions are transparently spilled and available for other uses. Even if the process of automatic spills and restores simplifies the work of the compiler by avoiding the need to insert specific spill and restore instructions, we would like to avoid expensive uses of the allocation instruction.

We may encounter the case where the spilling and restoring of stacked registers could be avoided. The compiler may have overestimated the stacked register requirement and allocated too many registers. We need to identify the parameters that influence the behavior of the register stack.

Since the allocation instruction is the only way for the compiler to communicate with the register stack, we only have to focus on this instruction.

As long as the stack is not filled, we assume the allocation instruction is virtually costless. We can allocate and deallocate registers without any further considerations. However, as soon as all the registers in the stack are in use and the current function needs to allocate more stacked registers, we can expect some latencies from the allocation instruction: the spill and restore operations might increase the memory traffic and require the processor to stall. Our goal is to minimize the occurrences of such situations.

Depending of the implementation of the allocation instruction, costs may vary. To simplify, we represent the allocation instruction cost as an binary function. The instruction has a cost of 0 as long as the total number of stacked registers in

use is less than the physical register stack size R_T. When the number of registers exceeds R_T and spills of stacked registers is necessary, the allocation instruction has an arbitrary cost of 1, no matter how many stacked registers are to be spilled. If registers need to be restored, the allocation instruction is also considered as having a cost of 1.

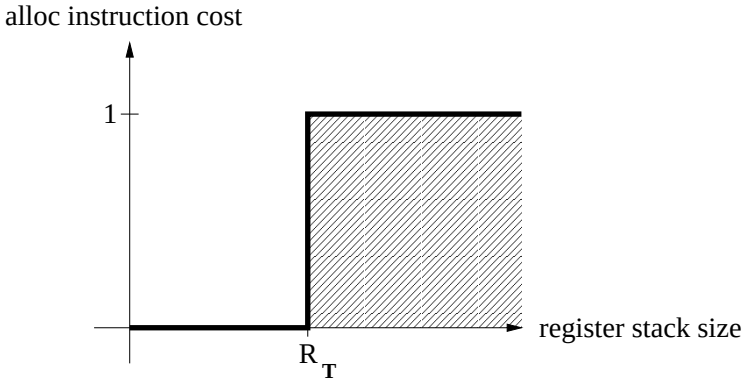


Figure 2.4: Cost of the allocation instruction depending on the number of stacked registers currently in use.

Therefore to reduce the cost of the allocation instruction, we need to limit the size of the virtual register stack. We want to keep the virtual register stack size below the physical register stack size value R_T. In other words, *we want to minimize the growth of the register stack.*

However, since register allocation already occurred, the only way to limit the growth of the register stack is to allocate only the registers needed and avoid useless allocations that could cause spills and restores. We identify the two sources of stacked register waste:

Overestimated allocation in the current routine We may want to allocate the maximum number of stacked registers we might need. However if we actually do not need the registers, because of a very specific control path in the control-flow graph, the allocated registers are *never* used. We wasted the registers. Note that once the registers have been allocated, deallocation is worthless because the earlier overallocation already caused the stacked registers to be spilled. Also, if all the registers allocated have been used at least once, then

there is no waste, even if later on only a subset of the allocated registers are used.

Frozen allocation of the previous register stack frames Every function call allocates a new register stack frame on the register stack. As a consequence, all the registers allocated for the previous function are frozen, used or not. In other words, a new register stack frame freezes the size of the previous stack frame. We do not have access to the previous frames anymore. If some registers allocated to previous functions were not used, they are wasted.

The two origins of stacked register waste are orthogonal.

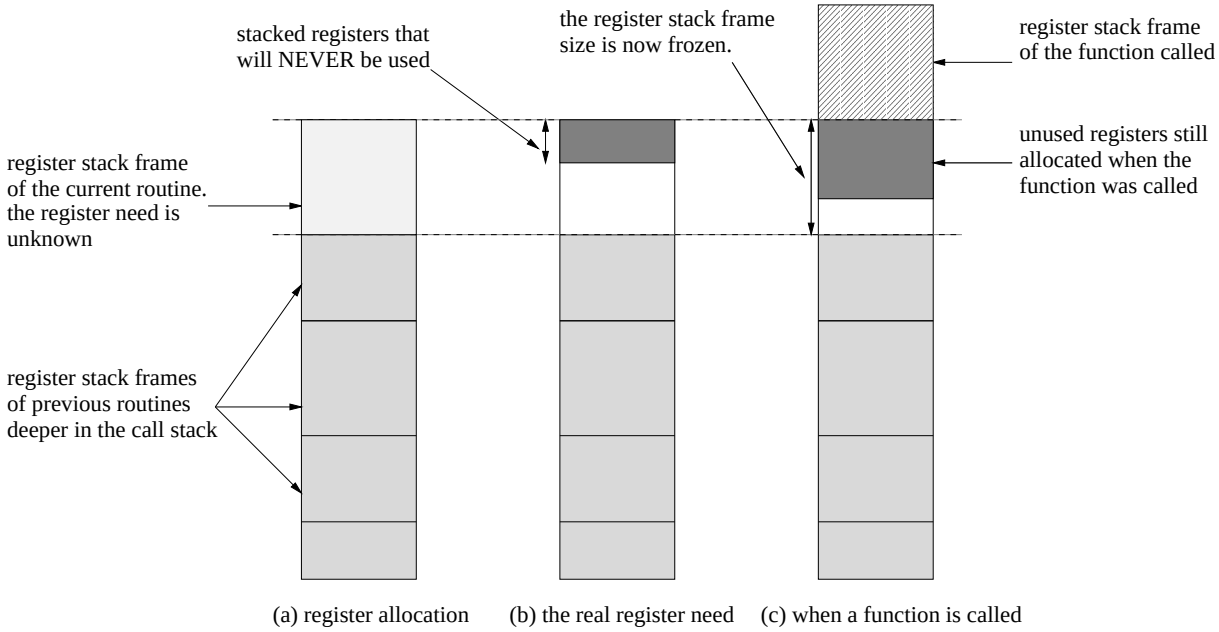


Figure 2.5: The two origins of the growth of the virtual register stack.

On Figure ??, we observe the behavior of the register stack. In (a), the current routine has already allocated registers in the register stack frame. Other stack frames from previous functions in the call stack are allocated too. In (b), we discover that we actually allocated too much because of a specific control path in the control-flow graph. There are unused registers. The current register stack frame and the virtual register stack size were not worth increasing that much. In (c), we now call another function. Unfortunately, at that point, we were using only a

small percentage of the registers allocated in the register stack frame. The unused registers of the caller function are now wasted and cannot be deallocated from the callee function. The function calls indirectly increased the size of the virtual register stack.

In order to achieve optimality, an algorithm must address the two sources of the stacked register waste: current registers in use and frozen register stack frame size due to function calls. The classification in the next section will help to identify which uses of the allocation instructions contributes to the growth of the register stack.

2.4 Allocation Instruction Classification

The following classification of the allocation instruction is only based on the position of the allocation instruction in the code and the status of the register stack when the allocation instruction is called. There is only one allocation instruction, but for a better understanding and for clearer explanations, some classification was necessary.

2.4.1 Required Allocation Instruction

Except for the case when a routine does not need any stacked register, there must be at least one allocation instruction at the beginning of each routine. The instruction is the first allocation instruction to appear in the code of the routine. We call it: *required allocation instruction*.

The required allocation instruction is necessary and is found in every routine. The instruction is directly linked to the first type of stacked register waste: a bad prediction, and the register stack frame size is increased too much.

2.4.2 Expanding Allocation Instruction

Every allocation instruction that increases the current register stack frame size is called *expanding allocation instruction*. Note that the required allocation instruction is not considered as so, because the current register stack frame has not been allocated yet.

The expanding allocation instruction is directly linked to the first type of stacked register waste: overestimation. The required and expanding allocation instructions are the only two types of allocation instruction that increase the register stack frame size.

2.4.3 Shrinking Allocation Instruction

The opposite allocation instruction to the expanding allocation instruction is called *shrinking allocation instruction*.

Note that the shrinking type allocation instruction is useless and ideally should not appear in the code. Since the registers have already been allocated to the current register stack frame, we do not gain anything into deallocating some stacked registers.

The shrinking allocation instruction is not responsible for wasting stacked registers. However, because the instruction is worthless, we would rather not use it.

2.4.4 Preallocation Instruction

In order to deallocate useless registers before calling another function, we introduce an artificial shrinking allocation instruction called the *preallocation instruction*.

The preallocation instruction is necessary to reduce the number of unused registers due to function calls and is used as a prevention against the second type of wasted registers.

2.4.5 Postallocation Instruction

When a preallocation instruction is used, too few registers are now allocated when the function returns. We need to reallocate the deallocated registers using a specific expanding allocation instruction: *the postallocation instruction*.

The postallocation instruction is a necessary allocation instruction.

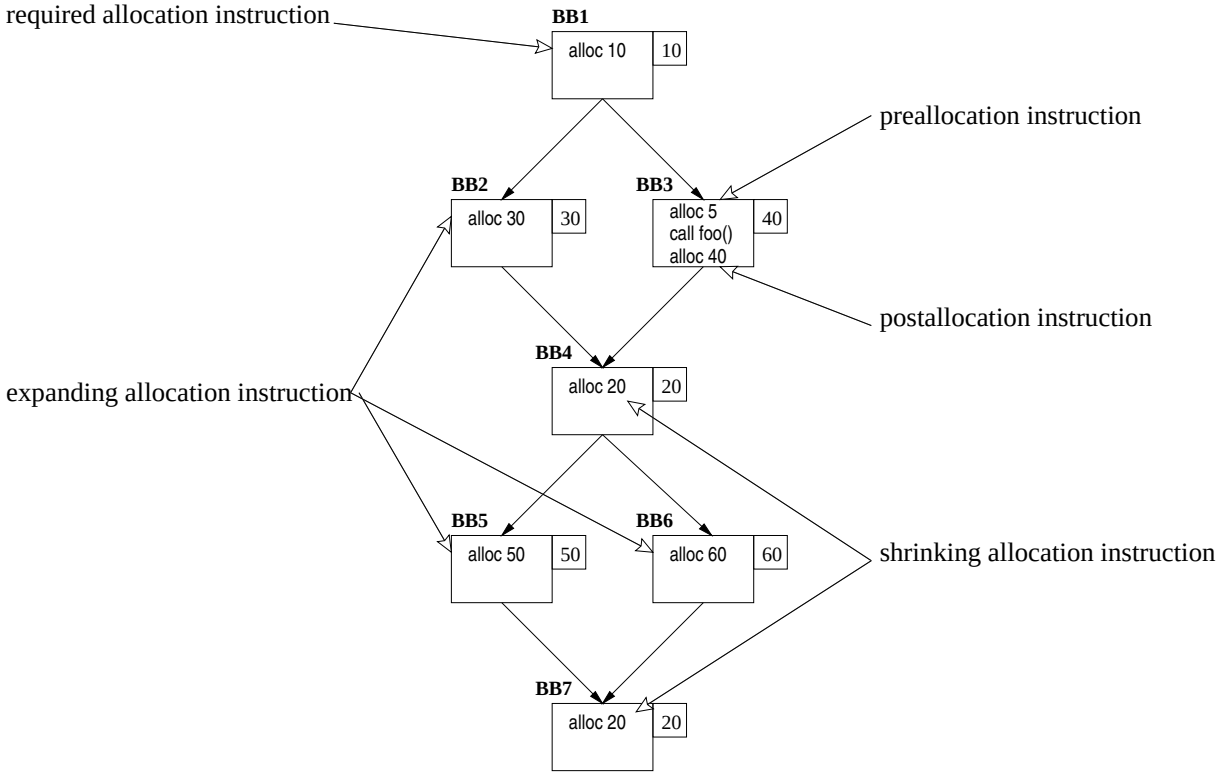


Figure 2.6: Example of different types of allocation instructions.

On Figure **??**, we can observe all the types of allocation instruction. The $alloc10$ in $BB1$ is the first allocation in the control-flow graph and therefore is the required minimum allocation instruction. The allocation instructions in $BB2$, $BB5$ and $BB6$ are expanding instructions because they increase the register stack size compared to the parents of the basic block of the allocation instructions. The allocation instructions on $BB4$ and $BB7$ are shrinking allocation instructions because they decrease the register stack size (compared to the parents in the control-flow

graph). The two allocation instructions in $BB3$ are preallocation and postallocation instruction because *alloc* 5 has been inserted right before the function call and *alloc* 40 right after.

Note that the same allocation instruction can be a shrinking and an expanding allocation instruction depending on which control path was taken to reach the basic block that contains the instruction.

2.5 Notations

Let us consider a node A in a control-flow graph G.

- $succ(A)$ is the set of direct successors of A in G. In Figure **??**, $succ(BB4) = \{BB5, BB6\}$.
- $pred(A)$ is the set of direct predecessors of A in G. In Figure **??**, $pred(BB4) = \{BB2, BB3\}$.
- $need(A)$ is the highest number of stacked registers alive at any point of the basic block A. $need(A)$ is the width of the fat point of the basic block A. In Figure **??**, $need(BB3) = 40$.
- $alloc(A)$ is the value of the parameter of the allocation instruction in A. The value can be greater but never less than $need(A)$, except if the value is 0. If $alloc(A) = 0$, then A does not have any allocation instruction. The stacked registers of the basic block A have been allocated by a previous allocation instruction in G. In Figure **??**, $alloc(BB4) = 20$

In our examples, the basic block we are interested in will be called A. The direct predecessors of A will be called $P1$, $P2$,... and direct successors $S1$, $S2$,...

The control-flow graphs are represented using large and small boxes. The large boxes represent basic blocks. Only the allocation and the function call instructions are shown. The other instructions are hidden. The small boxes attached

to the basic block boxes contains the highest number of stacked register used by the corresponding basic block ($need(A)$). In Figure **??**, two allocation instructions and one function calls appears in $BB3$. $need(BB3)$ is equal to the parameter value of one of the allocation instructions present in $BB3$.

2.6 Register Allocation Optimality

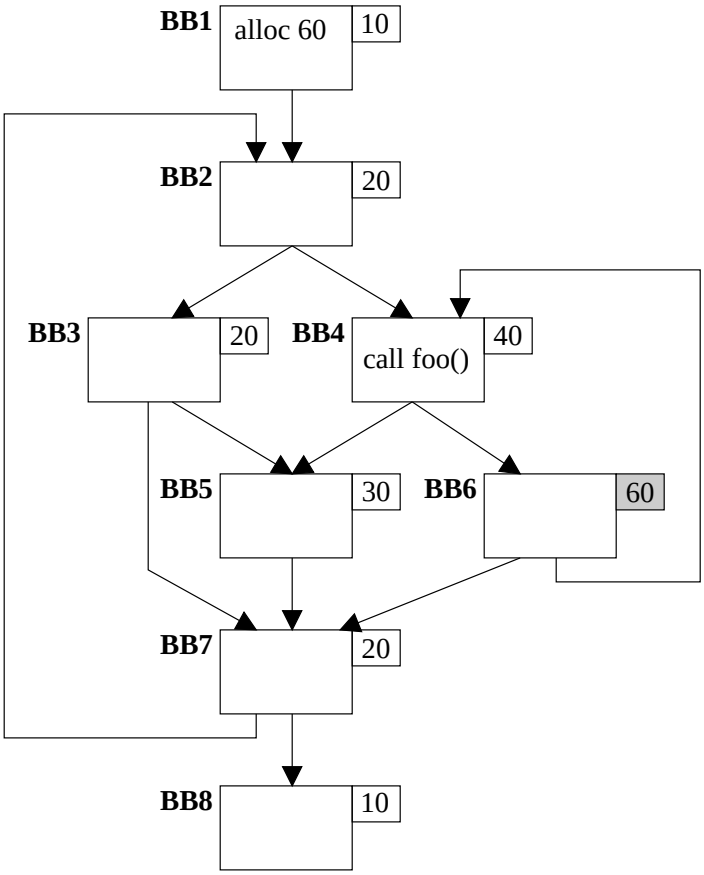


Figure 2.7: Example of a non-RA optimal procedure.

We saw that the number of stacked register spills/restores is directly linked to the size of the register stack. Consequently, we want to avoid useless register allocations. We saw there were two types of such allocations: overestimation of

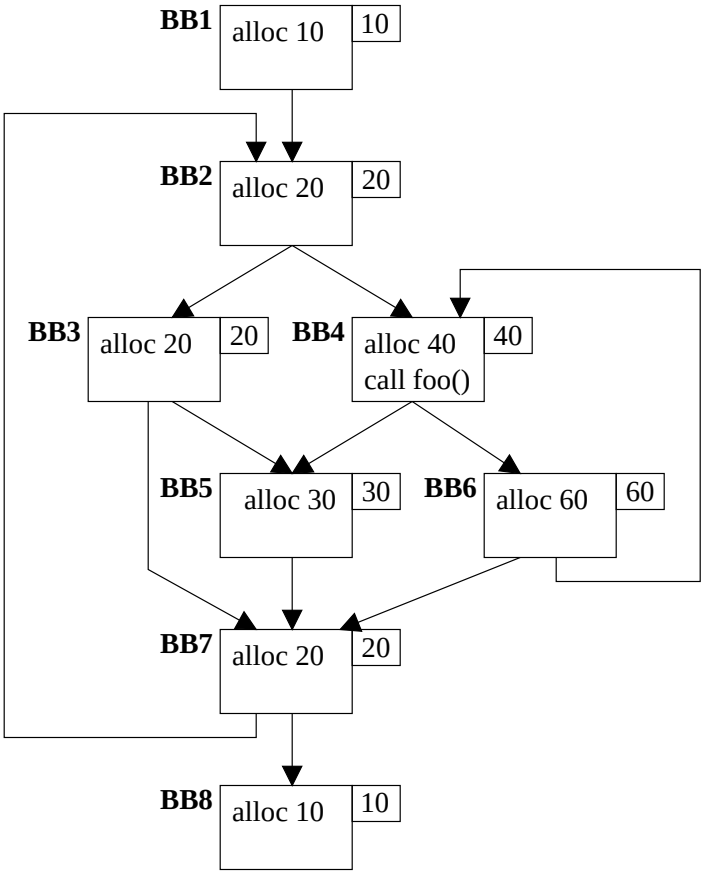


Figure 2.8: Example of an RA optimal procedure.

the current register stack frame and frozen allocation in the previous register stack frames.

Therefore, for the control-flow graph and a register assignment of a given procedure P, the number of stacked registers allocated for the variables of P is optimal if and only if the number of wasted stacked registers is minimized.

The number of wasted registers is minimized if and only if the two types of useless allocations are minimized. The overestimation is minimized by a fine-grain allocation (for any control path in the control-flow graph, we allocate the exact number of registers needed). The frozen allocation is minimized by the use of preallocation and postallocation instruction (we shrink the register stack frame

as much as possible just before any function call and restore the frame size to the original value afterwards).

Definition 2.1 (RA Optimality) *Given a control-flow graph G of a procedure P and a register assignment for the variables of P, the number of stacked registers allocated for the variables of P is minimized if and only if:*

(i) for every control path in G, the minimum number of stacked registers is allocated,

(ii) and for every function call in P, the current register stack frame is shrunk to the minimum.

The procedure is then said to be Register Allocation (RA) optimal.

The first bullet of Definition ?? means that, given a specific control path in G, we allocate exactly the maximum number of stacked registers needed at any point of the execution of the control path.

The procedure, whose control-flow graph is represented on Figure ??, is not RA optimal. For instance, if the flow of execution is composed of the basics blocks $BB1$, $BB2$, $BB3$, $BB5$, $BB7$ and $BB8$, then we allocated 60 stacked registers and used only 20. The first bullet of the RA optimality definition is not verified.

However, the same procedure with different allocation instructions represented on Figure ?? is RA optimal. Every allocation instruction ensures that, for every control paths in the control-flow graph, we allocate only the number of stacked registers needed along the path. For instance, the path $BB1-BB2-BB3-BB7-BB8$ requires only 20 stacked registers at most and only one 20 stacked registers are allocated. Also the current register stack frame is shrunk to the minimum before the only function call in $BB4$. Note that only allocation instructions and function calls are shown in the basic blocks. The two conditions of Definition ?? are verified.

2.7 Optimal Allocation Instruction Placement

The Optimal Allocation Instruction Placement (OAIP) problem is formulated in the following way: *Given a control-flow graph G for a procedure P and a register assignment for the variables of P, insert the minimum number of allocation instructions in the procedure P in such a way that, for every path in G, the number of registers allocated is minimum.*

Part of the OAIP problem consists in solving the register allocation problem described in the previous section. The second part consists in minimizing the number of allocation instructions used to achieve optimality for the register allocation problem. Therefore we can define optimality for the OAIP problem.

Definition 2.2 (OAIP optimality) *A procedure P is said to be OAIP optimal for a given register assignment for the variables of P if the following conditions are true:*

(i) P is RA optimal and if

(ii) P uses the minimum number of allocation instructions to achieve (i).

The example described on Figure **??** is not OAIP optimal, although the RA optimality is reached. The number of allocations used to achieve the RA optimality is not minimum. For instance, the allocation instruction of $BB3$ could be removed without any impact on the register allocation. The allocation instruction of $BB2$ already allocated the necessary stacked registers in the corresponding control path. Figure **??** presents a OAIP optimal solution for the same procedure. There is only one allocation per control path in the control-flow graph. No allocation instruction can be removed without sacrificing the RA optimality.

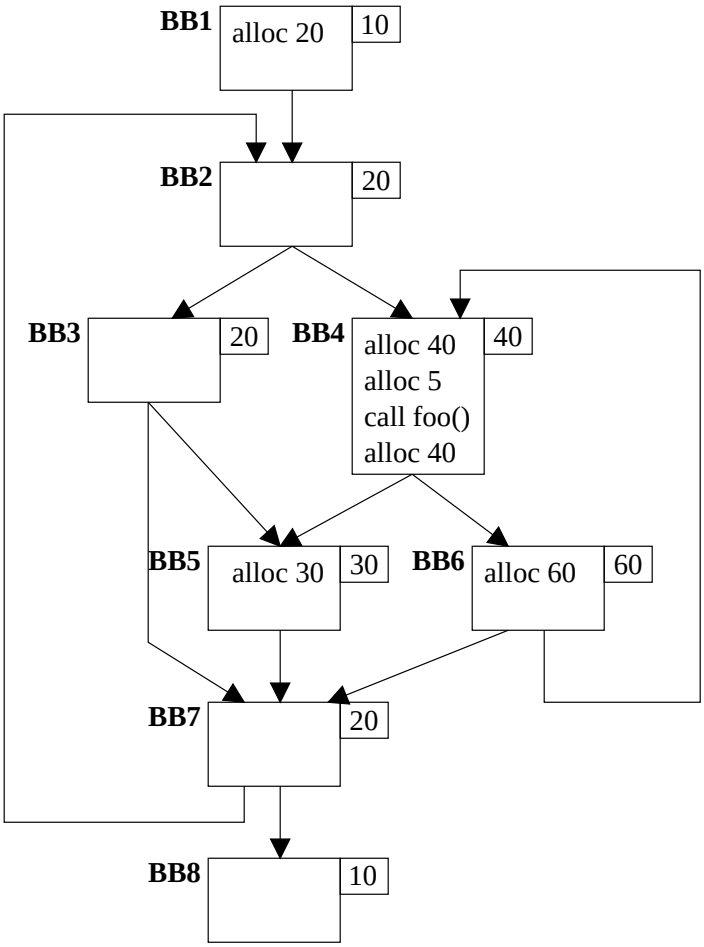


Figure 2.9: Example of a OAIP optimal procedure.

Chapter 3

NON-OPTIMAL STRAIGHTFORWARD ALGORITHMS

We present three non-optimal straightforward algorithms for the OAIP problem that we will use as reference for MAIA presented in chapter ??. The algorithms presented in this chapter are important to understand the philosophy behind MAIA. The minimum requirement solution is the simplest solution we may think about. The around-call shrinkage solution only deals with the stacked register waste caused by frozen allocations. The brute-force solution minimizes the waste of stacked registers but uses too many allocation instructions.

3.1 The Minimum Requirement Solution

3.1.1 Overview

The minimum requirement solution to the OAIP problem is straightforward. We simply compute the maximum number of registers used anywhere in the function and allocate that number on the first basic block executed. The value is computed by examining every basic block requirement and taking the maximum. The allocation instruction is the first instruction of the routine.

The solution is called minimum requirement solution because every function requires at least one allocation instruction to allocate the stacked registers and the minimum requirement solution inserts a single allocation instruction for every function.

The minimum requirement solution is the implementation chosen for the IA-64 architecture. The IA-64 allocation instruction has been designed to allocate

stacked registers once and for all for every function. However the allocation instruction has other uses and may appear more than once in the code (for rotating registers in a loop for instance).

On Figure ??, we see that only one allocation instruction has been inserted. The number of stacked registers allocated is equal to the highest basic block stacked register requirement: 60 registers for $BB6$. For every control path in the control-flow graph, there is always enough registers allocated on the stack. Note that more registers could be allocated, but they would not be used in any of the control path, and therefore would be wasted.

3.1.2 Optimality

The minimum requirement solution is obviously not OAIP optimal. The minimum number of allocation instructions has been reached, but depending on which control path of the control-flow graph is executed, we may allocate more registers than needed: the RA optimality has not been achieved.

On Figure ??, if the flow of execution only follows the left branches *e.g.* the basic blocks $BB1$, $BB2$, $BB3$, $BB7$ and $BB8$, then we have allocated an excess of 10 registers.

3.2 The Around Call Shrinkage Solution

3.2.1 Overview

The around call shrinkage of the stacked register frame directly prevents the unused stacked registers allocation from freezing when a function call occurs (cf. Chapter ??). Every time a function call appears in the code, we surround the call instruction with a preallocation instruction and a postallocation instruction. We use the preallocation instruction to shrink the current register stack frame to the minimum by deallocating unused registers and the postallocation instruction to reallocate the registers that had been deallocated by the preallocation instruction.

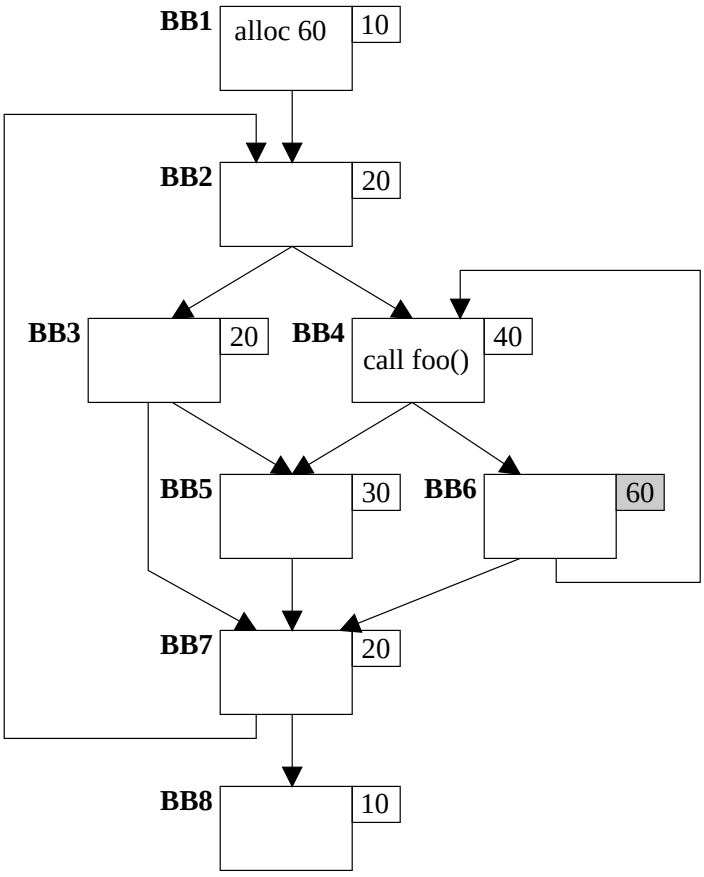


Figure 3.1: Minimum Requirement Example.

The around call shrinkage solution is based on the minimum requirement solution. The required allocation instruction cannot be avoided.

In order to make the algorithm simpler, we want to have only one allocation instruction per basic block. Therefore we may to split existing basic blocks into several basic blocks. For instance, a basic block with 3 allocation instructions will be split into a linear chain of 3 basic blocks. The split will occur right before each allocation instruction. Thus the allocation instructions are always the first instructions on the basic blocks. The split is necessary for MAIA presented in Chapter **??**.

On Figure **??**, we look for the function calls in the control-flow graph. Then we insert the preallocation and postallocation instructions in $BB4$ and split the basic block into two blocks $BB4a$ and $BB4b$. The preallocation instruction is the first allocation of $BB4a$. The postallocation is the first instruction of $BB4b$. The value of the preallocation instruction is the number of greatest stacked registers currently in use when the function call occurs (here 5). The postallocation instruction value is the number of stacked registers allocated before the corresponding preallocation instruction (60).

3.2.2 Optimality

Only the second hypothesis of the RA optimality definition Definition **??**) is fulfilled. The around call shrinkage solution does not ensure that the minimum number of stacked registers is allocated for every control path in the control-flow graph. The around call shrinkage solution is not RA optimal, and therefore cannot be OAIP optimal.

For instance, the control path $BB1$-$BB2$-$BB3$-$BB7$-$BB8$ on Figure **??** uses only 20 stacked registers maximum. But the required allocation instruction, the only allocation instruction in the control path, allocates 60 stacked registers. The around call shrinkage solution is not optimal.

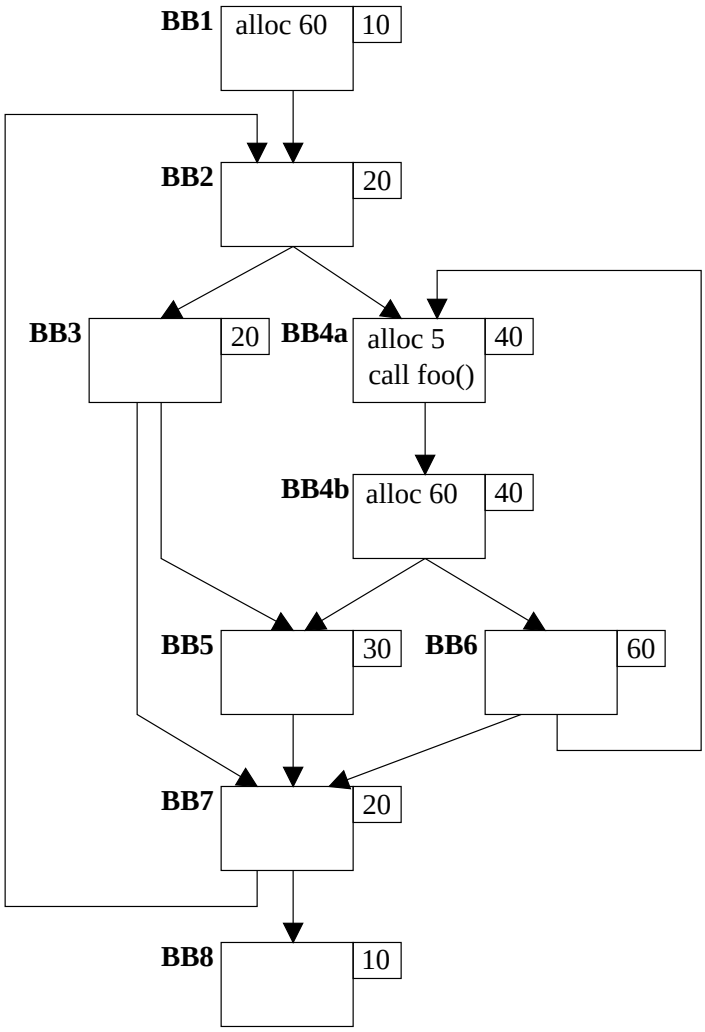


Figure 3.2: Around Call Shrinkage Example.

We may note that the required allocation instruction could allocate only 40 registers and the postallocation instruction 60. Nevertheless, since the solution is based on the minimum requirement solution, the around call shrinkage solution cannot achieve such results without more advanced optimizations.

3.3 The Brute-Force Solution

3.3.1 Overview

The brute-force solution takes an opposite approach from the minimum requirement solution. Instead of inserting the minimum number of allocation instructions, the brute-force solution inserts one allocation instruction per basic block. The value of the allocation instruction is equal to the stacked register requirement of the basic block. The solution is based on the around call shrinkage solution.

Because of existing preallocation and postallocation instructions from the around call shrinkage solution, we split basic blocks to maintain one allocation instruction per basic block. The first instruction of every basic block is always an allocation instruction then.

On Figure **??**, we consider each basic block of the control-flow graph. We look at the register requirement and insert the corresponding allocation instruction. The algorithm is linear.

Although the brute-force solution seems to be very inefficient, the solution is used as a starting point for the MAIA solution.

3.3.2 Optimality

The brute-force solution is not be OAIP optimal in general. The maximum number of allocation instructions, one per basic block, is inserted. There may be a different way to insert the allocation instructions that would use less instructions while allocating the same number of stacked registers.

However the brute-force solution is RA optimal. We allocate the minimum number of stacked registers needed by each basic block. We shrink or expand the register stack frame only when necessary. The two conditions of the RA optimality definition (Definition ??) are fulfilled.

On Figure ??, we notice that we never allocate more stacked registers than needed by every basic block. However the brute-force solution does not solve the OAIP problem because the number of allocation instructions is not minimal. For instance we could have removed the allocation instruction of $BB8$. The register requirement of $BB8$ is covered by the allocation instruction in $BB7$.

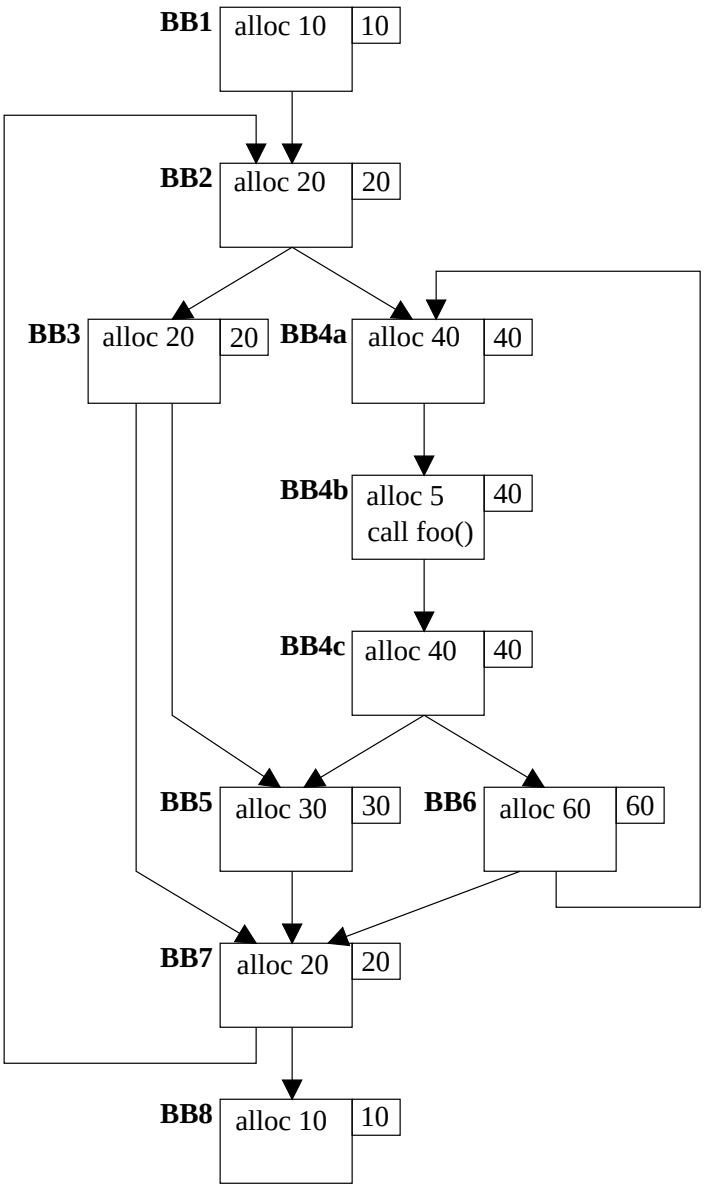


Figure 3.3: Brute Force Example.

Chapter 4

MAIA, MINIMUM ALLOCATION INSTRUCTION ALGORITHM

4.1 Problem statement

All the solutions presented in Chapter ?? do not give an optimal solution to the OAIP problem. Either the number of allocation instructions is minimized (Section ??), or the the number of stacked registers allocated is minimized (Section ??), but not both. The around call shrinkage solution (Section ??) is a simple compromise and only considers one part of the problem: the register stack increase due to function calls.

In this section, we present an algorithm that efficiently addresses the OAIP problem: *Given a control-flow graph G of a procedure P and a register assignment for the variables of P, insert the minimum number of allocation instructions in the procedure P in such a way that, for every path in G, the number of registers allocated is minimum.* Our solution is conjectured to be optimal in the sense that we only allocate the stack registers that are needed for allocate for the stacked registers we need for a given control path while using a minimum number of allocation instructions

Note that we assume that all the paths in the control-flow graph are equally likely to be executed. Later we will introduce control-flow graphs with frequency of execution information. Also we assume that there is no predicated instruction and that the cost of the allocation instruction is ignored.

We also restrict to programs with *reducible flow-graphs* only [**?**]. Therefore there are only natural loops and the edges of the control-flow graph can be partitioned into two groups: forward edges and back edges. From now on, we only consider the forward edges. The control-flow graphs we consider are the actual control-flow graphs from which we removed the back edges. The back edges and the natural loops will be considered later.

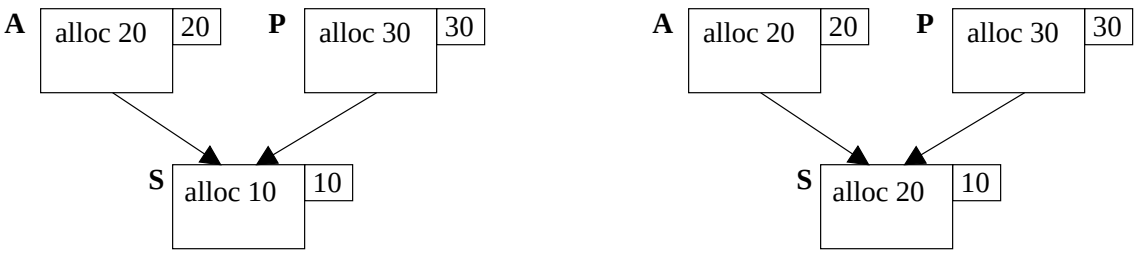
4.2 Rule Definitions

The idea behind MAIA is to start with the brute-force solution (Section **??**) and to remove all the unnecessary allocation instructions. For every control path in the control-flow graph, we want the register stack frame size to only increase as the stacked register demand grows. We shrink the frame only before function calls. Therefore the register allocation increases along every control path in the control-flow graph. We use three rules to obtain an efficient allocation instruction placement: the downward propagation rule, the upward propagation rule and the reduction rule.

Definition 4.1 (Downward Propagation Rule) *Given a basic block A and $k = min_{P_i \in Pred(A)} alloc(P_i)$. If $alloc(A) < k$ then $alloc(A) = k$.*

The downward propagation rule propagates downward the information of a higher allocation value. Given a basic block B, if a dominant basic block of B has a higher allocation value, then B will be informed. The downward propagation phase acts a pre-pass to the two others rules. The downward propagation rule provides allocation information to the basic blocks below in the control-flow graph for a finer allocation instruction placement.

In Figure **??**(a) each basic block has an allocation instruction. The basic block A has only one direct successor S and $alloc(A) > alloc(S)$. Moreover all the direct predecessors of S have an higher allocation value. In any case, when a flow



(a) Before downward propagation

(b) After downward propagation

Figure 4.1: Example of the downward propagation rule

of execution reaches S, at least 20 stacked registers will be allocated. Propagating downward the allocation instruction of A does not hurt the register allocation. However the allocation instruction of P cannot be propagated downward to S. Otherwise, if the flow of execution comes from A to S, we would allocate more stacked registers than required. The result of the propagation is shown on Figure **??**(b).

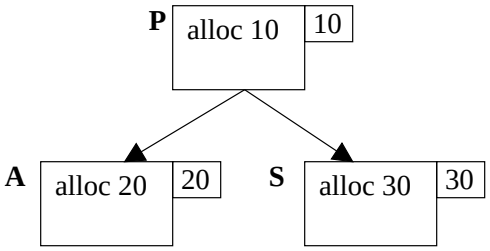
Definition 4.2 (Upward Propagation Rule) *Given a basic block A, if, for every direct predecessor P of A, all the following conditions are true:*

(i) $alloc(A) \geq alloc(P)$

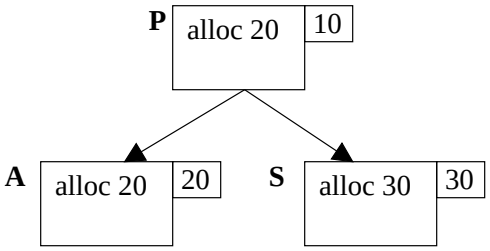
(ii) $\forall S \in succ(P)$, $alloc(A) \leq alloc(S)$

then $alloc(P) = alloc(A)$*, for every direct predecessor P of A.*

Intuitively, the upward propagation rule considers that, since for any control path starting from a predecessor of A we have to increase the register stack frame size, we would better increase the size before the flow of execution reaches A and maybe avoid the execution of some unnecessary allocation instructions. The allocation instruction is not propagated upward if another parent B of A has a lower allocation value. If a flow comes from B to A, the allocation instruction may be needed.



(a) Before upward propagation



(b) After upward propagation

Figure 4.2: Example of the upward propagation rule

In Figure ??(a) each basic block has an allocation instruction. All the direct successors of P have a higher allocation need. In Figure ??(b), the basic block A, which has the minimum allocation needs among the direct successors of P, does not need the allocation instruction anymore. The allocation instruction of A is propagated to the basic block P, which now allocates 20 stacked registers instead of 10. The redundant allocation instruction in A is removed by the *reduction rule*.

Definition 4.3 (Reduction Rule) *Given a basic block A, if all the following conditions are true:*

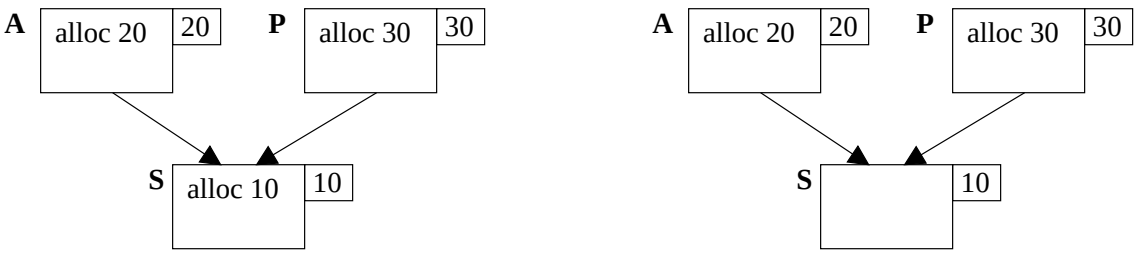
(i) $alloc(A)$ is not a preallocation instruction

(ii) for every direct predecessor P of A, $alloc(P) \geq alloc(A)$

then remove $alloc(A)$.

On the other hand, the reduction rule considers that the registers have already been allocated in the predecessors of A and do not need to be allocated again in A. If all the direct predecessors of A have a higher allocation value, then the allocation instruction of A is reduced.

In Figure ??(a) each basic block has an allocation instruction. All the direct predecessors of S have a higher allocation need. Therefore S does not need the allocation instruction. The registers have already been allocated. In Figure ??(b), after



(a) Before reduction **(b)** After reduction

Figure 4.3: Example of the reduction rule

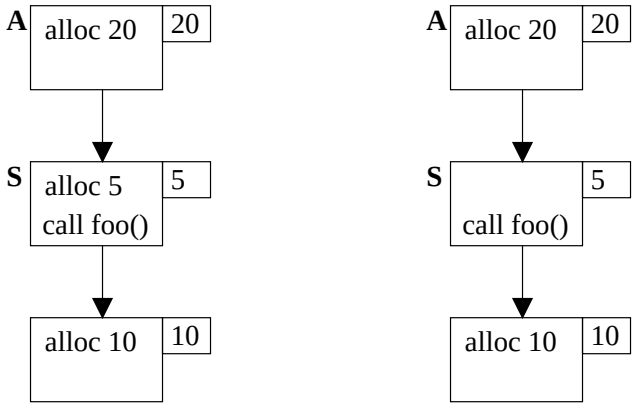
reduction, the allocation instruction from S has been removed. The two allocation instructions of A and S have been reduced to a single allocation instruction (we assume that A was considered before P by the algorithm).

To complete the algorithm we have to consider the function calls. We saw in Section ?? that the function calls must be surrounded by two allocation instructions: the preallocation instruction to shrink the register stack frame to the minimum number of registers necessary for the call and the postallocation instruction to reallocate the deallocated registers.

The preallocation instruction cannot be avoided or removed. We do not want to allocate more registers than the ones required for the function call. However we may move the instruction earlier in the control-flow graph if possible. The only condition that needs to be satisfied is that, when the control path reaches the function call, only the necessary registers are allocated. Therefore the preallocation cannot be reduced or be the recipient of a propagation.

In Figure ??, we can see that the reduction of preallocation instruction causes 20 registers to be allocated in basic block A before the function call, although only 5 registers were necessary in S.

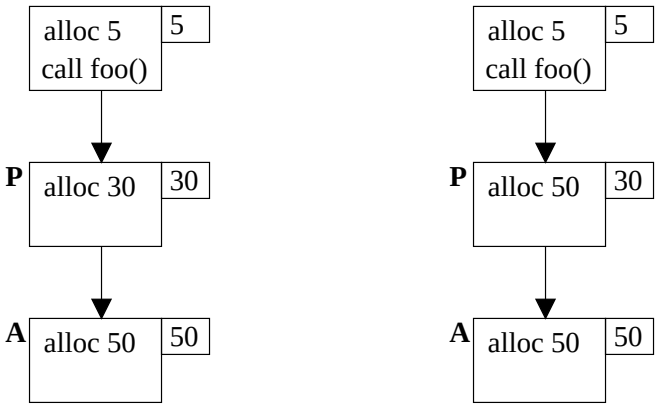
The postallocation instruction must appear after the function call to reallocate the registers that the preallocation instruction might have deallocated. Therefore the postallocation instruction cannot be propagated across the function call.



(a) Before reduction **(b)** After reduction

Figure 4.4: The preallocation instruction cannot be reduced.

Consequently we must forbid any propagation of the postallocation instruction. However the instruction can be replaced by another allocation instruction that is propagated. The replacing instruction is then declared as the new postallocation instruction.



(a) Before propagation **(b)** After propagation

Figure 4.5: The postallocation cannot be propagated, but the allocation value can be replaced by a propagated allocation instruction value.

In Figure **??**, the postallocation instruction cannot be propagated in the place of the preallocation instruction. The propagation would break the entire purpose of

these the preallocation and postallocation instructions. However the *alloc* 50 can be propagated in place of the current postallocation instructions. the *alloc* 50 must then be declared as a postallocation instruction in order not to be propagated across the function call.

For the same reasons, a preallocation instruction cannot be downward propagated. The instruction must appear before the function call. If the preallocation instruction appears after the function call in the code, the all purpose of shrinking the current stack frame before the call is broken.

Consequently the allocation instructions surrounding function calls have a special status that could be marked with a flag. Also, the propagation and reduction rules would apply only if the flags are correct. Which brings us to a revised version of the definitions of the rules:

Definition 4.4 (Downward Propagation Rule) *Given a basic block A and $k = min_{P_i \in Pred(A)} alloc(P_i)$ such that the allocation instruction in P_i is not a preallocation instruction. If all the following conditions are true:*

(i) $alloc(A)$ *is not a preallocation instruction*

(ii) $alloc(A) < k$

then $alloc(A) = k$

Definition 4.5 (Revised Upward Propagation Rule) *Given a basic block A, if, for every direct predecessor P of A, all the following conditions are true:*

(i) $alloc(A)$ *is not a postallocation instruction*

(ii) $alloc(A) \geq alloc(P)$

(iii) $\forall S \in succ(P)$, $alloc(A) \leq alloc(S)$

then $alloc(P) = alloc(A)$, *for every direct predecessor P of A.*

Definition 4.6 (Revised Reduction Rule) *Given a basic block A, if all the following conditions are true:*

(i) $alloc(A)$ is not a preallocation instruction

(ii) for every direct predecessor P of A, $alloc(P) \geq alloc(A)$

then remove $alloc(A)$.

4.3 MAIA

MAIA is a three-pass algorithm. The control-flow graph is traversed for every transformation to be applied: the downward propagation rule, the upward propagation rule and the reduction rule.

```
G = brute-force solution of the original CFG;

/* Downward propagation phase */
Traverse G top-down in topological order {
        BB = current basic block;
        if (downward propagation rule applies on BB) {
           apply downward propagation rule on BB;
        }
}

/* Upward propagation phase */
Traverse G bottom-up in reverse topological order {
        BB = current basic block;
        if (upward propagation rule applies on BB) {
           apply up propagation rule on BB;
        }
}

/* Downward propagation phase */
Traverse G bottom-up in reverse topological order {
        BB = current basic block;
        if (reduction rule applies on BB) {
           apply reduction rule on BB;
        }
}
```

Figure 4.6: MAIA

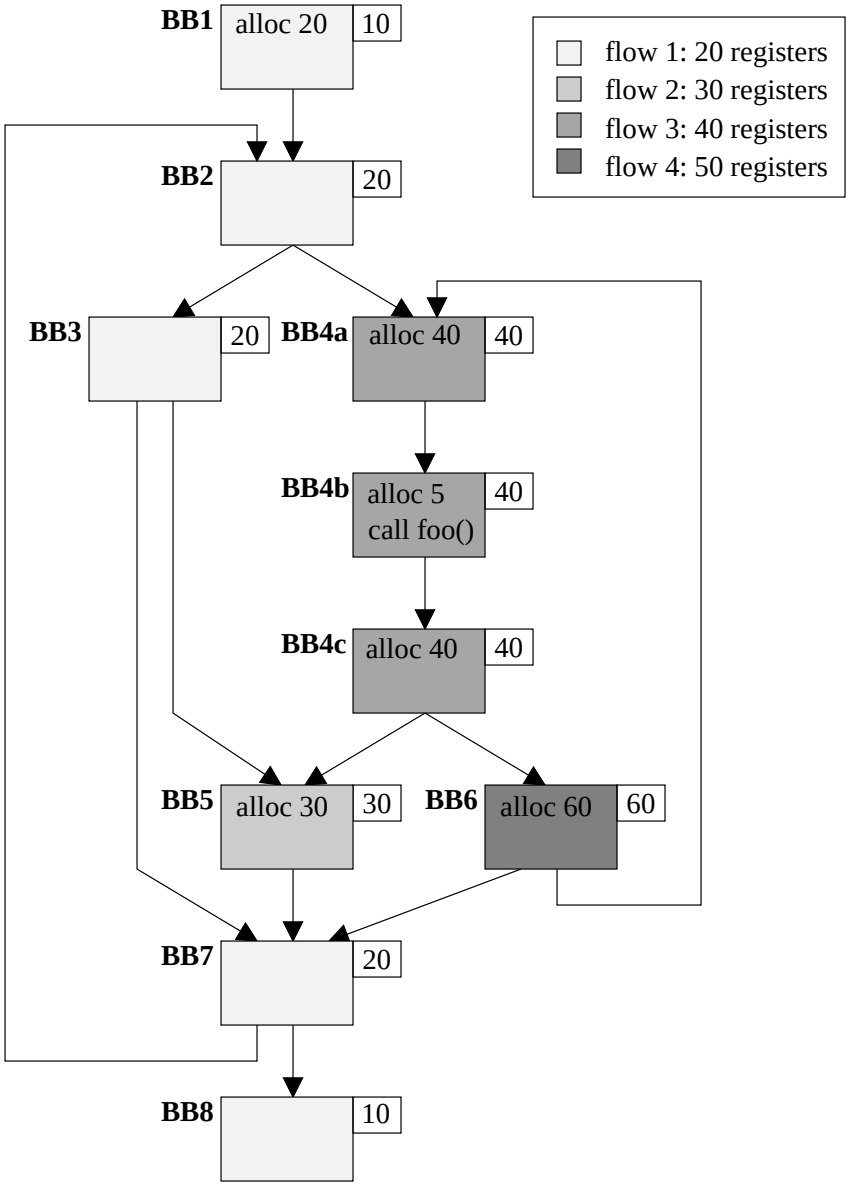


Figure 4.7: The different control paths and the allocation instructions inserted by MAIA. Note that the result is OAIP optimal.

Figure ?? shows a pseudo-code algorithm that uses the three rules. The order in which the rules are applied is not arbitrary. The downward propagation rule needs to be applied first to provide information about higher allocation value earlier in the control-flow graph for the upward propagation rule. The reduction rule uses the results of the upward propagation rule.

Note that "the rule applies" means that the revised definition hypothesis are verified (Definitions ??, ?? and ??).

The three rules are applied on the basic blocks of the control-flow graph, traversed in topological order. The downward propagation rule traverses the control-flow graph from the entry node to the exit node, while the two others rules start from the exit node and end at the entry node.

On Figure ??, we show the different steps of the algorithm that ended up with Figure ??. The algorithm starts on Figure ??(a).

The top-down downward propagation phase propagates only the allocation instructions from $BB7$ to $BB8$. The allocation instruction in $BB1$, $BB2$ and $BB3$ are not propagated because Definition ??(iii) is not satisfied. The allocation instruction in $BB4b$ is a preallocation instruction. Therefore, the allocation instruction of $BB4a$ and $BB4b$ cannot satisfy Definition ??(i) or (ii). The instruction of $BB4c$ cannot be propagated downward to $BB6$ because Definition ??(iii) is not satisfied or to $BB5$ because Definition ??(iv) is not satisfied. Definition ??(iv) cannot be satisfied for $BB5$ and the corresponding allocation instruction is not propagated. Because $BB8$ has no child, there is no downward propagation. The end of the downward propagation phase is shown on Figure ??(b).

With the same reasoning and using Definition ??, we apply the bottom-up upward propagation phase to the control-flow graph. Only one change is visible. The allocation instruction of $BB2$ has been propagated to $BB1$. Some other allocation instructions have been moved, but there is no change to the control flow-graph. For

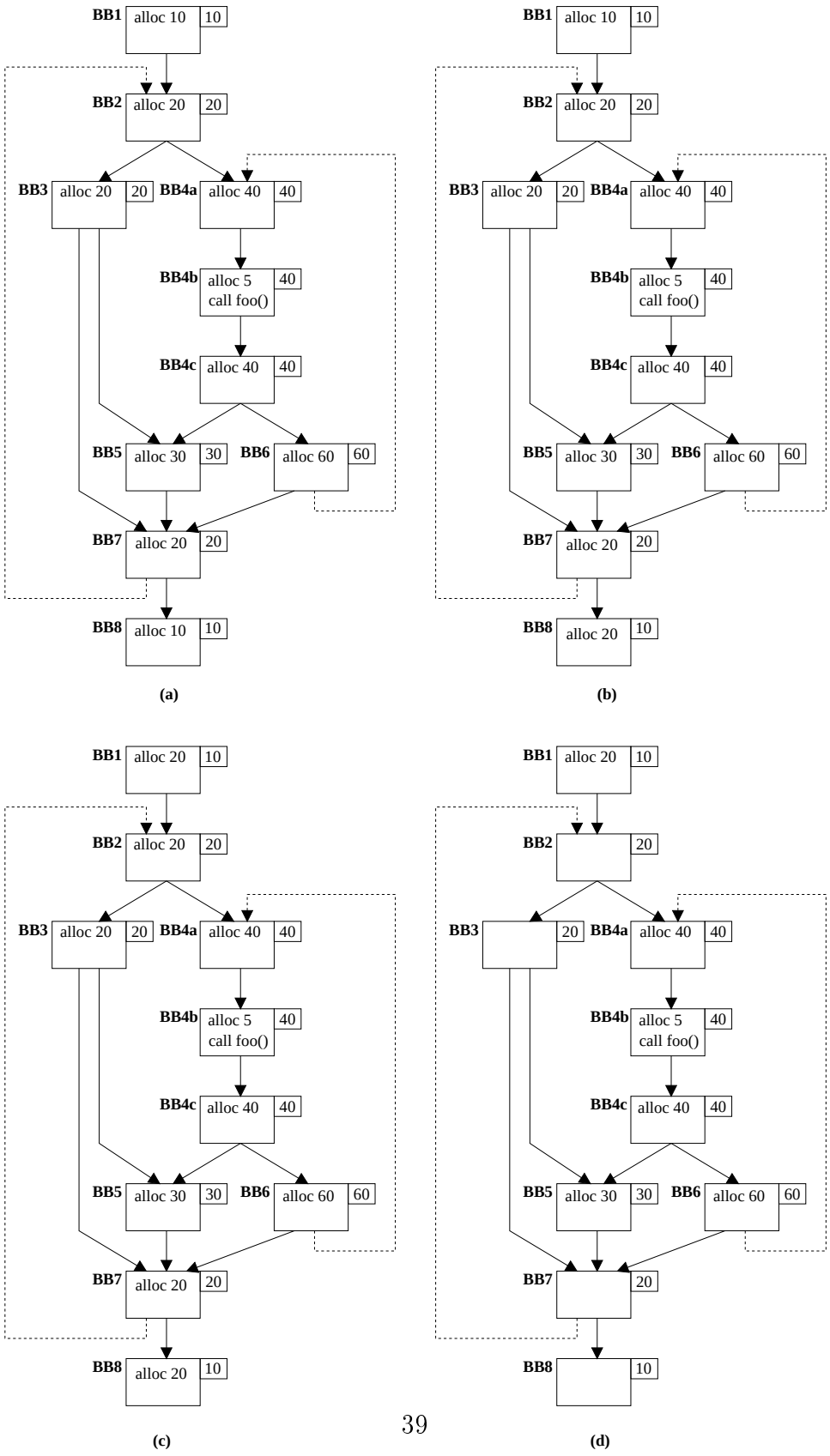


Figure 4.8: Application of MAIA on an example.

instance, the instruction in $BB3$ has been moved to $BB2$, but since the allocation value is the same, nothing happens. The resulting control-flow graph appears on Figure ??(c).

We then apply the bottom-up reduction phase to the control-flow graph. The allocation instruction of a basic block is removed if all the parents have a higher allocation greater or equal to the basic block. The allocation instructions in $BB2$, $BB3$, $BB7$ and $BB8$ are reduced. The others instructions cannot be reduced because at least one of the parent had a lower allocation value. Figure ??(d) shows the final result.

4.4 Time Complexity

In this section we study the time complexity of MAIA. We want to establish the complexity in relation to the number of basic blocks in the control flow graph. Thus we start by showing, in Lemma ?? a bound in the number of edges in the original control flow graph.

Lemma 4.1 *The number of edges in a control-flow graph is proportional to the number of basic blocks.*

Proof. Each edge is an outgoing edge from a basic block. A basic block can only have at most two outgoing edges. Therefore there can at most be twice more edges than basic blocks in a control-flow graph. □

Theorem 4.1 *Assuming the number of parents of a given basic block is bounded by a constant k, independent of the number of basic blocks in the control-flow graph, MAIA is linear in the number of basic blocks in the original control-flow graph.*

Proof. The algorithm has three phases: the downward propagation phase, the upward propagation phase and the reduction phase.

The downward propagation rule visits the children and the parents of the children of the current basic block. Since the number of parents for a given basic block is bounded, the downward propagation rule visits 1 basic block, the children of the basic block (2 maximum) and the parents of the children (k maximum per child). At most, the rule visits $2k + 3$ basic blocks, where k is independent of the number of basic blocks in the control-flow graph. Therefore, the downward propagation rule can be applied in constant time.

The upward propagation rule visits the parents and the other child of every parent if any. Visiting the parents is equivalent to following upward all the incoming edges of a basic block. The edges of the control-flow graph are followed upward only once. Because a basic block has at most two children, visiting the other child of the parent requires that at most one edge be followed downward. Therefore, in the worst case, the upward propagation rule visits each edge in the control-flow graph at most three times. Using Lemma ?? we can conclude that the upward propagation rule can be applied in linear time in the number of basic blocks.

The reduction rule visits only the parents of a basic block. With a reasoning similar to the upward propagation rule, we show that the reduction phase is applied in linear time in the number of basic blocks in the control-flow graph.

We may have introduced new basic blocks because of the function calls and the corresponding preallocation and postallocation instructions. However, since we do not create more than two basic blocks per original basic block, the overall cost remains linear in the original number of basic blocks. Therefore, MAIA is linear in the number of basic blocks in the control-flow graph. □

The assumption that the number of parents of a basic block is bounded by a constant k, independent of the number of basic blocks in the control-flow graph is a fair assumption. Therefore, if the number of immediate predecessors of any basic block is bounded by a constant, then MAIA is linear.

4.5 A Two-Pass Version of the Algorithm

MAIA presented in Section ?? is a three-pass algorithm. However, it can be converted into a two-pass algorithm by merging the upward propagation phase and the reduction phase into one single phase.

We chose to describe a three-pass version of the algorithm in this thesis in order to provide a more straightforward description of the algorithm. To implement the algorithm in a two-pass version, we need to consider the basic blocks in a specific order. When the bottom-up algorithm considers a new level of basic blocks in the control-flow graph, the basic blocks with the lowest allocation value should be considered first. Otherwise reduction opportunities may be missed.

To implement the order in linear time, we use one hash table that is filled with the basic blocks of the current level. the index function of the hash table simply returns the allocation value of the basic block. Therefore, the size of the hash table is equal to R_T, the maximum number of stacked registers that an allocation instruction can allocate. If two basic blocks at the same level share the same allocation value, then they are stored in the linked list in the hash table. When a basic block is inserted in the hash table, a pointer is created. Thus, the basic block always points to its entry in the hash table. With the described implementation of the hash table, the insertion and deletion of a basic block from a hash table are done in constant time, and the overall bottom-up pass remains linear in the number of basic blocks in the control-flow graph.

To conclude, the two-pass algorithm is faster, because one pass of the control-flow graph is avoided. Moreover the memory usage does not dramatically increase because only one fixed-size hash table is used. The number of entries in the hash table is bounded by the number of basic block in the control-flow graph.

4.6 Optimality

To prove the OAIP optimality of MAIA, we first need to prove the RA optimality. Then we show that the minimum number of allocation instructions is used to achieve the RA optimality.

4.6.1 RA Optimality

Lemma 4.2 *Let B be a basic block of a control flow graph G and $C_1, \ldots, C_n$ be control paths from the entry node of G to B. After the phase (1) of MAIA, $alloc(B)$ is equal to the minimum path register requirement of $C_1, \ldots, C_n$.*

Proof. By induction. The algorithm traverses the basic blocks of G in topological order. We prove that the following property is satisfied at each step:

Induction Property: If B is a visited basic block and $C_1, \ldots, C_n$ are all the control paths from the entry node of G to B, then $alloc(B)$ is equal to the minimum path register requirement of $C_1, \ldots, C_n$.

Induction Base: The first node visited by the algorithm is the entry node S of G. The only control path to reach S is $C = S$. From the phase(0) of the algorithm, the initial value for $alloc(S)$ is equal the local register requirement of S, which is the minimum path register requirement for C. Applying the downward propagation rule to node S cannot change the value of $alloc(S)$ because S has no parents, therefore S still satisfies the induction property after it is visited.

Induction Step: Let B be a node in G and let $P_1, \ldots, P_n$ be the set of all immediate predecessors of B. We assume that $P_1, \ldots, P_n$ are already visited and therefore that they satisfy the induction property. We will prove that B satisfies the induction property after it is visited. Let P_m be the predecessor of B with minimum allocation. Upon visiting B the algorithm can only change the allocation of B to $alloc(P_m)$ or keep the allocation value as it is. If the allocation value of B changes to $alloc(P_m)$, then the new value for $alloc(B)$ is equal the register requirement

of the path $\{S, \ldots, P_m, B\}$ which is the path with minimum register requirement — otherwise P_m would not have the minimum allocation amongst the immediate predecessors of B. If the allocation of B does not change when B is visited, then B has a register requirement that is higher than all its predecessors, and thus all paths from S to B have a register requirement equal the local register requirement of B. Thus, in both cases, B satisfies the induction property. □

Lemma 4.3 *Given a control-flow graph G with a start node S and an exit node E. Let B be a basic block of G. After the upward propagation phase of MAIA, $alloc(B)$ is a lower bound to the path register requirement of all the paths from S to E that include B.*

Proof. By induction. The algorithm traverses the basic blocks of G in reverse topological order. We prove that the following property is satisfied at each step:

Induction Property: If B is a visited basic block and $C_1, \ldots, C_n$ are all the control paths from S to E that include B, then $alloc(B)$ is less or equal to the minimum path register requirement of $C_1, \ldots, C_n$.

Induction Base: The first node visited by the algorithm is the exit node E of G. $alloc(E)$ has been computed during the downward propagation phase. Using Lemma **??**, $alloc(E)$ is equal to the minimum path register requirement of all the incoming paths of E. Because E is the exit node of G, the incoming paths of E are all the control paths of G. Thus, before E is visited, $alloc(E)$ is the minimum path register requirement of all the control paths from the entry node of G to E. When E is visited, the upward propagation rule does not modify the allocation value of E. The induction property is satisfied for E.

Induction Step: Let B be a node in G and let $S_1, \ldots, S_n$ be the set of all immediate successors of B. Because the upward propagation phase of MAIA visits the nodes in reverse topological order, $S_1, \ldots, S_n$ are already visited and therefore that they satisfy the induction property. We will prove that B satisfies the induction

property after it is visited. According to Definition ??, $alloc(B) = alloc(S_i)$ if S_i has been propagated to B, and $alloc(B) < alloc(S_i)$ otherwise[1]. Therefore $alloc(B) \leq alloc(S_i)$. Since the allocation values of $S_1, \ldots, S_n$ represents a lower bound to the minimum path register requirement of all the paths going through $S_1, \ldots, S_n$, $alloc(B)$ represents a lower bound to all the paths from E to S that include B. The allocation value of B will not change when the upward propagation rule will be applied to B. □

Lemma 4.4 *Given a control-flow graph G, after the reduction phase of MAIA, for every control path C of G, the maximum number of stacked registers allocated does not exceed the path register requirement of C.*

Proof. After the upward propagation phase, thanks to Lemma ??, the number of stacked registers allocated for any control path C of G does not exceed the path register requirement of C. The reduction phase only removes allocation instruction. Therefore the number of stacked registers allocated for a given control path can only decrease. Lemma ?? is true. □

Lemma 4.5 *Given a control-flow graph G, after MAIA is applied to G, for every basic block B of G, the local register requirement of B is satisfied.*

Proof. After the initialization of MAIA, there is one allocation instruction per basic block and the allocation value is equal to the local register requirement of the basic block. Therefore the local register requirement of every basic block B of G is satisfied before applying the downward propagation phase of MAIA. We now prove that none of the three transformations applied in the successive phases of the algorithm will cause a basic block not to have enough stacked registers allocated.

[1] If Definition ??(iii) is not verified for S_i, then Definition ??(ii) will be verified for S_j with $i \neq j$ and the corresponding allocation instruction will be propagated to B.

Because of Definition ??(ii) and Definition ??(ii), the downward propagation rule and the upward propagation rule only increase the values of the allocation instructions already in place. Therefore, if the local register requirement of a basic block was satisfied by an existing allocation instruction, only more registers can be allocated and the local register requirement of the basic block is still satisfied after the propagation phases of MAIA.

During the reduction phase of the algorithm, the reduction rules only removes allocation instructions. Let A be a basic block whose allocation instruction is removed. According to Definition ??(ii), all the predecessors of A in G have an allocation instruction with a higher allocation value. Therefore, if the allocation instruction of A is removed, the stacked registers allocated by the removed allocation instruction will still be executed. If the local register requirement of A was satisfied before reduction, then the local register requirement of A will still be satisfied after reduction.

Therefore the local register requirement of every basic block B in G is satisfied at the beginning of MAIA and none of the phases of the algorithm will cause a basic block not to have enough stacked registers allocated. We proved Lemma ??. □

Theorem 4.2 *MAIA is RA optimal.*

Proof. Lemma ?? and Lemma ?? prove Definition ??(i). The use of preallocation and postallocation instructions ensures that Definition ??(ii) is satisfied. Therefore MAIA is RA optimal. □

4.6.2 OAIP optimality

Conjecture 4.1 *MAIA is OAIP optimal.*

Proof. Theorem ?? established the RA optimality of MAIA. We conjecture that the number of allocation instructions inserted by MAIA is minimum. The intuition is to

have at most one allocation instruction inserted in each control path in the control-flow graph. If a the path register requirement of a path is already covered by the allocation instruction of another path, then no allocation instruction is required.□

On Figure ??, the control-flow graph has 4 different control paths. The control paths are colored with the intensity proportional to the number of stacked registers needed. When control paths overlap on the same basic block, we color the basic block with the least intensive color. We notice that there is only one allocation instruction per control path, except for the control path 3 that has also one preallocation and one postallocation instructions too. The minimum number of allocation instructions has been inserted. Since the RA optimality is satisfied, the solution is OAIP optimal.

4.6.3 About Natural Loops

In our description of MAIA we only considered forward edges therefore the algorithm only optimizes the static placement of allocation instructions in the control-flow graph. However, if we consider back edges, the repeated execution of the same allocation instruction in a loop can be costly. In such a case only the first call to the allocation instruction is necessary. We consider the dynamic optimization of the allocation instruction placement in the next chapter.

4.7 Conclusion

We have a one-pass bottom-up efficient linear algorithm that fulfills all the requirements of the OAIP problem: the minimum number of registers is allocated for every control path of the control-flow graph while the minimum number of allocation instructions is used. The memory size necessary to execute MAIA is linear: we need one hash table with at most as many entries as there are basic blocks in the control-flow graph.

However MAIA is theoretical, and we did not consider implementation problems and conflicts that may appear with specific architectures. A set of allocation instructions that is OAIP optimal does not necessarily result in the generation of fast code. In the next chapter we will study the effect of optimal placement of instruction allocation on the execution time of the code generated.

Chapter 5

OPTIMIZATIONS AND FINE-TUNING

In the previous chapter we described an efficient linear algorithm. We assumed that the allocation instruction was free: there was no dependence or conflict of any type, and the execution time of the instruction was negligible. Also optimal was meant statically, not dynamically.

In this chapter, we will rise some issues when we try to implement the algorithm in a real compiler in a specific architecture (IA-64 architecture for instance). We will consider the cost of the allocation instruction itself, the redundant call to allocation instructions in loops and the use of frequency of execution information.

5.1 Cost of the Allocation Instruction and Tuning

Until now, the allocation instruction was assumed to have no cost. However, the instructions in the source code using stacked registers are very likely to depend on the allocation instructions: the stacked registers cannot be used before the allocation instruction terminates. The IA-64 architecture, based on Instruction-Level Parallelism (ILP), may see scheduling opportunities reduced because all the other instructions are waiting for the allocation instruction. The IA-64 architecture also requires that the allocation instruction be the first instruction in an instruction group ([?]).

Moreover the cost of the instruction itself should not be neglected. The register stack engine is expected to spill and restore registers when needed in a transparent manner. But depending on whether spills and restores are necessary

or not and how many registers need to be moved to/from memory, the cost of the allocation instruction may vary.

Overall we have to take into account the cost of the allocation instruction. Depending on the architecture, the implementation and the assembly code itself, the cost differs. Too many allocation instructions, even if optimal, may slow down the execution of the program.

We have to assign some priority to the allocation instructions. Some are less useful than others. Is it worth inserting an allocation instruction to save one or two registers?

What we propose here is a simple heuristic to reduce the number of allocation instructions in the code if the cost of the allocation instruction appears to be too high. We insert an allocation instruction if we save enough registers. We introduce a threshold value: R_m. R_m is the minimum number of stacked registers that must be saved when we insert an allocation instruction. Therefore the cost of the allocation instruction is inversely proportional to the number of stacked registers saved by the instruction.

The only change that needs to be done to the main algorithm concerns the upward propagation rule. We want to force the propagation if the number of registers saved is too low (less than R_m).

Definition 5.1 (Heuristic Upward Propagation Rule) *Given a basic block A, if, for every direct predecessor P of A, all the following conditions are true:*

(i) $alloc(A)$ *is not a postallocation instruction*

(ii) $alloc(A) \geq alloc(P)$

(iii) $\forall S \in succ(P)$, $alloc(A) \leq alloc(S) + R_m$

then $alloc(P) = alloc(A)$, *for every direct predecessor P of A.*

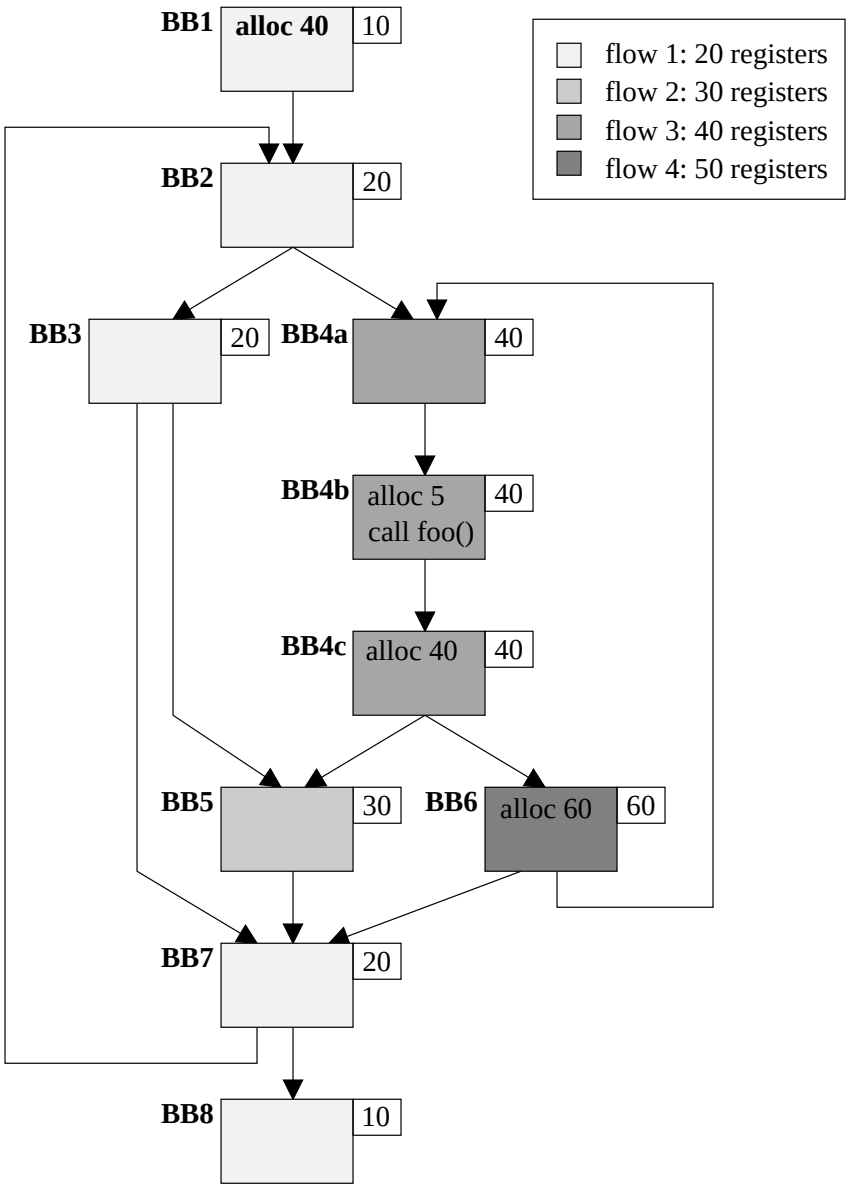


Figure 5.1: A new solution when the register-saving heuristic is used.

If we apply the new propagation rule instead with $R_m = 10$, the example of Figure ?? changes. The allocation instruction in $BB5$ is propagated all the way to $BB2$, where the allocation instruction of $BB4a$ is also propagated. Eventually, we decrease the number of allocation instructions by 33%. But *control path* 1 allocates 20 registers more than the minimum required for that control path, and *control path* 2 allocates 10 extra registers.

As a consequence, the solution cannot be optimal anymore. We may allocate more registers than needed. Experiments must be done to correctly choose the value R_m and make a compromise between stacked register allocation and execution speed.

5.2 Redundant Calls to Allocation Instructions in Loops

We described the algorithm as statically optimal. Which does not mean that dynamically the number of calls to allocation instructions is minimized. If a run-time flow of execution only follows forward edges in the control-flow graph, the number of calls is optimal. But if one back edge of the control-flow graph is used, we may call the same allocation instructions several times unnecessarily.

On Figure ??, if no back-edge is used, the four flows of execution are optimal. At run-time, we are very likely to iterate more than once the two loops by using the back edges of the control-flow graph. If we execute $BB3$ and $BB7$ in a loop using the left back edge, no harm is done. There is no allocation instruction on the way. On the other hand, if we execute $BB3$, $BB5$, and $BB7$ in a loop, then the allocation instruction in $BB5$ is going to be called more than once, even though the stacked registers are already allocated. A worse case exists for the loop $BB4a|b|c$ and $BB6$: the allocation instruction in $BB4a$ shrinks the register stack frame. We know that shrinking allocation instructions are useless.

We propose two solutions to reduce redundant calls to allocation instructions. The first one only avoids one special case without breaking the static optimality.

The second avoids all the redundant calls, but the solution is not optimal anymore: too many stacked registers might be allocated.

5.2.1 Avoiding the First Allocation Instruction of a Loop

There is one special case: the first instruction of a loop is an allocation instruction. Although only the first allocation is necessary, the allocation instruction is executed at every iteration of the loop. Since the register allocation increases along every control path, we know that the other calls are redundant. We avoid redundant calls by creating a fake basic block for the allocation instruction. The back edge of the loop points to the successor of the basic block. Thus, at the entry of the loop, the allocation instruction is executed once. Then the subsequent iterations skip the new basic block, and the allocation instruction is not called. In other words, the allocation instruction is moved to the header of the loop.

On Figure ??, $BB4a$ is the first basic block of a loop and contains one allocation instruction. We introduce another basic block ($BB4*$) to avoid the allocation instruction to be executed more than once. The solution appears on Figure ??.

The move of the first allocation instruction to the loop header does not modify the number of allocation instructions and the register allocation optimality is maintained.

5.2.2 Avoiding Redundant Calls to Allocation Instructions

We want to avoid redundant calls to allocation instructions within a loop. The proposed algorithm lists the basic blocks of the outer loop and consider the highest allocation instruction value of these basic blocks. Then we apply the solution than the previous section: we create a fake basic block with one allocation instruction that covers the needs of all the basic blocks in the loop.

We consider the outer loop only, because inner loops are part of the outer loop and their needs are therefore covered by the outer loop.

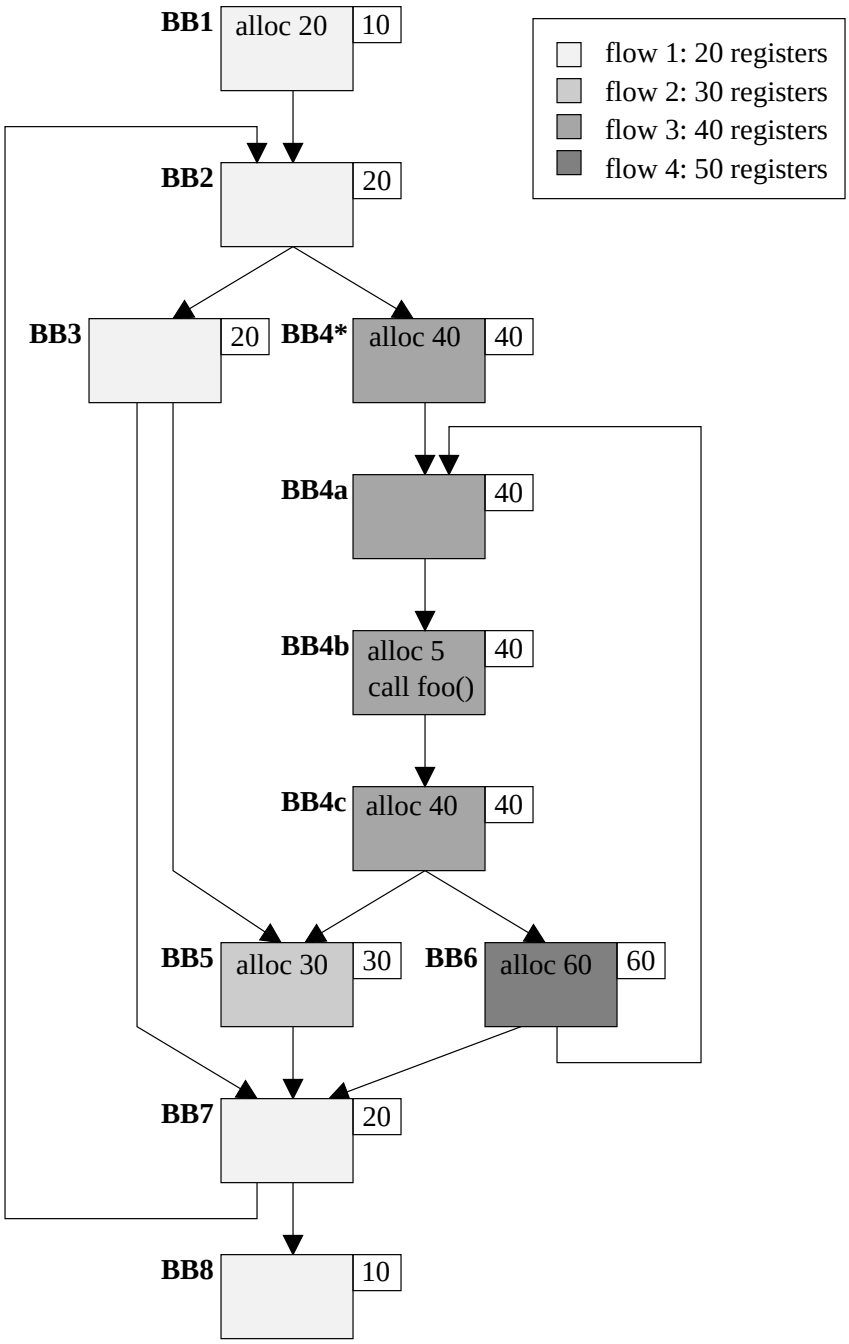


Figure 5.2: The first allocation instruction of a loop can be executed only once.

However, we need to apply the allocation instruction insertion algorithm again to consider the changes made. We want to avoid a second call to the allocation instruction insertion algorithm.

Another way to solve redundant calls to allocation instructions in loops is to set the stacked register requirement of all the basic blocks in the outer loops to the highest value needed by the basic blocks of the loops, even before applying the allocation instruction insertion algorithm.

When applied to our example, the stacked register requirement of the outer loop ($BB2$ to $BB7$) is increased to 60. All the basic blocks of the loops have the same register requirement. When MAIA is applied, we end up with only three allocation instructions for the entire control-flow graph (Figure ??). All the allocation instructions (except preallocation and postallocation allocation instructions) are called only once. Note that, by coincidence, the solution is the call-shrinkage one (Figure ??).

Several calls to preallocation and postallocation instructions are not considered as redundant because they are necessary to avoid the frozen allocation of stacked registers (Chapter ??).

The number of stacked registers allocated in the loops may be too high for specific flows of execution inside the loop, but we expect that the number of times the loop is executed will amortize the waste of stacked registers.

5.3 Frequency of Execution Information

We may expect from the compiler to provide static or dynamic information about the the frequency of execution of each basic block in the control-flow graph. The information can be used to avoid useless upward propagations: if a very expensive basic block is almost never executed, then we would like to see allocation instruction of the basic block to remain in place.

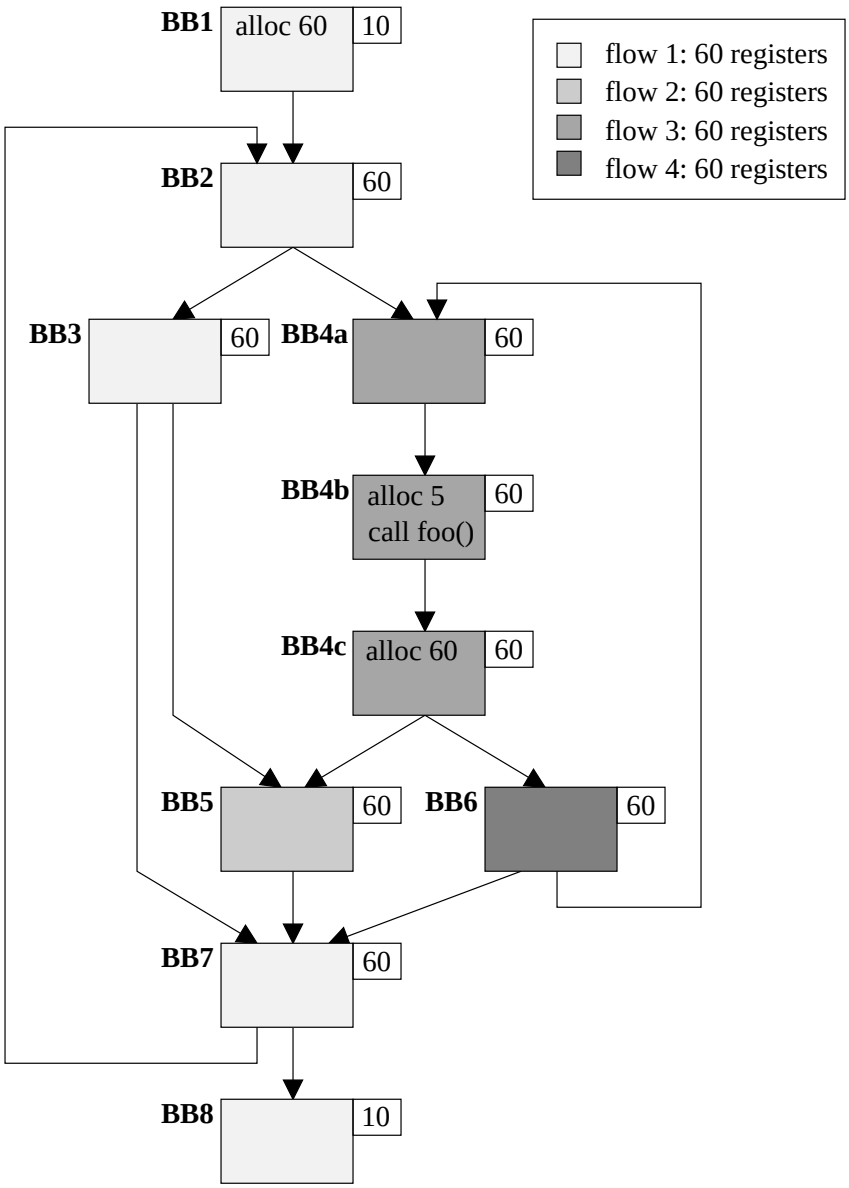


Figure 5.3: Control-flow graph without redundant calls to allocation instructions.

We assume we are given a weighted control-flow graph, i.e. where each edge has a specific weight. The higher the weight, the more likely the edge will be taken at run-time.

The idea is to form regions in the control-flow graph. There would be a main region (or hot region), that would include the basic blocks the most often executed at run-time. Then, there would be the other regions less often executed, and therefore less important. We would apply the algorithm on the entire control-flow graph but differentiate between the regions. Thus the allocation instructions from less important regions would not interfere with the other basic blocks of the control-flow graph: we only consider the important allocation instructions.

We might use the profile-sensitive region formation algorithm originally used in the IMPACT compiler[?] and described in [?][?]. The algorithm has four steps:

(i) **Choose a seed block:** We choose the basic block with the highest frequency of execution.

(ii) **Propagation to the successors:** From the seed block, we propagate the region to the successors. We only stop if there is a function call, if the frequency of execution of an edge is too low (less than 50% of the immediate predecessor and that of the seed block) or if the region becomes too large.

(iii) **Propagation to the predecessors:** From the same seed block, we propagate the region to the predecessors using the same method.

(iv) **Propagation to the successors of the blocks in the region:** We try to include other blocks by considering all the blocks in the current region as seed blocks. We extend the region only by following the successors. Thus we obtain a multi-flow region.

Note that we may have more than one seed block. The regions built from the seed blocks may merge into one single region or not.

Once the regions are formed, we apply our algorithm to the control-flow graph. The allocation instructions from the main region can be propagated into less important regions, whereas allocation instructions from the other regions cannot cross the boundaries between regions. Which brings us to a new propagation rule:

Definition 5.2 (Profile-Sensitive Propagation Rule) *Given a basic block A, if, for every direct predecessor P of A, all the following conditions are true:*

(i) $alloc(A)$ *is not a postallocation instruction*

(ii) $alloc(A) \geq alloc(P)$

(iii) $\forall S \in succ(P)$, $alloc(A) \leq alloc(S)$

(iv) A *is in the main region or* P *is not in the main region*

then $alloc(P) = alloc(A)$*, for every direct predecessor P of A.*

When applied to our example where weight has been added to the edges of the control-flow graph (Figure **??**), we obtain one main region composed of four basic blocks: $BB2$, $BB3$, $BB5$ and $BB7$. $BB2$ and $BB7$ were the seed blocks because of their highest frequency of information (1000). $BB3$ was included by the second step of the region formation algorithm, $BB5$ by the fourth step.

The allocation instructions are propagated as usual. The only difference is for the *alloc* 40 from $BB4a$, that cannot cross the boundary from its own region to the main region. The allocation instruction cannot be propagated anymore. We end up with a solution where 30 registers are allocated by default. More registers are allocated only if the very unlikely to be executed inner loop is executed.

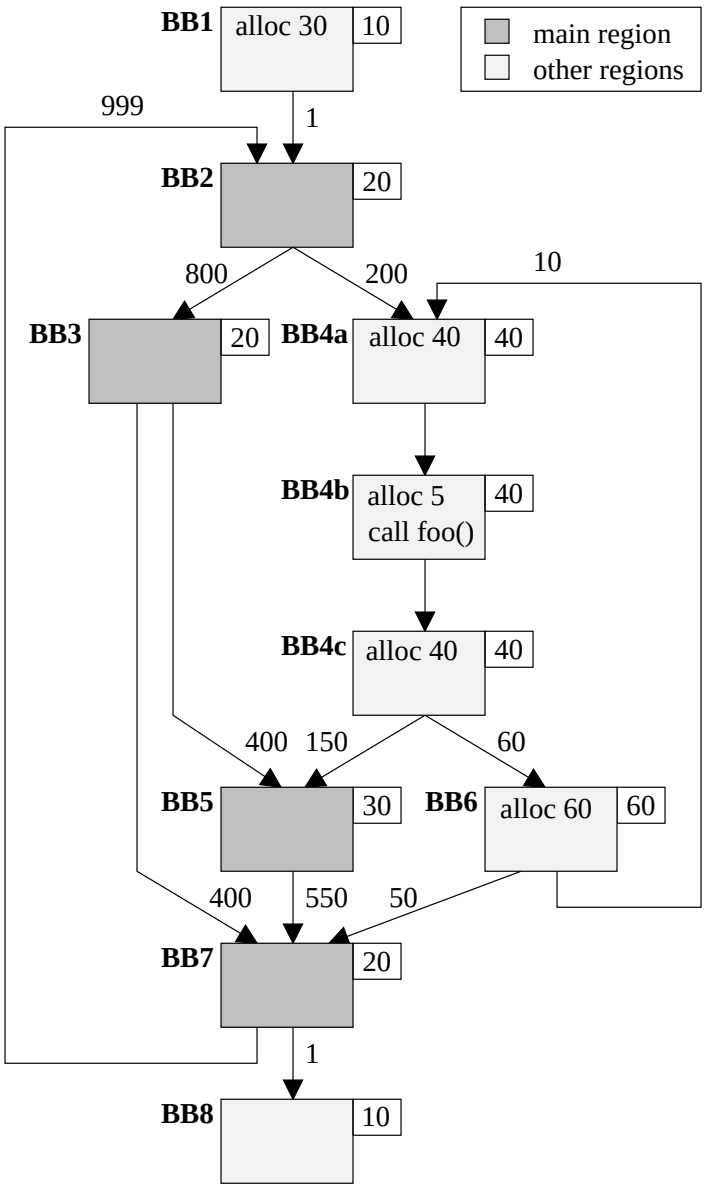


Figure 5.4: Solution when considering frequency of execution.

Chapter 6

CONSIDERATIONS FOR THE IA-64 ARCHITECTURE

So far, we have studied the management of a theoretical register stack in a fictitious architecture and solved the general related problems. We are now going to consider the IA-64 architecture, developed by Intel. The architecture provides the user with a register stack of 96 registers and a slightly larger interface.

In the next sections, we will specifically present the register stack in the IA-64 architecture and two major obstacles to MAIA: the rotating registers and predicated instructions.

6.1 The Register Stack

The IA-64 register stack is very similar to our theoretical model with $R_T = 96$. The restore and spill operations are transparently managed by the hardware-implemented Register Stack Engine (RTE). When the restore and spill operations become necessary, the RTE tries to take advantage of unused memory bandwidth to access the register *backing store* and hopefully does not stop the execution of the program. The outgoing parameters are passed by overlapping the stack frames of the caller and the callee functions.

The main differences with our theoretical model comes with the interface: there are 3 more instructions (`flushrs`, loadrs, cover) and the syntax of the allocation instruction is more complete. Also the allocation must obey more constraining rules.

Instruction	Description
`alloc`	allocate registers in the register stack frame
`flushrs`	flush the register stack to the backing store
`loadrs`	load the register stack from the backing store
`cover`	cover current stack frame

Figure 6.1: IA-64 register stack interface.

6.1.1 The *alloc* Instruction

Basically everything that we assumed to be managed by the hardware is explicitly controlled by the *alloc* instruction: the register stack frame pointer and the passing parameters. Moreover, the instruction also explicitly allocates *rotating registers* specific to the IA-64 architecture (cf. ??).

$$\text{alloc } gr_1 = ar.pfs, \text{ i, l, o, r}$$

Figure 6.2: Syntax of the alloc instruction.

When the alloc instruction is called, the value of *Previous Function State* register $ar.pfs$ is copied to gr_1. gr_1 is a static register caller/callee saved when another function is called. Thus the previous register stack frame parameters are saved and previous register stack frames can be restored.

The other arguments of the alloc instruction concerns the size of the current register stack frame and the partitioning of the registers in the frame.

Arguments	Description
i	number of incoming parameters
l	number of local registers
o	number of outgoing parameters
r	number of rotating registers

Figure 6.3: *Alloc* instruction argument description.

The incoming registers are considered as local. Therefore, the size of the register stack frame (sof) can be deduced from the previous values: $sof = l + o$.

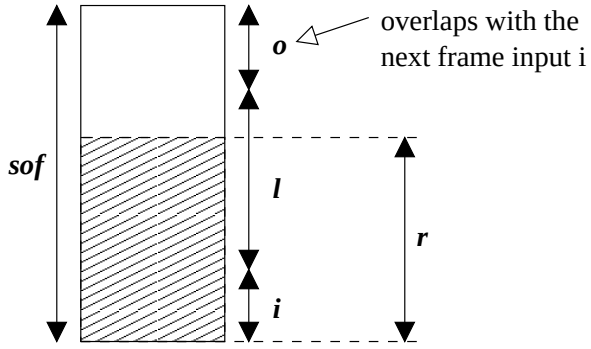


Figure 6.4: Partition of the IA-64 register stack frame.

The number of rotating registers must a multiple of 8 and less than the size of the register stack frame.

The alloc instruction must obey specific rules: the instruction has to be the first instruction in an instruction group and cannot be predicated. The effects of the alloc instruction are seen by the other instructions in the same group and thereafter.

6.1.2 The *flushrs* and *loadrs* Instructions

The *flushrs* instruction is used to explicitly save register stack frames from previous functions in memory and free the space for the future allocations.

The *loadrs* instruction loads a number of bytes from the memory to the register stack. The instruction is used to invalidate registers in the register stack.

As the *alloc* instruction, the *flushrs* and *loadrs* instructions have to be the first instruction in an instruction group and cannot be predicated.

6.1.3 The *cover* Instruction

The *cover* instruction allocates a new stack frame of size zero. All the stacked registers, even outgoing registers, are not available anymore.

The *cover* instruction must be the last instruction in an instruction group and cannot be predicated.

6.2 The Rotating Register Allocation

In the IA-64 architecture, the decrease in the execution speed of the loops induced by the lack of out-of-order execution was compensated by the use of rotating registers. At each iteration of loops, these special registers are shifted in a register-renaming process. For instance, a value that would have appeared in $r35$ in the first iteration, would appear in $r36$ in the next, and thus reducing dependences between iterations within the same loop.

The rotating registers are allocated in the register stack with the *alloc* instruction. The number of rotating registers must be a multiple of 8 and less than 96 (R_T). The rotating register base register must be cleared before allocating any rotating register.

If there are at least two loops with different rotating register requirements in the same routine, we need to insert extra *alloc* instructions to reallocate the rotating registers. The newly inserted instructions do not change anything to the size of the register stack and can be inserted independently from the allocation algorithm. However we may want to merge the *alloc* instructions if possible, or even take into consideration the new type of *alloc* instruction (rotating registers) right from the beginning.

6.2.1 Insertion After The Allocation Algorithm

The first optimization is a simple patch to the solution given by MAIA. We simply insert the *alloc* instructions we need to allocate the rotating registers and then try to move the inserted instructions up to an existing *alloc* instruction. Note that the moving operation is not a propagation as defined in the previous chapters.

However, because two *alloc* instructions could be moved up to the same existing *alloc* instruction and not be aware of it, the move would be limited to a linear chain of basic blocks. As soon as a split node is encountered, the move operation must stop and the *alloc* instruction cannot be merged with an existing one. Therefore, the efficiency of the simple patch is limited.

Also there is no easy way to choose a value for the other arguments of the *alloc* instruction. The number of local registers we need was determined by the allocation instruction insertion algorithm and forgotten since. We could look for all the existing *alloc* instructions above in the control-flow graph, but the operation would be too expensive. Therefore the patch to MAIA is too time-consuming.

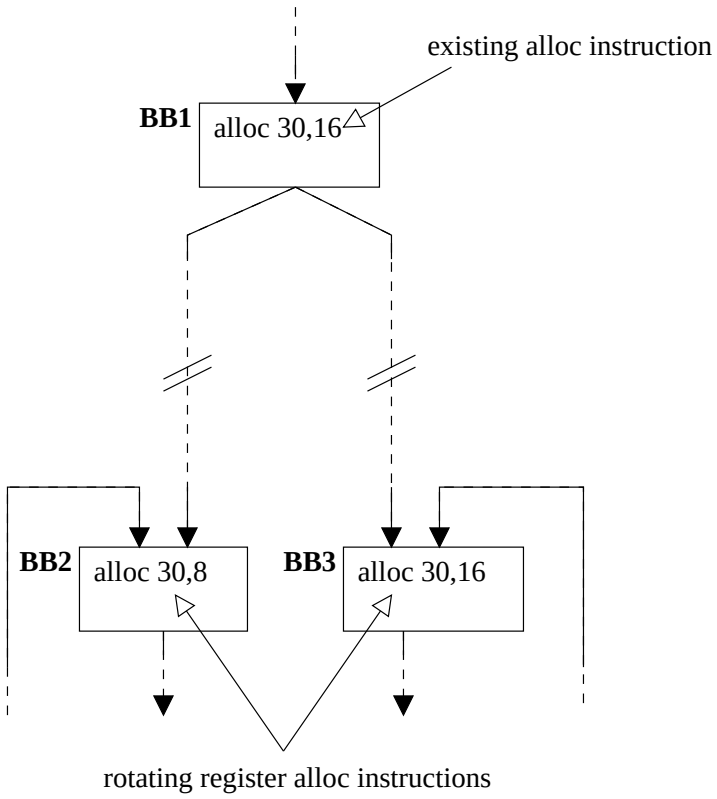


Figure 6.5: Example of inefficiency of a post-pass patch for the rotating register allocation.

On Figure ??, the *alloc* instruction of $BB1$ covers the stacked register requirements of the two control paths. The instruction has been inserted by MAIA. The second number appearing with the allocation instructions is the number of rotating registers allocated. The default value is the maximum used by all the loops of the routine. There is no other *alloc* instructions between $BB2/BB3$ and $BB1$ (the intermediary basic blocks are not shown).

Then we insert two rotating register allocation instructions in $BB2$ and $BB3$. The number of local registers for the two instructions is computed by looking all the way up to the existing *alloc* instruction (30). We assumed the number of rotating registers needed for each loop is known (8 for $BB2$ and 16 for $BB3$). In the example, we cannot propagate the rotating register *alloc* instruction into the split node without investigating the entire other branch of the split node. Therefore there is no way to cheaply merge the inserted *alloc* instruction with an existing in the specific example with a post-pass patch to MAIA.

6.2.2 A Rotating Register-Aware Allocation Algorithm

The best way to handle the rotating registers is to modify the existing algorithm to take into account the rotating register allocation instruction type and the rotating register value.

We start with one allocation instruction per basic block as in the original MAIA. But we now have to give a default rotating register value: all the allocation instructions start with a rotating register default value of 0, except the instructions in basic blocks included in a loop using rotating registers. A way to compute the list of basic blocks in a natural loop can be found in [?].

Since we use the algorithm of Section ??, all the allocation instructions in the same loop at the beginning of the algorithm are the same: maximum number of local registers needed in the loop and maximum number of rotating registers needed in the loop.

Then we apply the algorithm with some modifications to the propagation and reduction rules. Besides the register stack frame size, the rules now consider the rotating register values. The allocation instructions can be reduced or propagated if the conditions about the allocation values AND the rotating register values are respected. The rules are therefore more strict.

Notation: $rot(A)$ is the number of rotating registers allocated by the allocation instruction of A. If A has no allocation instruction or if there is no rotating register allocated, the rotating register allocation value is 0. If $alloc(A) = 0$ and $rot(A) = 0$, then there is no allocation instruction in A.

Definition 6.1 (Rotation Register-Aware Upward Propagation Rule) *Given a basic block A, if, for every direct predecessor P of A, all the following conditions are true:*

(i) $alloc(A)$ *is not a postallocation instruction*

(ii) $alloc(A) \geq alloc(P)$

(iii) $rot(A) \geq rot(P)$

(iv) $\forall S \in succ(P)$, $alloc(A) \leq alloc(S)$

then $alloc(P) = alloc(A)$, *for every direct predecessor P of A.*

Definition 6.2 (Rotation Register-Aware Reduction Rule) *Given a basic block A, if all the following conditions are true:*

(i) $alloc(A)$ *is not a preallocation instruction*

(ii) for every direct predecessor P of A, $alloc(P) \geq alloc(A)$ *and* $rot(P) \geq rot(S)$

then remove $alloc(A)$.

The upward propagation rule does not change concerning the register stack frame value. However the rotating register value adds a new constraint: the allocation instruction can be propagated as normally if the rotating register allocation value is at least as high as the preceding allocation instruction. Unlike the register stack frame value, the rotating register value does not only increase along every control path. The upward propagation rule does not take into account the rotating register value of the other children of the parent like for the register stack frame value. Therefore we allow the number of rotating registers to shrink. Although shrinking was considered as not efficient for the register stack frame value, reducing the number of rotating registers has no impact on the execution speed of the program. There is no spill/restore due to a change in the number of rotating registers: the stacked registers are already allocated using the register stack frame value. Note that we are sure not to allocate more rotating registers than the register stack frame size, because both $alloc(A)$ and $rot(A)$ are propagated at the same time.

The reduction rule is modified with the same idea in mind. If P covers the needs of S in size of the frame and number of rotating registers, then the allocation instruction of S is useless and can be reduced.

The solution of the rotating register aware algorithm used on our example is shown on Figure ??. The outer loop is using 8 rotating registers, while the inner loop does not use any. Only one loop at the time can make use of the rotating registers. The solution is close to the one given in Figure ??. The rotating register value parameter has been added to the alloc instructions. One may notice that the preallocation instruction value has increased from 5 to 8. Since the number of rotating register value has to be a multiple of 8 and must fit in the register stack frame, we had to adjust the register stack frame value. The updates of the value of the preallocation instructions are automatically done at the beginning of the algorithm when all the allocation instructions are inserted.

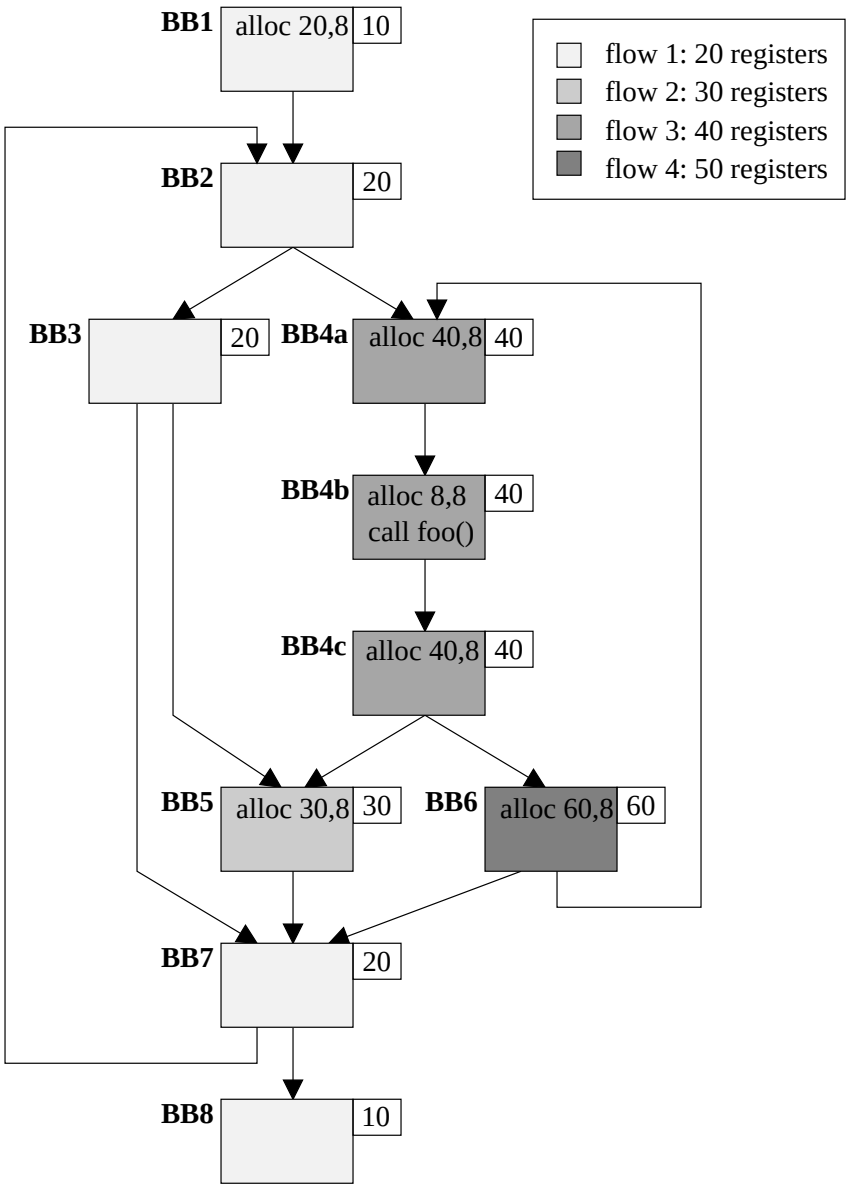


Figure 6.6: Rotating register aware solution.

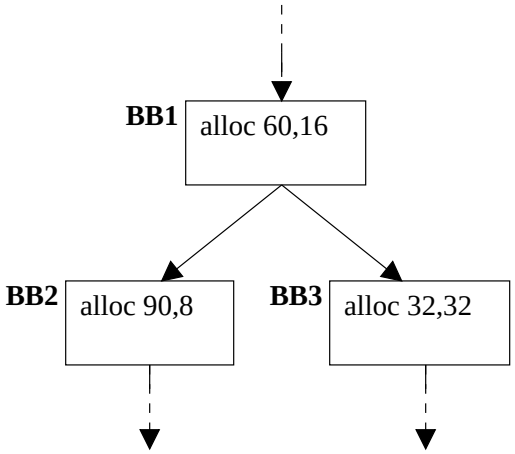


Figure 6.7: Consequence of a partial order of the basic block: no allocation can be propagated.

However, from the the point of view of the upward propagation rule, we now have a partial order of the basic blocks. Before, from the two children of a split node, one allocation instruction was going to be propagated and reduced. Now, one instruction may have the largest register stack frame while the second has the higher number of rotating registers. Therefore none of the two allocation instructions are propagated or reduced, resulting in a higher number of allocation instructions. Figure ?? shows a specific configuration where no allocation instruction among the children are propagated or reduced: $BB2$ does not allocate enough rotating registers, while $BB3$'s register stack frame value is too small.

As a conclusion, the rotating register may bring a lot of constraints. But the rotating register aware algorithm is very likely to be used in correlation with the optimization described in Section ??. Thus, the impact of the rotating registers on the efficiency of MAIA is limited.

6.3 Predicated Control Path of Instructions

Unfortunately the *alloc* instruction cannot be predicated. As a consequence, MAIA must be applied after predication is done. Otherwise predication, one of the

hot features of the IA-64 architecture, would be limited. The limitation of the *alloc* instructions is preferable to the limitation of predication.

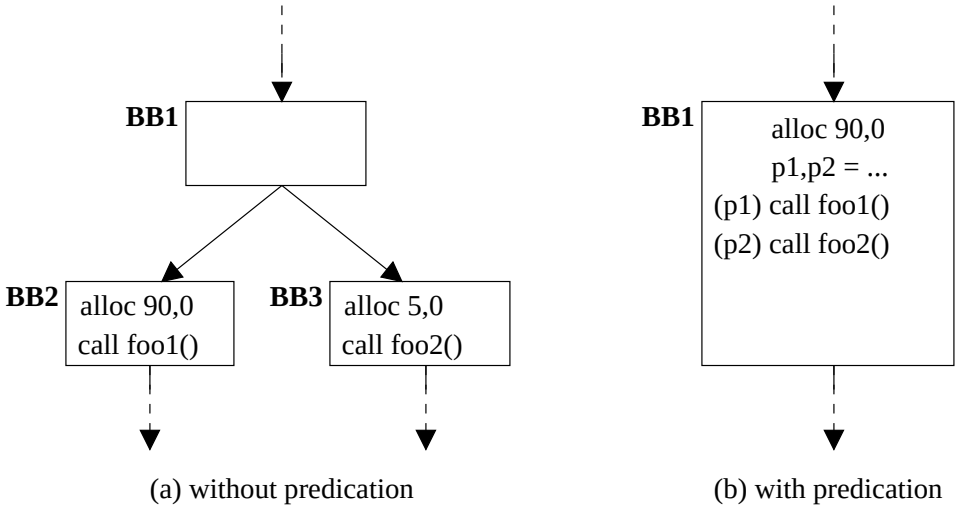


Figure 6.8: Predication limits the efficiency of the algorithm.

To take into account the predicated instructions, we look at the control-flow graph of the routines and consider all the instructions as always executed (the predicate values are considered as true). Therefore we lose the fine-grain control we had over the insertion of the *alloc* instruction: if two parallel control paths are predicated and with very different stacked register requirements, we have to consider the worst case and insert an *alloc* instruction to cover both needs instead of one fine-tuned *alloc* instruction per control path. Predication is the main limit to the allocation instruction insertion algorithm.

On Figure ??, $foo2$ can be preceded by a fine-tuned preallocation if the call is not predicated. However, with predication, we must allocate for the two control paths and 85 stacked registers are wasted if $foo2$ is called.

Unless the *alloc* instruction can be predicated, there is so far no easy solution to solve the predication problem. We may want to consider huge differences in stacked register requirements between two paths with function calls before applying

an if-conversion for instance. However we are not able to list all the cases. A study of the impact of each optimization (predication or *alloc* instruction) would be necessary beforehand.

Chapter 7

FUTURE WORK

So far, we introduced MAIA, an efficient allocation insertion algorithm. We presented different architecture independent optimizations and considered real-life constraints with the IA-64 architecture. Besides the conflict with predicated instructions, there are still opened questions. The efficiency of the algorithm could be enhanced if some other optimizations were more aware of the *alloc* instructions. Also a deeper study of the *alloc* instruction would help to fine-tune the presented algorithms.

7.1 Interactions With Other Optimizations

7.1.1 Copy Propagation

The copy propagation, a classic compiler optimization, consists of, given an assignment $x \leftarrow y$, replacing later uses of x with uses of y, as long as intervening instructions have not changed the value of either x or y ([**?**]).

Copy propagation might reduce the efficiency of the allocation insertion algorithm. Let us consider the IA-64 architecture. Assuming there is a function call, the parameters are passed using the last registers of the register stack frame. If a preallocation instruction is inserted, the register stack frame is shrunk and stacked registers are saved. However if copy propagation is used, the definition of the parameters may appear far above in the control-flow graph, forcing the *alloc* instruction to be inserted before (and reducing shrinking opportunities) or the insertion of copy instructions (cancelling the copy propagation algorithm job).

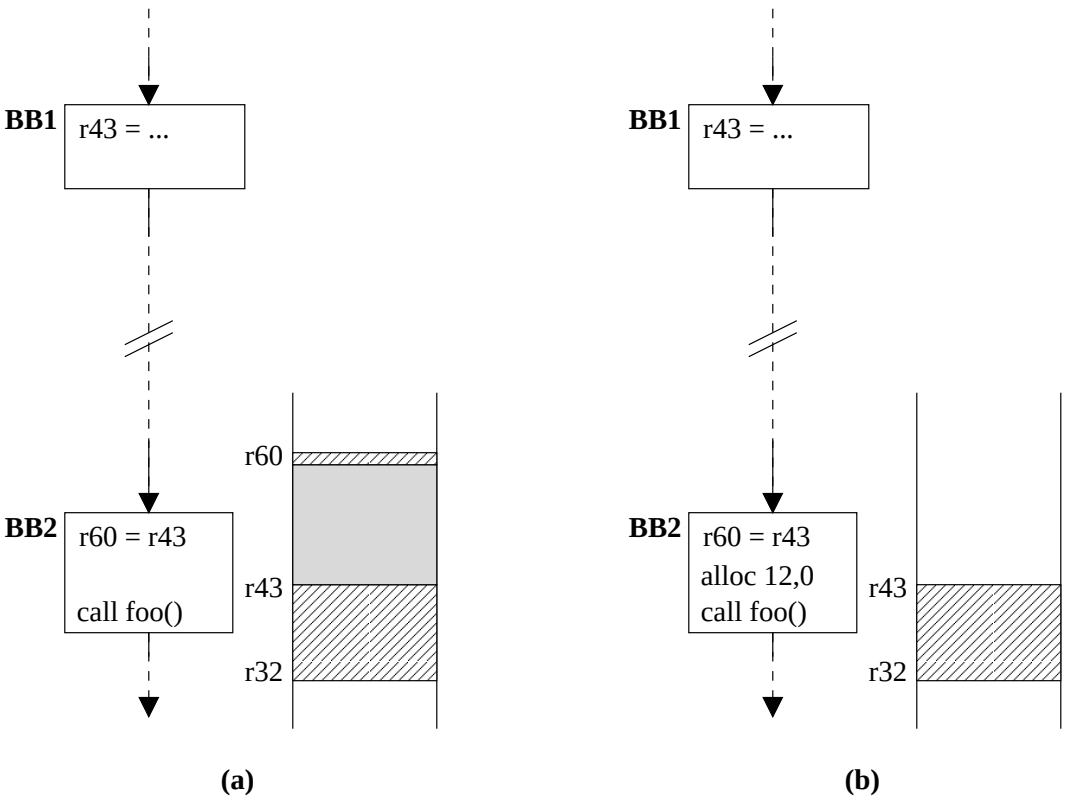


Figure 7.1: Allocation instruction insertion with no copy propagation.

On the example Figure ??-a, *foo*() is called with only one parameter. The value is passed using the register stack and stored in the last register allocated in the current register stack frame: $r60$. However all the registers between $r43$ and $r60$ are not used when the function is called, resulting in a waste of 16 stacked registers.

When the allocation insertion algorithm is used (Figure ??-b), we can reduce the waste from 16 to 0. We shrink the register stack frame to the point where the last register is $r43$ and no copy instruction is even needed. The shrink is possible because the allocated stacked registers above $r43$ were not used.

Now, when copy propagation is used, $r60$ happens to be defined far above in the control flow graph in place of $r43$. The allocation instruction cannot shrink anything when the call occurs and 16 stacked registers are wasted again (Figure ??-a).

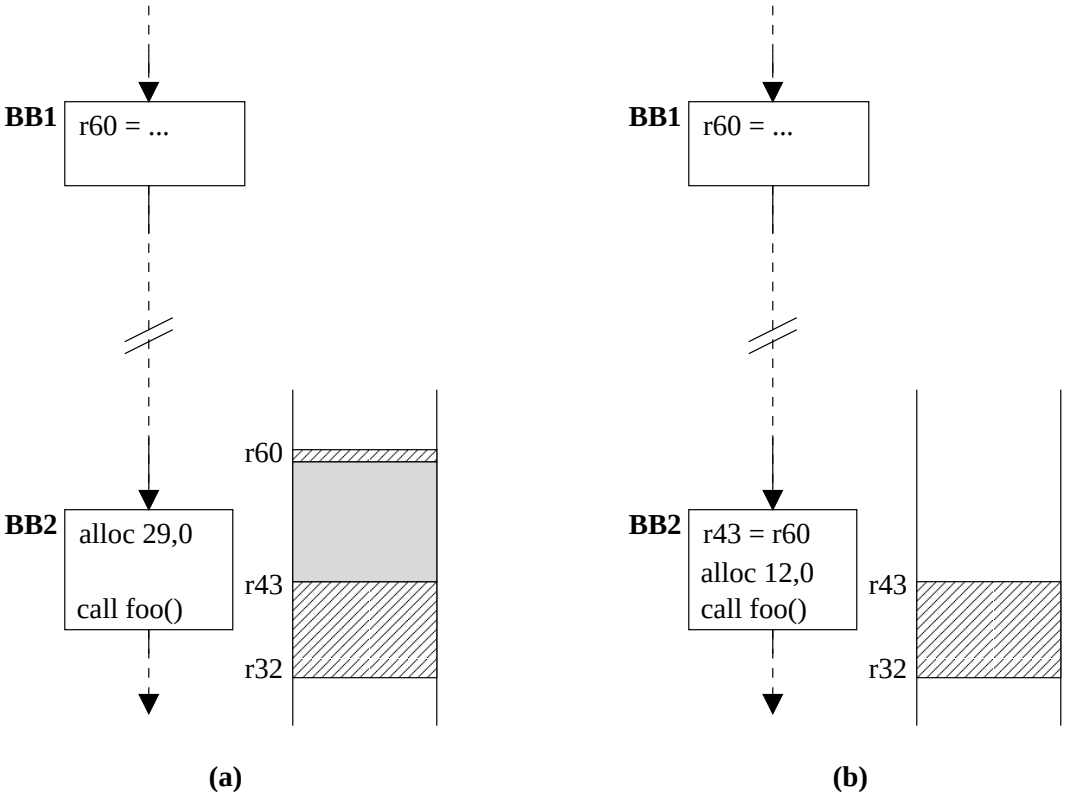


Figure 7.2: Allocation instruction insertion with copy propagation.

Assuming we have an algorithm to figure out when we can easily a large number of registers, we could copy the value of $r60$ into $r43$ and shrink the register stack frame. However we inserted an extra instruction and simply undid the copy propagation job (Figure ??-b).

A future work would be to study the impact of copy propagation on the allocation insertion algorithm. Maybe the copy propagation has more to offer than a smart allocation instruction insertion and we should not bother about it. To forbid copy propagation of outgoing parameters with some conditions, like the number of stacked registers saved/wasted, might also be interesting to study..

7.1.2 Register Allocation

The register assignment algorithm and the allocation instruction insertion algorithm are obviously deeply linked. While the first algorithm assigns register number to virtual registers, the second makes sure the stacked registers are effectively allocated on the stack when needed. Therefore we might expect a good cooperation between the two algorithms.

The allocation instruction insertion algorithm does not make any change to the register assignment. The algorithm assumes the assignment is fixed. Consequently, the register assignment algorithm is responsible for the assignments that may not looked interesting from an allocation instruction insertion point of view. The allocation instruction insertion algorithm cannot do anything against a large number of consecutive stacked registers not used and surrounded by live registers. The register assignment algorithm is responsible of making the register stack frame as dense as possible.

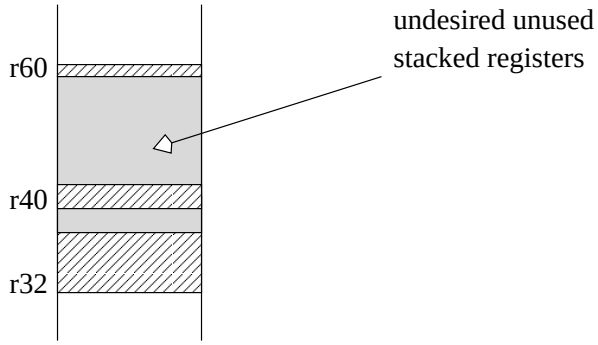


Figure 7.3: Non dense register stack makes the allocation instruction less efficient.

On Figure **??**, the registers $r41$ to $r59$ are not used while $r40$ and $r60$ are. The register stack frame is not dense. If the situation occurs before a function call, the allocation instruction is useless. We would like to see the register allocator assigning $r41$ to hold the value currently assigned to $r60$. 18 registers would be

saved. We do not expect all the holes in the frame to be filled, but at least to reduce their size.

The idea is to assign long-life registers to the lowest register numbers while short-life registers are assigned at the top of the register stack frame. Assuming the compiler uses graph coloring for register allocation ([?][?][?]), we want to make sure the color assignment routine follows the previous recommendation. Instead of choosing the first color available, the life range would be considered.

However the allocation instruction aware register assignment algorithm needs to know in advance the final live ranges (after register coalescing) and the maximum number of stacked registers going to be used. Therefore we would better imagine a post-pass algorithm that would simply switch register number assignment depending on the live ranges.

The register allocator is so linked to the allocation instruction insertion algorithm that the idea deserves to be explored and studied. Depending on different compiler choices and optimizations, the optimization may not be needed.

7.2 Study of the Allocation Instruction

7.2.1 Register Stack Use

The goal of the thesis is to reduce the size of the register stack in order to avoid spills and restores. But if the maximum number of registers in the stack R_T is never reached, there is no need for such an optimization. We might expect recursive functions to be very good candidates. The depth of the call stack is an important factor. The larger the stack is, the more likely the limit of the register stack is reached and spills occur.

Therefore there is a need to study the behavior of the register stack. Is the maximum number of stacked registers quickly reached? For every routine? Can we characterize the routines that are very likely going to need the allocation instruction insertion optimizations?

A simple study would be to monitor the overall size of the register stack. Whenever the register stack size is increased and the resulting size is greater than R_T, we know that the allocation instruction spills registers. By monitoring the size of the register stack, we can evaluate how often the case appears and judge about the importance of an efficient allocation instruction placement algorithm.

Interesting results could appear as a curve of the size of the register stack over the time, or as a ratio of the execution time spent with a register stack size greater than R_T. The ratio could be subdivided for each function in the measured application. By knowing which functions are more likely to create the spills, we could choose to turn on the optimizations only on specific part of programs by using feedback profiling.

The study could also lead to a better tuning of the allocation instruction insertion algorithm. Maybe only preallocation and postallocation instructions matter and the impact of the other types of allocation instructions is negligible. Then the algorithm could be simplified and the final code cleaned from useless allocation instructions.

7.2.2 Cost of the Allocation Instruction

Another issue about the fine-tuning of the algorithm is the real cost of the allocation instruction. The IA-64 architecture manuals ([**?**][**?**]) tells us that the *alloc* instruction makes use of unused memory bandwidth and stalls the processor if needed.

However the real specifications depends of the actual processor and may vary with the context. Is it as cheap to stall the processor for one register than for 30? How often can we make use of memory bandwidth? How long does it take to stall the processor, spill/restore stacked registers to/from the memory and restart the processor again?

Such a study could be combined with the register stack use and helps to understand the need for an efficient allocation instruction insertion algorithm. To choose a threshold value for the number of register that must be saved before allowing propagation for instance (Section ??) would help.

Chapter 8

CONCLUSION

8.1 Contributions

This thesis proposed a theoretical linear efficient solution to the OAIP problem. Starting from non-optimal straightforward solutions, we built an algorithm that allocates the exact number of stacked registers needed by every control path in a given control-flow graph with the minimum number of allocation instructions. The idea is to introduce one allocation instruction in every basic block and simplify the control-flow graph in a bottom-up fashion until there is only one allocation instruction per control path.

Then we considered more practical versions of the algorithm, where the allocation instructions are inserted only when the number of stacked registers was large enough. We also proposed solutions to avoid redundant calls of allocation instructions in loops and to take advantage of the frequency of execution information if available. All the solutions are still linear but the static optimality is lost. However the solutions are expected to run faster.

We studied the implementation of the allocation instruction in the IA-64 architecture and the constraints related to the specific Intel architecture like the rotating registers and predication.

8.2 Future Work

The allocation instruction and the register stack as implemented in the IA-64 architecture are a relatively new concept and more investigations need to be

done. The need for an efficient allocation instruction insertion algorithm must be evaluated, as well as the efficiency of the algorithms proposed in this dissertation. The interactions of the solutions with existing compiler optimizations, like copy propagation or register allocation, must also be assessed.

MAIA should be studied furthermore and the optimality proved or a counter-example found. If MAIA is not optimal, how far from an optimal solution is the solution given by MAIA? An acceptable value for the upper bound for the number of parents for any basic block in the control-flow graph should also be studied.

Mainly, as soon as the processors using the register stack are available on the market, the algorithms and the allocation instructions should be evaluated.