The Power of Belady's Algorithm in Register Allocation for Long Basic Blocks

Jia Guo, María Jesús Garzarán, and David Padua
University of Illinois at Urbana-Champaign
{jiaguo, garzaran, padua}@cs.uiuc.edu
http://polaris.cs.uiuc.edu

Abstract

Optimization techniques such as loop-unrolling and trace-scheduling can result in long straight-line codes. It is, however, unclear how well the register allocation algorithms of current compilers perform on these codes. Compilers may well have been optimized for human written codes, which are likely to have short basic blocks. To evaluate how state-of the art compilers behave on long straight-line codes we wrote a compiler that implements the simple Belady's MIN algorithm.

The main contribution of this paper is the evaluation of Belady's MIN algorithm when used for register allocation for long straight-line codes. These codes were executed on a MIPS R12000 processor. Our results show that applications compiled using Belady's MIN algorithm run faster than when compiled with the MIPSPro or GCC compiler. In particular, Fast Fourier Transforms (FFTs) of size 32 and 64 run 12% and 33% faster than when compiled using the MIPSPro compiler.

1 Introduction

In modern processors, well-known optimizations such as loop-unrolling and trace-scheduling are useful to increase the Instruction Level Parallelism (ILP). These techniques can be applied by a compiler or a library generator. Two examples of the latter are: SPIRAL [20, 23] that generates Digital Signal Processing (DSP) libraries, and ATLAS [22] that generates Basic Linear Algebra Subroutines (BLAS). These systems generate highly tuned libraries by applying transformations and using empirical search to find the best shape of the transformations. One of the values they search for is the degree of loop unrolling, and they find that the unrolled versions with large degrees of unroll obtain better performance. These unrolled versions contain very long basic blocks.

ATLAS and SPIRAL produce high-level code incorporating program transformations. A compiler is then used to translate the high-level code into machine code. The compiler is thus responsible for the low-level optimizations such as instruction scheduling and register allocation. However, given that the code produced by these library generators may contain long basic blocks (with sizes ranging from 300 to 5000 statements), it is unknown whether the register allocation algorithms in the today's compilers are effective. If they do not, we could miss the most important optimization technique since register allocation adds the largest single performance improvement to compiled programs [12].

A standard solution to the problem of register allocation is to use graph coloring [5]. Graph coloring tries to assign registers so that simultaneously live values are not assigned the same register. Coloring produces an optimal solution when there are no spills. However when the number of values is larger than the number of registers, some registers need to be spilled. Since coloring and spilling is an NP-complete problem [9], some heuristics are used to select the register to spill [3, 4, 5, 10]. It was argued in [14] that these heuristics work well in most cases because programs usually contain small procedures and basic blocks where register spilling is unlikely to happen.

In the literature, the task of assigning values to registers over an entire block of straight-line code is done by local register algorithms. The goal of these algorithms is to minimize the traffic between memory and registers. Horwitz [13] published the first algorithm on register allocation in straight-line programs that minimizes the number of load and stores. Faster algorithms [14, 15, 17] were later proposed to achieve the same goal. A simpler algorithm that can be applied for register allocation in long basic blocks is based on Belady's MIN [2] algorithm. The idea is that, on a register replacement, the register to replace is the one that contains the value with the farthest next use. This heuristic guarantees the minimum number of reloads. However, it does not guarantee the minimum number of stores because it does not take into account whether the register to replace needs to be written back to memory. In this paper, we evaluate the performance of the Belady's MIN algorithm for register allocation. We developed a back-end compiler that implements Belady's MIN algorithm and used it to compile codes implementing FFTs and Matrix Multiplication (MM), that were generated by SPIRAL and ATLAS, respectively.

The main contribution of this paper is the evaluation of Belady's MIN algorithm for register allocation for long basic blocks (more than 1,000 lines). We measured performance by running the codes on a real machine (MIPS R12000 processor), and we compare with the performance obtained by codes generated by the MIPSPro and GCC compilers. To the best of our knowledge this is the first report of the performance of Belady's MIN algorithm for a modern out-of-order superscalar processor. In fact, previous papers have evaluated the performance of this algorithm primarily

by measuring the number of load and stores, and rarely report the execution time on the real machine.

Our results show that our compiler always performs better than GCC. In addition, SPIRAL generated FFTs of size 32 and 64, when compiled using Belady's MIN algorithm run 12% and 33% faster than when compiled using the MIPSPro compiler. For the MM generated by ATLAS, Belady's MIN algorithm can also execute faster than the the MIPSPro compiler by an average of 10% for highly unrolled loops. However, for MM, the best performance in the MIPS processor is achieved by limiting the amount of unrolling. For the limited unrolling, our compiler and MIPSPro obtain the same performance. Our experiments show that when the number of live variables is smaller than the number of registers, MIPSPro and our compiler have similar performance. However, as the number of live variables increases, register allocation seems to become more important. In this situation of high register pressure, the simple Belady's MIN algorithm performs better than the MIPSPro compiler, although MIN is not an optimal register allocation algorithm.

This paper is organized as follows. Section 2 outlines some of the characteristics of the code that we used for register allocation, and shows the performance benefit obtained when unroll is applied. Section 3 explains the implementation of our compiler. Section 4 evaluates performance. Section 5 presents related work, and Section 6 concludes.

2 Long Straight-line Code

In modern processors, loop unrolling is a well-known transformation that compilers apply in order to increase the Instruction Level Parallelism (ILP). Loop unrolling improves performance because fewer bookkeeping operations are executed. Although unrolling is usually beneficial, too much unrolling produces code explosion that could result in instruction cache overflow or register spilling. If either of these happens, performance degrades. Thus, different methods are used to control the amount of unrolling. One approach applied by many compilers is to use heuristics to decide the degree of unrolling. Another approach, taken by SPIRAL [20, 23] and ATLAS [22], is to search for the best amount of unrolling.

As mentioned above, SPIRAL and ATLAS use empirical optimizations to find the best values for important transformation parameters. Empirical optimizers generate different versions of the program with different forms of the transformation they apply, run the code after applying the transformation, and choose the transformation that achieves the best performance. One of the transformations tested is the degree of unrolling of loops. The codes tested are implementations of FFT in the case of SPIRAL and MM in the case of ATLAS.

Figure 1 shows the speedup of the unrolled versions over the non-unrolled versions in FFT codes of size 16-64, and MM. For each case, two bars are shown. The first one (SPARC) corresponds to the speed-up obtained when the codes were executed on a UltraSparcIII 900 Mhz, and the compiler was the Workshop SPARC

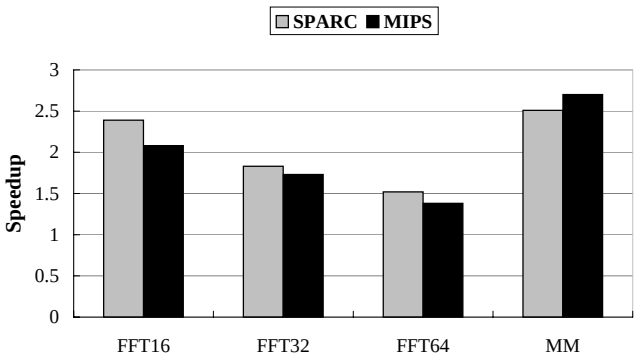


Figure 1. Speedup obtained by unrolling FFTs of sizes 16, 32 and 64 and Matrix Multiplication (MM).

compiler, version 5.0. In the second bar (MIPS) the codes were executed on a R12000 270 Mhz processor, and the compiler was the MIPSpro compiler version 7.3.3.1m. For the MIPSPro compiler, the compilation flags were set to the values specified in Table 2. For the SPARC compiler, the flags were set to "-fast -O5 -silent".

The results for the the FFT bars were collected using SPIRAL which applies dynamic programming as a heuristic to find the formula that results in the best performance for a particular FFT size. We used SPIRAL to find the best formula with unroll and the best one without unroll. We used these two formulas to compute the speedup shown in Figure 1. Notice that these two formulas can be different, and what the plot shows is the performance benefit obtained from unrolling.

Figure 1 shows that the best unrolled FFT formula runs between 2.4 and 1.4 faster than the best non-unrolled formula of size 16, 32 and 64. In all the cases, the version that achieved the best performance is the one totally unrolled. Notice that the speedups obtained for SPARC and MIPS are quite similar.

Figure 1 also presents the benefits of unrolling for the Matrix Multiplication(MM). These results were collected using ATLAS [22]. On MIPS, we used a matrix size of 64x64; on SPARC, we used a matrix size of 80x80. These are the optimal tile sizes that ATLAS found for these platforms. ATLAS also does register tiling. The resulting code has several nested loops that are unrolled and jammed.

The optimal degree of unrolling for the two outer loops depends on the number of registers of the machine since too much unrolling can result in register spilling. The degree of unrolling that obtained the best performance is 4x4x64 for MIPS, and 2x2x80 for SPARC [24]. Figure 1 shows that the unrolled version is 2.5 times faster that the non-unrolled version.

The code generated by SPIRAL and ATLAS is compiled using a conventional compiler. However, these codes have basic blocks with sizes that range from 300 - 5,000 statements. These sizes are by far much larger than what any person would write. Although those fully unrolled versions perform much better than the same versions without unrolling, it is not clear whether the compiler does a good job on register allocation for these very long basic blocks, where register replacements are likely to happen fre-

quently. In the next section, we explain the compiler we designed to perform the register allocation on the FFT and MM codes.

3 A Compiler for Long Straight-line Code

Belady's MIN [2] is a replacement algorithm for pages in virtual memory. On a replacement, the page to replace is the one with the farthest next use. The MIN algorithm is optimal because it generates the minimal number of physical memory block replacements. However, in this context of virtual memory the MIN algorithm is impractical in most of the applications, because it is usually not known which memory block will be referenced in the future.

Belady's MIN algorithm has been proposed to use for register allocation in long basic blocks, where the compiler knows exactly the values that will be used in the future. In this context, the MIN algorithm is also known as *Farthest First*(FF) [6] since, on a register replacement, the register to replace *first* is the one holding the value with the *farthest* next use.

The MIN algorithm is not optimal for register allocation since the replacement decision is simply based on the distance and not on whether the register has been modified. When a register holds a value that is not consistent with the value in memory, we say that the register is *dirty*. Otherwise, we say that the register is *clean*. If the register to be replaced is dirty, the value of the register needs to be stored back to memory before a new value can be loaded into it. Thus, for a given instruction scheduling the MIN algorithm guarantees the minimum number of register replacements, but it does not guarantee the minimum traffic with memory, that is, the minimum number of load/stores. In our implementation, when there are several candidates for replacement with the same distance, our compiler chooses the one with the clean state to avoid an extra store.

In order to further reduce the number of stores, another simple heuristic called *Clean First* (CF) was proposed in [6]. With this heuristic, when a live register needs to be replaced, CF first searches in the clean registers. The clean register which contains the value with the farthest next use is chosen. If there are no clean register, the most distant dirty one is chosen.

We implemented a back-end compiler that uses the MIN and the CF algorithms for register allocation. Next, we describe the implementation details of our compiler.

3.1 Implementations Details

We built a simple compiler that translates high-level code into assembly code. Our compiler assumes that all the optimizations, including instruction scheduling, have been applied before the high-level code is generated. It only performs register allocation using Belady's MIN or the CF heuristic explained above.

The compiler has two steps. In the first step we transform the long straight-line code into a static single-assignment (SSA) form and build the definition-use chain for all the variables. At the second step we do register allocation as shown in Figure 2. The second step is fairly much along the lines of the simple algorithm described in [1].

The algorithm can be implemented more efficiently. The register file can be a priority queue implemented with a binary heap, where the higher priority is given to the farther next use. Operations such as extracting the register with the farthest next use can be executed in $O(logR)$, where R is the number of registers. So the time complexity for MIN algorithm is $s \times O(logR)$, where s is the number of references to the variables in the program.

```
DATA STRUCTURES:
register_file:
    Array of registers. Each register r has 3 fields:
        - state: Indicates whether the register is clean.
        - var: The current variable in the register.
        - addr: The address of the variable in the register.
next_use_list:
    List used to keep the stmt where each variable is used/defined.
    Each node in the list has 2 fields:
        - stmt: The statement number.
        - status: Definition or use.
FUNCTIONS:
NEXT_USE (reg, stmt):
    Returns the statement number where reg.var is used next after stmt.
MIN_REG_ALLOC (var, stmt):
    Returns the register that is going to be used for the variable var at stmt.

MAIN
    for each stmt in the program do begin
        for each v on RHS do begin
            reg ← MIN_REG_ALLOC (v, stmt)
            generate instruction "load reg, reg.addr"
        end
        for v on LHS do begin
            reg ← MIN_REG_ALLOC (v, stmt)
        end
    end
    generate assembly instruction "op r1 r2 r3"

MIN_REG_ALLOC (v, stmt)
    if v is in reg
        return reg
    farthestUse ← 0
    regToUse ← 0
    for each reg in register_file do begin
        if reg is empty
            return reg
        if variable in reg is dead
            return reg
        nextUse ← NEXT_USE (reg, stmt)
        if farthestUse < nextUse
            farthestUse ← nextUse
            regToUse ← reg
        else if farthestUse=nextUse and reg.state = CLEAN
            regToUse ← reg
    end
    if regToUse.state = DIRTY
        generate register spill "store regToUse, regToUse.addr"
    return regToUse
```

Figure 2. Pseudocode for Belady's MIN algorithm.

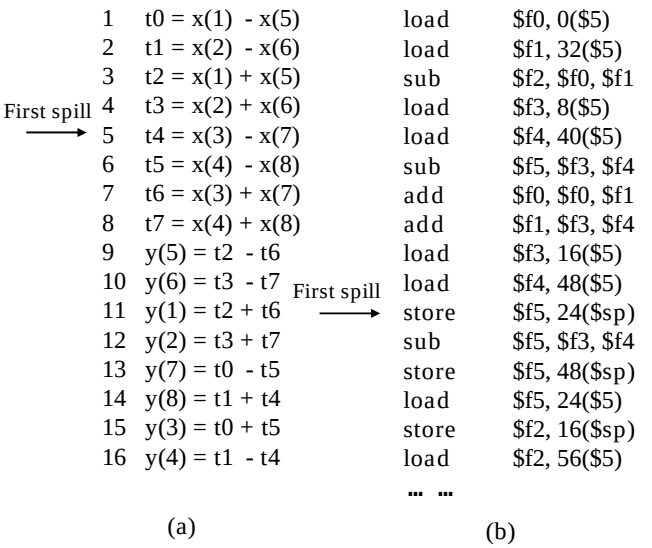


Figure 3. An example of register allocation using Belady's MIN algorithm. (a) Source code for FFT of size 4. (b) The resulting assembly code using Belady's MIN algorithm.

Figure 3 gives an example of how our compiler works using the MIN algorithm. The code in Figure 3-(a) corresponds to the code for the FFT transform of size 4. It has been generated using SPIRAL [20, 23]. Suppose the number of Floating Point (FP) registers is 6. At statement 5, a register needs to be replaced for the first time. Table 1 gives a snapshot of the data structures at statement 5. The variable t1 has the farthest next use. As a result, register $f5 is replaced. Since $f5 is dirty, there is a register spill. Figure 3-(b) shows the assembly code after doing register allocation for the code in Figure 3-(a).

Reg	$f0	$f1	$f2	$f3	$f4	$f5
Current variable	~~x(1)~~ t2	~~x(5)~~ t3	t0	~~x(2)~~ x(3)	~~x(6)~~ x(7)	t1
State	dirty	dirty	dirty	clean	clean	dirty
Addr	40($sp)	32($sp)	16($sp)	16($5)	48($5)	24($sp)

(a)

Variable	t2	t3	t0	x(3)	x(7)	t1
Du-chain	(3,d) (9,u) (11,u)	(4,d) (10,u) (12,u)	(1,d) (13,d) (15,u)	(5,u) (7,u)	(5,u) (7,u)	(2,d) (14,u) (16,u)
Next use	9	10	13	7	7	14

(b)

Table 1. Important data structures used in the MIN algorithm. (a) The register file (b). The du-chain and next use at statement 5.

Notice that our compiler follows the instruction scheduling embedded in the source code; that is, our compiler schedules the arithmetic operations in the same order as they appear in the source code. Also, while most compilers do aggressive scheduling for load operations, ours does not. Compilers try to hoist loads a significant distance above the register is used so that cache latency can be hidden on a miss. However, our compiler loads values into the registers inmediately before they are used. Although load hoist can increase register pressure, some loads could be moved ahead of their use without increasing register pressure. This would simply require an additional pass to our compiler. However, we have not implemented that.

We placed each long basic block in a procedure in a separate file, and then we call our compiler to do the register allocation and generate the assembly code. When the generated code uses registers that contain values from outside of the procedure, we save them at the beginning of the procedure, and restore them at the end.

4 Evaluation

4.1 Environmental Setup

In this section, we compare our compiler against GCC and MIPSPro compilers and evaluate how well they behave on long straight-line codes. For the evaluation, we have used the already optimized unrolled codes obtained from SPIRAL and ATLAS. SPIRAL produces Fortran codes, while ATLAS produces C code. Tabled 2 shows the version and flags that we used for MIPSPro and GCC compilers. Our compiler is the one described in Section 3.1 that implements MIN and CF algorithms. Remember that our compiler schedules operations in the same order as they appear in the source code generated by ATLAS and SPIRAL. Both ATLAS and SPIRAL perform some kind of instruction scheduling in the source code. MIPSPro and GCC, however, rearranges the SPIRAL or FORTRAN code. As a result, their instruction scheduling is different from ours.

Appl.	Compiler	Version	Flags
SPIRAL (Fortran code)	MIPSPro	7.3.2.1.m	-OPT:Olimit=0 -03
	G77	3.2	-O3
ATLAS (C code)	MIPSPro	7.3.1.1m	-O3 -64 -OPT:Olimit=15000 -TARG:platform=IP27 -LN0:blocking=OFF -LOPT:alias=typed
	GCC	3.2	-fomit-frame-pointer -03

Table 2. Compiler Version and Flags for MIPSPro and GCC.

All the experiments were done on a MIPS R12000 processor with a frequency of 270 MHz. The machine has 32 floating point registers, a L1 Instruction Cache of 32 KB, and a L1 Data Cache of 32 KB. In all the experiments that use MIPSPro and our compiler, the code fits into the L1 Instruction cache, like the data fit into

the L1 Data Cache. However, in a few cases where the code was compiled with GCC, it did not fit into the instruction cache (we point this out in the evaluation in next section). Finally, notice that integer registers are not a problem in the FFT or MM codes since the machine has 32 registers and we only need a few of them.

Next, we study the effectiveness of our compiler on the long straight-line code of FFT (Section 4.2) and MM (Section 4.3). Finally, in Section 4.4, we summarize our results.

4.2 FFT

The FFT code that we use is the code generated by the SPIRAL compiler. In this section, we first study the characteristics of the FFT code generated by the SPIRAL compiler (Section 4.2.1) and then we evaluate the performance (Section 4.2.2).

4.2.1 SPIRAL and FFT Code

SPIRAL translates formulas representing signal processing transforms into efficient Fortran programs. It uses intelligent search strategies to automatically generate optimized DSP libraries. In the case of FFT, SPIRAL first searches for a good implementation for small-size transforms, 2 to 64, and then searches for a good implementation for larger size transforms that use the small-size results. For FFT sizes smaller than 64, SPIRAL found that straight-line code achieves the best performance since loop control overheads are eliminated and all the temporary variables can be scalars.

To better understand the performance results, in next section we first study the patterns that appear in the FFT code. Some of these patterns are due to the way SPIRAL generates code, while others are intrinsic to the FFT nature. Patterns that come from SPIRAL are: 1) Each variable is defined only once; that is, every variable holds only one value during its lifetime. For that, SPIRAL uses renaming. 2) If a variable has two uses, at most one statement is between the two uses of the variable. This means that the two uses of a variable are very close to each other. On the other side, patterns that are intrinsic to FFT are: 3) Each variable is used twice at most. 4) If two variables appear in a pair on the RHS of an expression, then they always appear in a pair, and they appear twice. Figure 3-(a) shows an example of FFT code of size 4, where these patterns are shown. In that example, array x is the input, array y is the output, and $ti's$ are temporary variables. As it can be seen, the input array x has two uses. The uses of $x(i)$ or ti variables always appear in pairs, and there is only one statement between these two uses.

This FFT code generated by SPIRAL is used as the input for our compiler. Therefore, given the proximity of the two uses of each variable in the SPIRAL code, any compiler would minimize register replacements by keeping the variable in the same register during the two uses. As a result, the two uses of a variable can be considered as a *single* use. Thus, the problem of register allocation for the FFT code generated by SPIRAL can be simplified as the problem of register allocation where each variable is defined once and used once.

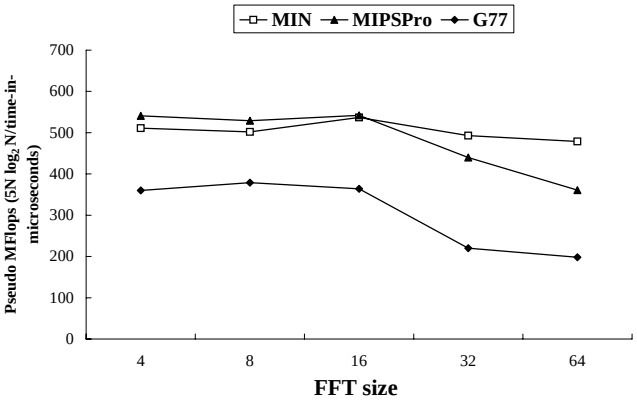


Figure 4. Performance of the best formula for FFTs 4 - 64.

Based on this simplified model, register replacement only occurs between the definition and the use of the variable. One consequence is that the MIN and CF algorithm behave similarly and they always choose the same register to replace. In addition, since the MIN algorithm implemented on our compiler is known to produce the minimum number of register replacements, we can claim that for the FFT problem and given SPIRAL scheduling, our compiler generates the *optimal* solution, the one with the *minimum number of loads and stores*. In the next section we evaluate the performance differences between this *optimal* solution and the MIPSPro or G77 compilers.

4.2.2 Performance Evaluation

Size	Backend	LOC	spills	reloads
4	MIPSPro	34	0	0
	MIN	34	0	0
8	MIPSPro	90	0	0
	MIN	95	0	0
16	MIPSPro	266	9	14
	MIN	276	2	2
32	MIPSPro	921	150	212
	MIN	764	34	34
64	MIPSPro	2468	552	606
	MIN	1944	112	112

Table 3. Characteristics of the code that MIPSPro and MIN generate for FFTs 4 - 64.

SPIRAL does an exhaustive search to find the fastest formula for a given platform. We studied the performance obtained by the best formula when the code was compiled using the MIPSPro compiler, G77, or our compiler. Figure 4 shows the best performance obtained for FFTs of size 4 - 64 using the MIPSPro compiler (MIPSPro), the G77 compiler (G77), or our compiler (MIN). The performance is measured in terms of "pseudo MFlops", which is the value computed by using the equation *5Nlog2N/t*, where *N* is the size of the FFT and *t* is the execution time in microseconds. Notice

that the formula that achieves the best performance can be different in each case.

We focus on MISPPro and MIN, since G77 is always much slower. To help understand the results, Table 3 shows the number of lines of the assembly code (LOC), spills, and reloads for each point in Figure 4. A spill is a store of a value that needs to be loaded again later. A reload is a load of a value that previously was in a register. The data in Table 3 show that, using MIN, the number of spills and reloads is always the same. This is due to SPIRAL scheduling. As Table 3 shows, for FFTs of size 4 and 8, the 32 FP registers are enough to hold the values in the program, and as a result there is no register replacement. Thus, the difference in performance between MIPSPro and MIN comes from the differences in instruction scheduling. From FFTs of size 16, we start to see some spills and reloads, and MIN overcomes the effects of instruction scheduling and obtains the same performance as MIPSPro. Finally, for FFTs of size 32 and 64, since the amount of spilling is larger, the effect of instruction scheduling becomes less important, and MIN outperforms MIPSPro. MIN performs 12% and 33% better than MIPSPro for FFTs of size 32 and 64 respectively.

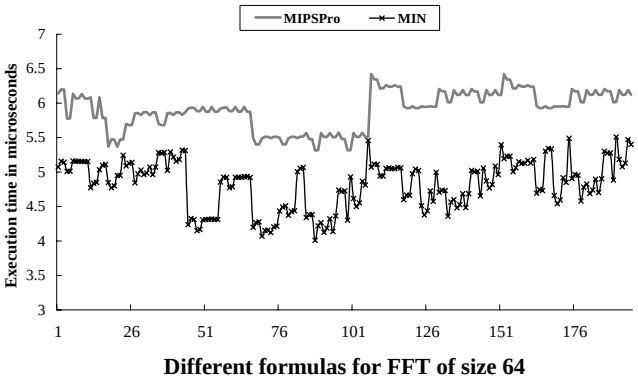


Figure 5. Performance of the different formulas for FFTs of size 64.

In Figure 5, we show the execution time of several FFT codes of size 64 that SPIRAL produced using different formulas. For each formula we show two points. One corresponds to the performance obtained when the SPIRAL code for that formula was compiled using the MIPSpro compiler (MIPSPro), or using our compiler (MIN). On average, MIN runs 18% faster than MIPSPro. In addition, the figure shows that our compiler *always* performs better. As before, it seems to us that as register pressure increases, register allocation becomes the dominant factor.

4.3 Matrix Multiplication

In this section, we study the performance of register allocation for the matrix multiplication code produced by ATLAS. We first describe ATLAS (Section 4.3.1) and then present the performance evaluation (Section 4.3.2).

4.3.1 Overview of ATLAS

ATLAS is an empirical optimizer whose structure is shown in Figure 6. ATLAS is composed of i) a *Search Engine* that performs empirical search of certain optimization parameters values and ii) a *Code Generator* that generates C code given these values. The generated C code is compiled, executed, and its performance measured. The system keeps track of the values that produced the best performance, which will be used to generate a highly tuned library.

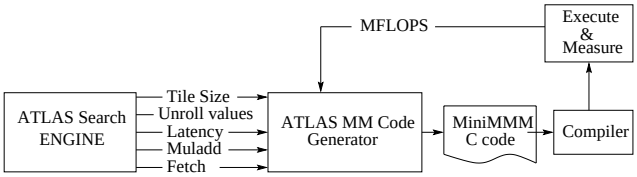


Figure 6. Empirical optimizer in ATLAS.

For the search process, ATLAS generates a matrix multiplication of size *Tile Size* that we call MiniMMM. This code for the MiniMMM is itself tiled to make better use of the registers. Each of these small matrix multiplications multiplies a MUx1 submatrix of A with a 1xNU submatrix of B, and accumulates the result in a MUxNU sub-matrix of C. We call these micro-MMMs. Figure 7 shows a pictorial view and the pseudo-code corresponding to the mini-MMM after register tiling and unrolling of the micro-MMM. The codes of a micro-MMM are unrolled to produce a straight-line of code. After the register tiling, the K loop in Figure 7 is unrolled by a factor KU. The result is a straight-line code that contains KU copies of the micro-MMM code. Figure 9-(a) shows two copies of the micro-MMM that corresponds to the unrolls MU=4 and NU=2 shown in Figure 7. Notice that the degree of unroll MUxNU determines the number of FP registers required. This number is MUxNU to hold the C values, MU to hold the A values, and NU to hold the B values.

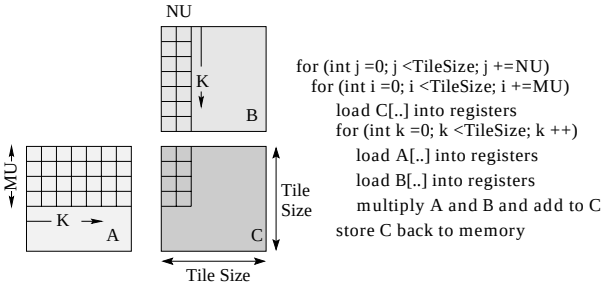


Figure 7. Mini-MMM after register tiling.

4.3.2 Performance Evaluation

To evaluate our compiler, we ran it on the miniMMM code generated by ATLAS that contains the straight-line code explained above. Figure 8 compares the performance in MFlops for different values of MUxNU unroll using the MIPSpro compiler (MIPSPro), GCC compiler (GCC), or our compiler with the MIN algorithm (MIN) (the line MINSched in the figure will be explained later).

For this experiment the rest of the parameters of the miniMMM have been set to the values that ATLAS found to be optimal [24]. In particular, TileSize and KU have been set to 64 [1]; that is, the innermost k loop in Figure 7 is totally unrolled. Notice that for the order that ATLAS uses for register blocking, while unrolling along the k dimension does reduces loop overheads, it does not increase register pressure.

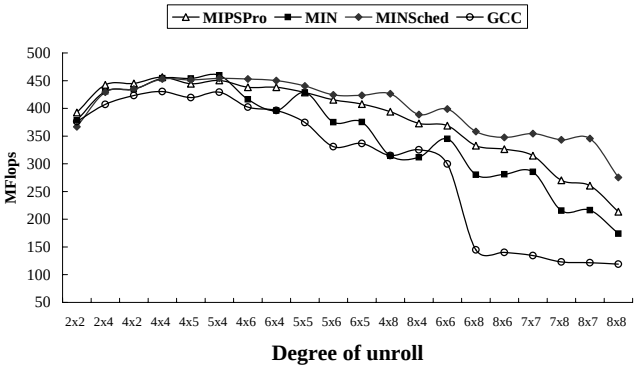


Figure 8. Performance versus MUxNU unroll for the mini-MMM.

Figure 8 shows that, as before, GCC has the worst performance. In particular the sharp drop for unrolls 6x8 and larger are due to the size of the code, that overflows the 32KB instruction cache of the MIPS R12000 processor. We now focus on MIPSPro. Figure 8 also shows that MIN behaves almost like MIPSPro when the MU and NU are small and, as a result, there is no register replacement. Thus, the slightly better performance of MIN is mostly due to differences in instruction scheduling. Register replacement occurs for unrolls 4x4 and larger in MIPSPro, and unrolls 4x5 and larger in MIN. Unfortunately, for unrolls 4x6, where there are more values than registers and register replacement occurs, MIN performs worse than MIPSPro.

The MIN algorithm performs worse because of the particular scheduling of the operations in the MM in ATLAS. The certain code scheduling incurs a register spill for each replacement and additional dependences. Figure 9-(a) shows the micro-MMM generated by ATLAS for an unroll of 4x2 that is the input to our compiler. The code in Figure 9-(b) is the resulting assembly code after our compiler does register allocation for the first few instructions. For the example we have assumed that we have only 6 FP registers. It can be seen that when register replacement starts (at statement 3), the farthest next use is the variable $c1$ that we have just computed. For this particular scheduling, variables ci always have the furthest next use. The registers holding the ci variables are in dirty state and, as a result, its contents need to be written back to memory. This results in an increase in memory traffic because, at each register replacement, we have a register spill that generates one $store$. In addition to the spills, the compiler introduces additional dependencies by allocating the same register to independent instructions (storage-related dependence). For instance, although all the $madds$ are independent, due to the farthest next use of the MIN algorithm, we always spill the register \$f4. Thus, we have created a chain of dependences along the instructions using register \$f4, as shown in Figure 9-(b). Thus, the performance of MIN decreases, as shown in Figure 8. We also tried our compiler using the CF heuristic, but the performance was even worse. The CF heuristic always replaces the registers containing the A and B values that tend to be needed shortly again.

We looked then at the instruction scheduling in the MIPSPro assembly code. Since the MIPSpro assembly code was the result of instruction scheduling and register allocation, we extracted the scheduling of the $madd$ instructions in the MIPSpro assembly code and obtained the code shown in Figure 9-(c). We ran our compiler on the code with the new scheduling to do the register allocation. The resulting assembly code is shown in Figure 9-(d). We executed this code and the performance obtained is line MINSched in Figure 8. Now for unrolls larger than 4x6, our compiler behaves better. As Figure 9-(d) shows, with this new scheduling, the ci variables have a higher reuse rate. Since the registers containing these variables are in dirty state, this new scheduling helps to reduce register spilling.

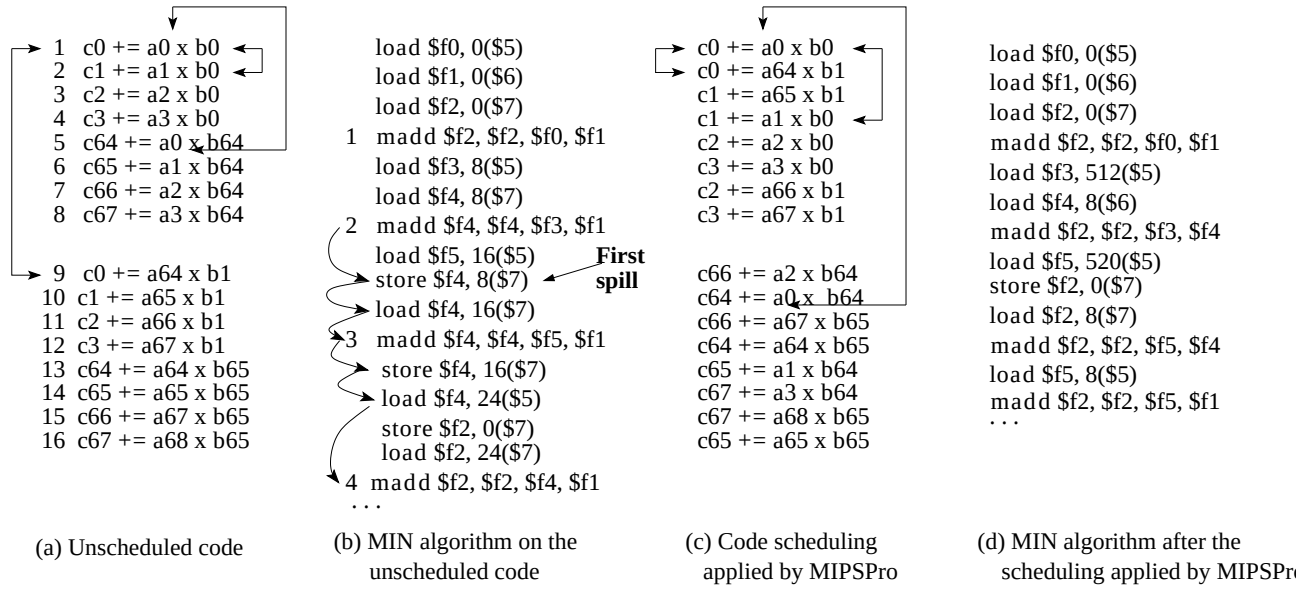
Table 4 helps to understand the results in Figure 8. For each degree of unroll, we show the number of lines of the assembly code (LOC), spills, and reloads. Table 4 shows that MINSched always has fewer spills and reloads than MIPSPro. As the unroll grows and register pressure becomes more prominent, the fewer spills and reloads result in the better performance of MINSched (Figure 8). On average, for unrolls larger than 4x6, MINSched performs 10% better than MIPSPro. Finally, we also tried the CF algorithm with the new scheduling, but it performed worse than MIN, so we do not show results for it.

Unroll	Compiler	LOC	spills	reloads
2x4	MIPSPro	917	0	3
	MIN	921	0	0
	MINSched	921	0	0
4x4	MIPSPro	1684	48	64
	MIN	1577	0	0
	MINSched	1604	9	18
4x8	MIPSPro	4046	438	709
	MIN	4673	892	892
	MINSched	3607	356	362

Table 4. Characteristics of the MM code for different degrees of unroll.

We have used the long straight-lines code in the MM in ATLAS as an example of where we apply register allocation. We have shown that MINSched performs better than MIPSPro for large degrees of unroll. However, this improvement is not useful. The reason is that the unroll that obtained the best performance corresponds to the largest unroll before register replacement starts (this point is 4x4 in Figure 8). As a result, ATLAS will select this unroll, where register replacement heuristics are not used.

[1] ATLAS only tries square tiles



(a) Unscheduled code

(b) MIN algorithm on the unscheduled code

(c) Code scheduling applied by MIPSPro

(d) MIN algorithm after the scheduling applied by MIPSPr

Figure 9. Two micro-MMM codes for a miniMMM of size 64, MU=4 and NU=2.

4.4 Analysis

Next we summarize our results. When the straight-line code is such that the number of simultaneously live values is smaller than the number of registers, there is no need to do register replacement. In that case, instruction scheduling is the dominant factor in optimization. This is the case for FFTs of size 4 and 8, as well as for MM, because it is possible to search for an unroll without register spilling or reloads.

When the number of simultaneously live values is larger than the number of registers, register replacement becomes important. In FFT. we observed that as register pressure increases, register allocation becomes more important than instruction scheduling. For FFTs of size 32 and 64, where the number of spills and reloads is larger, register allocation becomes important, and our compiler achieves a higher performance. The higher performance of our compiler also can be due to the use of the SPIRAL scheduling together with the MIN algorithm, which result in a *optimal* register allocation for that scheduling.

On the other hand, for unrolls larger than 4x6, when register pressure was high for the MM code, the use of ATLAS scheduling and the MIN algorithm resulted in additional dependences. As a result, MIPSPro performed better than our compiler. It is unclear to us whether, by using the scheduling in ATLAS, we could have found an optimal register replacement able to beat the MIPSPro instruction scheduling. However, it is clear that there are schedulings that can reduce register pressure, and these schedules should be used when register spills and reloads become important.

In summary, by using our compiler with the simple MIN algorithm we have beaten the performance obtained by the MIPSPro compiler in long straight-lines of code, when register pressure was high. Today's compilers like MIPSPro or GCC are not optimized to handle this type of codes and, as a result, highly optimized code like those with loop unrolling and trace scheduling could result in sub-optimal performance. Performance could be improved with an appropriate register allocator, and maybe an instruction scheduling chosen to minimize register pressure.

5 Related Work

There are two different approaches for register allocation: global register allocation, and local register allocation. Global register allocation assigns variables to registers throughout the program, while local register allocation assigns registers to variables within basic blocks.

The most commonly used register allocation is based on Graph Coloring [5]. Graph coloring is a global register allocator that tries to assign registers so that simultaneously live values are not assigned the same register. In this approach, the register allocation problem is translated into a graph coloring problem. Nodes in the graph represent live values that are candidates for register allocation. Edges connect live ranges that interfere, that is, that simultaneously live at least at one point in the program. Coloring the graph consists of assigning colors (registers) to the nodes so that two nodes connected by an edge do not receive the same color. Coloring works well when the graph is colorable. However, when the number of values is larger than the number of registers, some registers need to be spilled. Since coloring and spilling is an NP-complete problem[9], some heuristics are used to pick up the correct register to spill [10, 4, 3, 5]. These heuristics work well in most programs because humans tend to write programs will small procedures and basic blocks where register spilling is unlikely to happen [14]. However, the work in [16] showed that the effectiveness of graph coloring is strongly affected as the size of the basic block increases.

However, as shown in this paper, compiler optimization can result in long basic blocks where register spilling becomes necessary. In those cases, local register algorithms should perform better. Local register allocation is the task of assigning values to registers over an entire block of straigh-line code so that the traffic between registers and memory is minimized. Belady's MIN [2] and Horwitz [13] are often treated as optimal algorithms for register allocation in a basic block. Belady's MIN [2] optimizes for the minimal number of register replacements, and not for the minimum number of load/stores. As a result, it may not find the optimal solution. Horwitz's algorithm minimizes the number of loads and stores. Later algorithms [14, 15, 17] are mainly improvements to the compilation efficiency. However, they are still exponential in time and space. On the other hand, Belady's MIN algorithm runs in polynomial time, although it may not find the optimal solution.

Finally, Linear Scan [18, 19, 21] is a type of global register allocation that has become of interest lately. Its interest is that it is a fast technique that results in efficient code and, as a result, can be used in dynamic compilation systems and "just-in-time" compilers. Linear Scan can be seen as an extension of local register allocation algorithms [7, 8, 14], which in turn derive from Belady's MIN.

The problem of register allocation and instruction scheduling in straight-line code has also been studied in the literatur. In particular Goodman [11] proposes two different scheduling algorithms: one tries to minimize pipeline stalls, and the other one tries to reduce register pressure. The algorithm is chosen based on the register pressure. This agrees with our observation in section 4.4.

6 Conclusion

In this paper, we have shown that a simple algorithm like Belady's MIN can beat the performance of state-of the art compilers like the MIPSPro or GCC compilers in long straight-line codes. We have applied Belady's MIN algorithm to codes corresponding to FFTs transforms and Matrix Multiplication that are produced by SPIRAL and ATLAS, respectively. We have measured the performance by running these codes on a real machine (a MIPS R12000 processor).

Our results show that Belady's MIN algorithm is about 12% and 33% faster for FFTs of size 32 and 64. In the case of Matrix Multiplication, it can also execute faster than the the MIPSPro compiler by an average 10%. However, in this application, the unroll that achieves the best performance is the one without register spilling. Our compiler and MIPSPro perform similarly using this unroll. Our experiments show, that when the number of live variables is smaller than the number of registers, MIPSPro and our compiler have similar performance. However, as the number of live variables increases, register allocation seems to become more important. We believe that, in this case of high register pressure, instruction scheduling needs to be considered in concert with register allocation so that the number of register spills and reloads can be minimized.

References

[1] A. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Comapny, 1985.

[2] L. Belady. A Study of Replacement of Algorithms for a Virtual Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[3] P. Bergner, P. Dahl, D. Engebretsen, and M. T. O'Keefe. Spill code minimization via interference region spilling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 287–295, 1997.

[4] P. Briggs, K. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems*, 6(3):428–455, May,1994.

[5] G. Chaitin. Register Allocation and Spilling Via Graph Coloring . In *Proc. of the SIGPLAN 82 Symp. On Compiler Construction*, pages 98–105, 1982.

[6] C. Fischer and T. LeBlanc. *Crafting a Compiler*. Benjamin Cummings, 1987.

[7] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation* . Benjamin/Cummings, Redwood City, CA, 1995.

[8] R. A. Freiburghouse. Register allocation via usage counts. *Communications of the ACM*, 17:638–642, November,1974.

[9] M. Garey and D. Johnson. *Computers and Intracdtability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1989.

[10] L. George and L. Appel. Iterated Register Coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May,1996.

[11] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 2nd international conference on Supercomputing*, pages 442–452. ACM Press, 1988.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[13] L. Horwitz, R. M. karp, R. E. Miller, and S. Winograd. Index Register Allocation. *Jornal of the ACM*, 13(1):43–61, January,1966.

[14] W. Hsu, C. Fischer, and J. Goodman. On the Minimization of Load/Stores in Local Register Allocation. *IEEE Transactions on Software Engineering*, 15(10):1252–1260, October, 1989.

[15] K. Kennedy. *Index Register Allocation in Straight Line Code and Simple Loops*. Design and Optimization of Compilers, Englewood Cliffs, NJ: Prentice Hall, 1972.

[16] J. R. Larus and P. N. Hilfinger. Register allocation in the spur lisp compiler. In *Proceedings of the 1986 SIGPLAN symposium on Compiler contruction*, pages 255–263. ACM Press, 1986.

[17] F. Luccio. A Comment on Index Register Allocation. *Communications of the ACM*, 10(9):572–574, 1967.

[18] M. Poletto, D. Engler, and M. Kaashoek. tcc: A system for fast, flexible and high-level dynamic code generation. In *Proc. of the International Conference on Programming Language Design and Implementation*, pages 109–121, 1997.

[19] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21:895–913, September,1999.

[20] M. Puschel, B. Singer, J. Xiong, J. Moura, D. Padua, M. Veloso, and R. Johnson. SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms. *To appear in Journal of High Performance computing and Applications. http://www.ece.cmu.edu/ spiral*.

[21] O. Traub, G. Holloway, and M. Smith. Quality and speed in linear-scan register allocation. In *Proc. of the International Conference on Programming Language Design and Implementation*, 1998.

[22] R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT CS-97-366, LAPACK Workig Note No. 131, University of Tenbessee, 1997.

[23] J. Xiong, J. Johnson, R. Johnson, and D. Padua. Spl: A language and a compiler for dsp algorithms. In *Proc. of the International Conference on Programming Language Design and Implementation*, pages 298–308, 2001.

[24] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, , and P. Wu. A comparison of empirical and model-driven optimization. In *Proc. of the International Conference on Programming Language Design and Implementation*, pages 63–76, 2003.