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

Jia Guo, María Jesús Garzarán and David Padua

Department of Computer Science
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 the 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, optimizations such as loop-unrolling and trace-scheduling help increase the Instruction Level Parallelism (ILP). These techniques have traditionally been applied by compilers, and in the recent past, have also been incorporated into library generators. Two examples of the latter are SPIRAL [6, 18] that generates Digital Signal Processing (DSP) libraries and ATLAS [17] that generates linear algebra subroutines. These systems use empirical search to determine the best shape of the transformations to be applied. One of the values they search for is the degree of loop unrolling. Usually large degrees of unrolling improve the performance. The body of these unrolled loops takes the form of 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 are not, we could be missing the benefit of one of the most important compiler optimization techniques [12].

A standard solution to the problem of register allocation is based on graph coloring [4] and it tries to assign registers so that simultaneously live variables are not assigned to the same register. Coloring produces an optimal solution when there are no spills. However when the number of variables is larger than the number of registers, some registers need to be spilled. Since coloring and spilling is an NP-complete problem [9], heuristics are used to select the register to spill [3, 7, 4, 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.

The task of assigning values to registers over an entire block of straight-line code is usually done by local register allocation algorithms. Horwitz [13] published the first algorithm on register allocation for straight-line codes that minimizes the number of load and stores. Faster algorithms [14–16] 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 variable 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 on long basic blocks. 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 an 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 MIPS R12000 processor, and we compared the performance obtained with the performance of the 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, the FFT codes of size 32 and 64 generated by SPIRAL, 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 exper-

iments 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. Under 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 has the additional benefit of reducing the number of bookkeeping operations that are executed. Although unrolling is usually beneficial, too much unrolling could result in instruction cache overflow or increased 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 [6, 18], ATLAS [17], as well as several compilers, is to empirically search for the best amount of unrolling.

As mentioned above, SPIRAL and ATLAS use empirical search to find the best values for important transformation parameters. Empirical search generates 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. When compilers perform empirical search, they usually estimate execution time instead of actually executing the code. 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. In 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 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 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 use SPIRAL to find the formula that

leads to the best performance from all formulas implemented by fully unrolling, and the formula that leads to the best performance when partially unrolling is applied. 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.
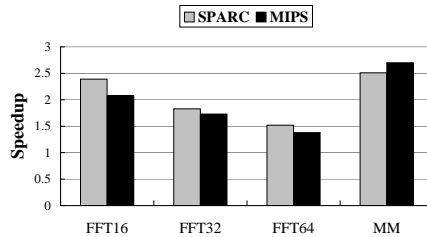

Fig. 1. Speedup obtained by unrolling FFTs of sizes 16, 32 and 64 and Matrix Multiplication (MM).

Figure 1 also presents the benefits of unrolling for the Matrix Multiplication(MM). These results were collected using ATLAS [17]. 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 is produced by applying unroll and jam to the innermost three loops.

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 [5]. Figure 1 shows that the unrolled version is 2.5 times faster that the non-unrolled version.

The code generated by SPIRAL and ATLAS was 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 known how good is the register allocation applied in these compilers to very long basic blocks where register replacements are likely to happen frequently. 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 the context of virtual memory the MIN algorithm is usually impractical because in most cases it is usually not known which memory block will be referenced in the future.

Belady's MIN algorithm has been proposed for use in register allocation of 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) [8] 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 [8]. 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 registers, 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 MIPS assembly code. Our compiler assumes that all optimizations other than register allocation have been applied to the high-level code. Our compiler 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 using two data structures: (1) the register file and (2) the definition-use chain. The register file is an array of registers. Each register has 3 fields: *current var*, *state* and *addr*. The task is to assign a register to a variable. First, our register allocator[1] checks 3 cases in order: 1) whether the variable is already in a register, 2) whether there is a free register, 3) whether there is a variable in a register that will never be used in the future. If one of these easy cases is satisfied, it returns that register. Otherwise, it begins to calculate the next use for every variable in the register

file based on the definition-use chain. It chooses the register with the farthest next use. If there are several candidates, it prefers those registers that have not been modified since loading (clean). If the only option is a dirty register, the compiler generates a store instruction before loading the new variable into the register.

For efficiency, the register file can be a priority queue implemented with a binary heap, where the higher priority is given to the farther next use. Registers not used are assigned to distance ∞ and therefore would have the highest priority. 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.

```
            1   t0 = x(1)  - x(5)       load    $f0, 0($5)
            2   t1 = x(2)  - x(6)       load    $f1, 32($5)
            3   t2 = x(1) + x(5)        sub     $f2, $f0, $f1
First spill 4   t3 = x(2) + x(6)        load    $f3, 8($5)
        →   5   t4 = x(3)  - x(7)       load    $f4, 40($5)
            6   t5 = x(4)  - x(8)       sub     $f5, $f3, $f4
            7   t6 = x(3) + x(7)        add     $f0, $f0, $f1
            8   t7 = x(4) + x(8)        add     $f1, $f3, $f4
            9   y(5) = t2 - t6          load    $f3, 16($5)
            10  y(6) = t3 - t7  First spill load  $f4, 48($5)
            11  y(1) = t2 + t6      →   store   $f5, 24($sp)
            12  y(2) = t3 + t7          sub     $f5, $f3, $f4
            13  y(7) = t0 - t5          store   $f5, 48($sp)
            14  y(8) = t1 + t4          load    $f5, 24($5)
            15  y(3) = t0 + t5          store   $f2, 16($sp)
            16  y(4) = t1 - t4          load    $f2, 56($5)
                                        ... ...
                    (a)                          (b)
```

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)

Fig. 2. 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.

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

Figure 2 gives an example of how our compiler applies the MIN algorithm. The code in Figure 2-(a) corresponds to the code for the FFT transform of size 4 generated by SPIRAL [6, 18]. 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 2-(b) shows the assembly code after doing register allocation for the code in Figure 2-(a).

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 so that cache latency can be hidden on a miss.

However, our compiler loads values into the registers immediately before they are used. Although load hoist can in some cases increase register pressure, some loads could be moved ahead of their use without increasing register pressure. Adding load hoisting 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. Table 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, rearrange the SPIRAL or FORTRAN code. As a result, the instruction schedule they generate is different from ours.

Appl.	Compiler	Version	Flags
SPIRAL	MIPSPro	7.3.2.1.m	-OPT:Olimit=0 -03
(Fortran code)	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, note that integer registers are not a problem in the FFT or MM codes because we only use floating point registers.

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 and then we evaluate the performance.

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 as their components. For FFT sizes smaller than 64, SPIRAL assumes 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, 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 due to the nature of FFT. Patterns that come from SPIRAL are: 1) Each variable is defined only once; that is, every variable holds only one value during its lifetime. 2) If a variable has two uses, at most one statement is between the two uses of the variable. Patterns due to the nature of FFT are: 3) Each variable is used at most twice. 4) If two variables appear on the RHS of an expression, then they always appear together, and they appear twice.

Thus, in Figure 2-(a), array x is the input, array y is the output, and $ti's$ are temporary variables. 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 the two uses.

This FFT code generated by SPIRAL is used as the input to 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 is just the problem of register allocation where each variable is defined once and used once.

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*. We evaluate next the performance differences between this *optimal* solution and the MIPSPro or G77 compilers.

Performance Evaluation. SPIRAL does an exhaustive search to find the fastest FFT 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 3 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 $5Nlog_2N/t$, where N is the size of the FFT and t is the execution time in microseconds. Notice that the formula that achieving the best performance can be different in each case.
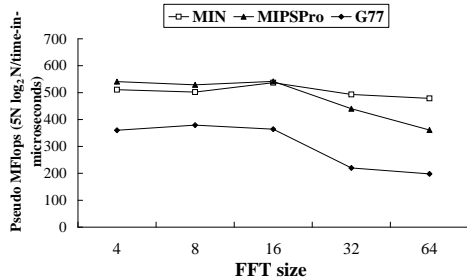

Size	Backend	LOC	spills	reloads
4	MIPSPro	34	0	0
	MIN	34	0	0
8	MIPSPro	90	0	0
	MIN	95	0	0
16	MIPSPro	266	9	14
	MIN	276	2	2
32	MIPSPro	921	150	212
	MIN	764	34	34
64	MIPSPro	2468	552	606
	MIN	1944	112	112

Fig. 3. Performance of the best formula for FFTs 4 - 64.

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

We focus on MISPPro and MIN, since G77 always produces much slower code. 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 3. 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.
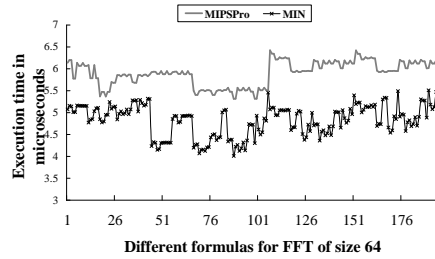
Fig. 4. Performance of the different formulas for FFTs of size 64.

In Figure 4, 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, 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 and then present the performance evaluation .

Overview of ATLAS. ATLAS is an empirical optimizer whose structure is shown in Figure 5. ATLAS is composed of i) a *Search Engine* that performs empirical search of certain optimization parameter 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 matrix multiplication routine.
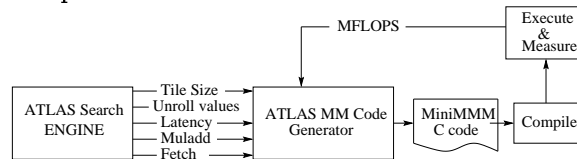
Fig. 5. Empirical optimizer in ATLAS.

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 sub-matrix 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 6 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 6 is unrolled by a factor KU. The result is a straight-line code that contains KU copies of the micro-MMM code. Figure 8-(a) shows two copies of the micro-MMM that corresponds to the unrolls MU=4 and NU=2 shown in Figure 6. 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.


```
for (int j =0; j <TileSize; j +=NU)
  for (int i =0; i <TileSize; i +=MU)
    load C 1..8 into registers
    for (int k =0; k <TileSize; k += KU)
      load A k, 1..4 into registers
      load B 1..2, k into registers
      C 1+= A k,1 * B 1,k
      C 2+= A k,1 * B 2,k
      ...
      C 8+= A k,4 * B 2,k
      Repeat * for k+1 .. k+KU-1
    store C back to memory
```
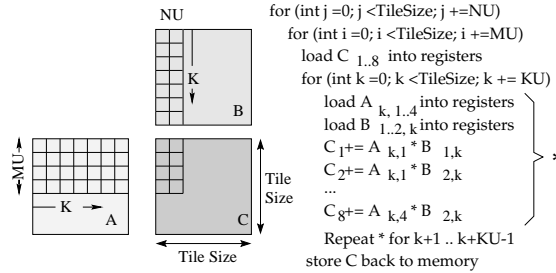
Fig. 6. Mini-MMM after register tiling.

Performance Evaluation. To evaluate our compiler, we ran it on the mini-MMM code generated by ATLAS that contains the straight-line code explained above. Figure 7 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 [5]. In particular, TileSize and KU have been set to 64 [1]; that is, the innermost k loop in Figure 6 is totally unrolled. Notice that while unrolling along the k dimension does reduces loop overheads, it does not increase register pressure.

Figure 7 shows that, as before, GCC produces the code with the worst performance. In particular the sharp drop for unrolls 6x8 and larger is due to the size of the code, that overflows the 32KB instruction cache of the MIPS R12000 processor. Figure 7 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. 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. Figure 8-(a) shows the micro-MMM generated by ATLAS for an unroll of 4x2 that is the input to our compiler. The code in Figure 8-(b) is the resulting assembly code after our compiler does register allocation for the first few instructions. For the example we have assumed

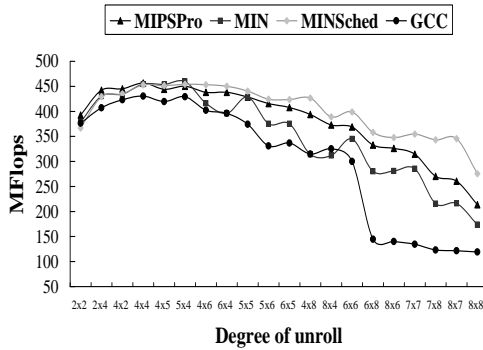---

[1] ATLAS only tries square tiles

Fig. 7. Performance versus MUxNU unroll for the mini-MMM.

Unroll	Compiler	LOC	spill	reload
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.

that we have only 6 FP registers. It can be seen that when register replacement starts (line 3 of 8-(a)), 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 consequently 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 8-(b). As a result, the performance of MIN decreases, as shown in Figure 7. We also tried our compiler using the CF heuristic, but the performance was even worse. The CF heuristic always replaces the registers containing the ai and bi 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 8-(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 8-(d). We executed this code and the performance obtained is line MINSched in Figure 7. Now for unrolls larger than 4x6, our compiler behaves better. As Figure 8-(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 understand the results in Figure 7. For each degree of unrolling, 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.
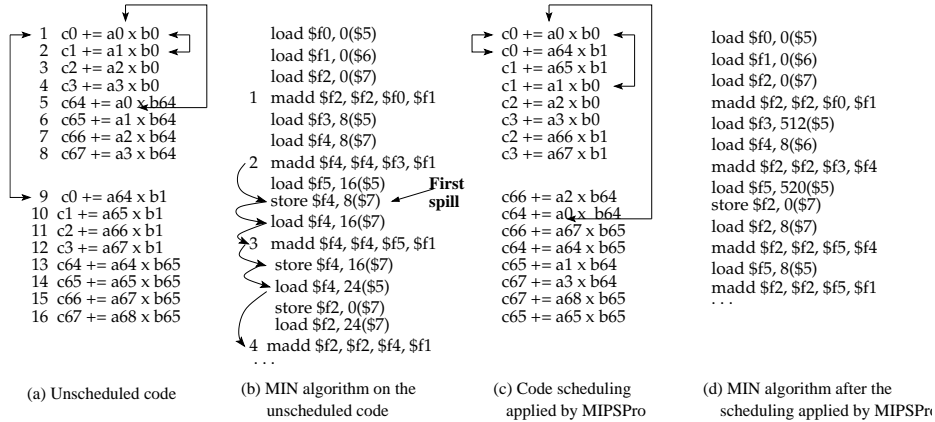
(a) Unscheduled code	(b) MIN algorithm on the unscheduled code	(c) Code scheduling applied by MIPSPro	(d) MIN algorithm after the scheduling applied by MIPSPro
1 c0 += a0 x b0	load $f0, 0($5)	c0 += a0 x b0	load $f0, 0($5)
2 c1 += a1 x b0	load $f1, 0($6)	c0 += a64 x b1	load $f1, 0($6)
3 c2 += a2 x b0	load $f2, 0($7)	c1 += a65 x b1	load $f2, 0($7)
4 c3 += a3 x b0	1 madd $f2, $f2, $f0, $f1	c1 += a1 x b0	madd $f2, $f2, $f0, $f1
5 c64 += a0 x b64	load $f3, 8($5)	c2 += a2 x b0	load $f3, 512($5)
6 c65 += a1 x b64	load $f4, 8($7)	c3 += a3 x b0	load $f4, 8($6)
7 c66 += a2 x b64	2 madd $f4, $f4, $f3, $f1	c2 += a66 x b1	madd $f2, $f2, $f3, $f4
8 c67 += a3 x b64	load $f5, 16($5) **First spill**	c3 += a67 x b1	load $f5, 520($5)
	store $f4, 8($7)		store $f2, 0($7)
9 c0 += a64 x b1	load $f4, 16($7)	c66 += a2 x b64	load $f2, 8($7)
10 c1 += a65 x b1	3 madd $f4, $f4, $f5, $f1	c64 += a0 x b64	madd $f2, $f2, $f5, $f4
11 c2 += a66 x b1	store $f4, 16($7)	c66 += a67 x b65	load $f5, 8($5)
12 c3 += a67 x b1	load $f4, 24($5)	c64 += a64 x b65	madd $f2, $f2, $f5, $f1
13 c64 += a64 x b65	store $f2, 0($7)	c65 += a1 x b64	. . .
14 c65 += a65 x b65	load $f2, 24($7)	c67 += a3 x b64	
15 c66 += a67 x b65	4 madd $f2, $f2, $f4, $f1	c67 += a68 x b65	
16 c67 += a68 x b65	. . .	c65 += a65 x b65	

Fig. 8. Two micro-MMM codes for a miniMMM of size 64, MU=4 and NU=2.

As the unroll grows and register pressure becomes more prominent, fewer spills and reloads result in better performance of MINSched (Figure 7). 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 did not show results for it.

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 7). As a result, ATLAS will select this unroll, where register replacement heuristics are not used.

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 the best performance is obtained for a degree of unrolling 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 an *optimal* register allocation for that scheduling.

On the other hand, for degrees of unrolling 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 better than 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 improved on the performance obtained by the MIPSPro compiler for 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 by an appropriate register allocator, and maybe an instruction scheduling chosen to minimize register pressure.

5 Related Work

Local register allocation is the task of assigning values to registers over a basic block so that the traffic between registers and memory is minimized. Belady's MIN [2] and Horwitz [13] are often used in local register allocation. 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 but it is for index registers, not for general purpose registers. Later algorithms [14–16] 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.

The problem of register allocation and instruction scheduling in straight-line code has also been studied in the literature. 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. It 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. Spill code minimization via interference region spilling. In *SIGPLAN Conf. on PLDI*, pages 287-295, 1997.
4. G. Chaitin. Register Allocation and Spilling Via Graph Coloring . In *Proc. of the SIGPLAN Symp. On Compiler Construction*, pages 98-105, 1982.
5. K. Y. et al. A comparison of empirical and model-driven optimization. In *Proc. of PLDI*, pages 63-76, 2003.
6. M. P. et al. SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms. *Journal of HPCA*, 2002.
7. P. B. et al. Improvements to Graph Coloring Register Allocation. *ACM Trans. on Programming Languages and Systems*, 6(3):428-455, 1994.
8. C. Fischer and T. LeBlanc. *Crafting a Compiler*. Benjamin Cummings, 1987.
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 Trans. on Programming Languages and Systems*, 18(3):300-324, 1996.
11. J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *Proc. of the 2nd ICS*, 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. *Journal of the ACM*, 13(1):43-61, 1966.
14. W. Hsu, C. Fischer, and J. Goodman. On the Minimization of Load/Stores in Local Register Allocation. *IEEE TSE*, 15(10):1252-1260, 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. F. Luccio. A Comment on Index Register Allocation. *CACM*, 10(9):572-574, 1967.
17. R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT CS-97-366, University of Tenessee, 1997.
18. J. Xiong, J. Johnson, R. Johnson, and D. Padua. Spl: A language and a compiler for dsp algorithms. In *Proc. of PLDI*, pages 298-308, 2001.