

MASTER'S THESIS | LUND UNIVERSITY 2015

Performance Evaluation of ISO C restrict on the Power Architecture

Anton Botvalde, Andreas Larsson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-02



Performance Evaluation of ISO C restrict on the Power Architecture

Anton Botvalde

anton.botvalde@gmail.com

Andreas Larsson

andreas.lasse@gmail.com

January 21, 2015

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Examiner: Per Andersson, per.andersson@cs.lth.se

Abstract

The C99 standard for the C programming language introduced the new type qualifier `restrict` which acts as a hint for the compiler and the programmer that the specified pointer is not aliased by any other pointer if the pointed object can be modified. By using `restrict` on pointers the compiler may, if implemented and allowed, optimize code even further. This thesis investigates how well the C compilers GCC, Clang and XL C implements optimizations based on `restrict` on the Power architecture. By running a modified Livermore benchmark consisting of different loops that are suitable for `restrict` based optimizations we show that all three compilers are capable of doing `restrict` based optimizations. Furthermore we investigate loops using a pipeline simulator in order to understand the performance characteristics of the optimizations. We show that the performance for each loop vary by compiler, some loops have their running time significantly reduced while others, somewhat surprisingly, actually have their run time increased. Finally we provide some general suggestions for programmers and compiler developers on how to best use `restrict`.

Keywords: C programming language, `restrict`, optimizing compilers, Power architecture, pipeline simulation

Acknowledgements

We would like to thank our friends and family for their support during this master thesis.

We would especially also like to thank our supervisor, Jonas Skeppstedt, for all his support and encouragement. Jonas suggested evaluating ISO C restrict as a master thesis topic and we are grateful for the very interesting topic that he provided us with. Without Jonas' expertise in optimizing compilers and the Power architecture this thesis, simply put, would not be possible. Thank you for all the support, patience and commitment during during our work.

Another special thanks goes to John Griswell at IBM Austin for sharing his knowledge of ScrollPipeViewer and PowerPC pipeline analysis.

Contents

1	Introduction	7
1.1	Problem statement	8
1.2	Method	8
1.3	Related work	9
1.4	Contributions	9
1.5	Outline	9
2	Background	11
2.1	The alias problem	11
2.2	Restrict definition in C99	12
2.3	The PowerPC 970	16
2.4	GCC, Clang and XL C	19
2.5	The Livermore loops	20
2.6	Simulation tools	21
3	Results	23
3.1	General results	24
3.2	Loop 1	28
3.3	Loop 5	38
3.4	Loop 8	43
3.5	Loop 11	44
3.6	Loop 12	45
3.7	Loop 17	50
4	Discussion	53
4.1	Compiler discussion	53
4.2	Issues and problems encountered	55
5	Conclusions	57
5.1	Recommendations regarding restrict	57
5.2	Further work	58

Bibliography

59

Chapter 1

Introduction

In 1952 Grace Hopper produced the world's first functioning compiler for the A-0 System language. With it she forever changed the world of programming by making high level programming languages possible. Instead of writing code directly in assembly language, which is specific for a computer architecture, programmers could now write code in a different new language with higher levels of abstraction than assembly code. The compiler would then translate the code into a specific machines assembly code to produce the specified program. This made life easier for programmers because software development became easier: by programming in a higher level language one could produce code at a faster rate than directly working in assembly. By not having to worry too much about low level details such as which register to load and save a value from the programmer could focus on the actual program at hand: writing code that solved high level problems. This was of course not the only benefit of writing code in a language other than assembly - the porting of software to different architectures became much less of a problem. If a compiler existed for another platform chances were one could simply let the compiler translate the source code on the new machine and hopefully one had a working program, whereas if one had written the code in assembly the program would pretty much had to be rewritten to the new machine. While compilers enabled programmers to write programs faster it came with a cost. The produced code of early compilers were often worse than hand written assembly. Early compilers could not optimize code as well as an assembly programmer and some programmers did not want to write code in high level languages because of this reason. Due to the limited resources of historic computers it was often essential to have close to optimal code running. Nowadays the situation is very different and an optimizing compiler is capable of producing high quality optimized code which a human would have a hard time writing.

The C language was invented by Dennis Richie along with Ken Thompson during the period of 1969-1973 at Bell Labs. The authors saw a need for a small, fast, low level language that was portable and easy to implement a compiler for. Over 40 years later the language is still popular and frequently used when high performance and low level

programming is required. The first standard of the language came in 1989, called ANSI C, made by the American National Standards Institute. The standardization of the language was taken over by ISO, International Organization for Standardization, and since ANSI C there have been three language revisions made called C90, C99 and C11.

1.1 Problem statement

C99 introduced a new reserved word, `restrict`, as a type qualifier with the purpose of making C as fast as Fortran. C, unlike Fortran, allows parameters that can introduce aliasing which makes it harder to develop advanced optimization algorithms. Previously there was no way for the compiler or the programmer to know whether or not a parameter may be aliased. The `restrict` qualifier indicate to both that a parameter is alias free and thus the compiler can potentially optimize code even further if such optimizations are implemented. However because a program with `restrict` does not change the program output in any way, the compiler may simply choose to ignore the keyword entirely.

The purpose of this master thesis is to investigate whether or not the optimizing compilers GCC, Clang and XL C implement `restrict` based optimizations on the Power architecture. By investigating C code that is suitable for `restrict` based optimizations it is possible to evaluate if and how well these compilers work with the new type qualifier `restrict`. By studying the performance characteristics and assembly output of a program with and without the usage of `restrict` it is possible to get an understanding of whether or not `restrict` has any actual impact on the code and performance.

1.2 Method

Due to the open nature of the thesis it was initially decided that in order to best investigate the problem stated in the previous section 1.1 the thesis work would be divided into three major phases:

- Study phase.
- Benchmark phase.
- Analysis phase.

During the first phase a lot of different literature was read with the aim at gaining in depth knowledge of `restrict`. Resources such as the official C99 standard were studied in order to fully grasp what `restrict` really is and what it tries to solve. Various small `restrict` related coding experiments were written initially on both the Power platform and x86 with different compilers. For the x86 platform code was written for both a normal desktop PC and a MacBook Pro laptop (both using x86 CPUs). Eventually it was decided to limit the scope of the thesis to only focus on the Power architecture and the three compilers GCC, Clang and XL C. The said compilers were chosen due to their popularity on the Power platform, the fact that they support the C99 standard and finally because the authors had them available. The Power architecture was chosen mainly for three reasons - simplicity, available simulation tools and finally available knowledge. Since much of the thesis work

would consist of studying a lot of assembly code, the Power instruction set was considered more suitable to work with than the Intel x86 architecture. In order to better understand the performance of the assembly code, two simulators provided by Jonas Skeppstedt and IBM would be heavily used and it is uncertain if there are any equivalent for x86 for free. Finally, Jonas Skeppstedt, the supervisor of this thesis, has a very long history with Power and posses a very deep knowledge of the architecture which was deemed important for the success of the thesis. During this phase related previous work was also researched.

Initially at the thesis start it was thought that during the second phase a custom tailored made benchmark to evaluate restrict would be written from scratch. However during the second phase it became clear that using an already established benchmark would be more preferable. An old Fortran benchmark called Livermore loops was discovered to be an excellent restrict benchmark which had already been ported to C. Due to the port being old various updates to the C code were made in order to better suit the benchmark needs of this thesis.

In the final stage of the thesis a lot of in depth analysis of the performance data and assembly output from the benchmark were made using a pipeline simulator. During this phase the majority of this report was written as well.

1.3 Related work

A paper by Mock [8] states that programmer specified alias is bad because of the risk of introducing errors as well as meager performance gains. The paper describes techniques used to detect alias statically at compile time and at run time. With these analysis techniques Mock runs a series of benchmarks with alias optimizations to conclude that there is little performance gains when using restrict. However since aliasing is an undecidable problem [9], static alias analysis can only do so much, and even with advanced algorithms there will always be cases where algorithms can not rule out the possibility of aliasing. Doing alias analysis checks at run time will always have overhead involved that can hinder any sort of performance gain. As seen in section 3.1, even when a compiler does perform alias analysis on its own there are merits in having programmers specifying restrict to indicate that a pointer is alias free.

1.4 Contributions

The thesis work, consisting mainly of the study of restrict performance impact and report writing, has been equally distributed and contributed by both authors.

1.5 Outline

Chapter 2 provides a thorough theoretical background needed to fully understand the data presented in Chapter 3. Section 3.1 presents the performance results of the evaluated C code as well as in depth analysis of the results. In Chapter 4 we provide a discussion about the compiler results and issues encountered during the thesis. Lastly in Chapter 5 a set of conclusions and possible future work are presented.

Chapter 2

Background

2.1 The alias problem

The alias problem in C refers to the situation when for instance two pointers point to the same object in memory. Consider the code presented in Listing 2.1. At first glance, it might look like the compiler can rearrange line 3 and 4 or load `b[i]` at the same time as `d[i]`. But can it really do this? Because of the possibility of alias being present it might be the case that `a` and `d` actually refer to the same object in memory. Rearranging these lines would lead to an incorrect value being written to `c` which would be a serious error, thus the compiler can not rearrange the order of the execution without doing any pointer analysis first.

```
1 void foo1(int *a, int *b, int *c, int *d) {
2     for (int i = 0; i < 5; i++) {
3         a[i] = b[i] + i;
4         c[i] = d[i];
5     }
6 }
```

Listing 2.1: C pointer alias example.

Next, consider instead the code shown in Listing 2.2.

```
1 void foo2(int *a, int *b, int *c) {
2     for (int i = 0; i < 5; i++) {
3         a[i] = b[i] + b[i + 1];
4         c[i] = a[i] + i;
5     }
6 }
```

Listing 2.2: Scalar replacement of array references example.

In 1990 Carr et al [5] introduced an optimization for compilers that enabled scalar replacement of array references. This optimization would allow a compiler to potentially remove a memory load when working with arrays that have their value read several times. By saving the value into a register the compiler can replace a slow memory load with a fast register read (in Listing 2.2 this would be `b[i + 1]`). However, in C, a compiler has to be restrictive when performing optimizations of this kind due to the possibility of aliasing. Both `a` and `c` can potentially alias `b` and thus the compiler can not save the `b[i + 1]` value into a register because the assignments to `a[i]` and `c[i]` might have overwritten the value of `b[i + 1]`. The compiler is thus forced to load the value from memory each iteration of the loop.

```
1 void foo3(void) {
2     int a[5];
3     int *b = a;
4     int *c = b;
5     int *d = malloc(sizeof(int) * 5);
6     int *e = d;
7
8     for (int i = 0; i < 5; i++) {
9         a[i] = i;
10    }
11
12    for (int i = 0; i < 5; i++) {
13        *b++ = i;
14    }
15
16    free(d);
17 }
```

Listing 2.3: Named and unnamed objects in C.

Finally, consider the code in Listing 2.3. In this example `a` is a stack allocated named object. It can be accessed either through its own name, `a[]`, or through the pointer `b` and its copy, `c`, by dereferencing one of them. The pointer `d` points to an unnamed object which can be accessed with either `d` or `e`. Pointer analysis becomes complicated due to the presence of copies, and in real world situations copies are frequently present in code when passing around pointers through various functions. Analysing named objects such as `a[i]` is often easier for the compiler due to their fixed location in memory while it is more difficult for the compiler to perform pointer analysis when accessing an object through pointers, such as `*b++`. Doing static pointer analysis of large programs is especially difficult and will often not be able to rule out the presence of alias. On the other hand, analysis at run time involves overhead costs to check whether or not two pointers might overlap each other (i.e if they are disjoint).

2.2 Restrict definition in C99

Now that the alias problem has been explained in detail, focus is shifted to the exact meaning of `restrict` in C. As mentioned previously in section 1.1 `restrict` is one of the new addi-

tions to C in the C99 standard. With it compiler writers can implement new optimizations which previously were hard to write due to the possibility of alias being present. The following is an excerpt from section 6.7.3.1 of the C99 standard [7] which formally describes the definition of restrict:

- 1 Let **D** be a declaration of an ordinary identifier that provides a means of designating an object **P** as a restrict-qualified pointer to type **T**.

This simply denotes that **D** is a declaration such as `int * restrict P`; (where the type **T** in this case would be `int`).

- 2 If **D** appears inside a block and does not have storage class **extern**, let **B** denote the block. If **D** appears in the list of parameter declarations of a function definition, let **B** denote the associated block. Otherwise, let **B** denote the block of **main** (or the block of whatever function is called at program startup in a freestanding environment).

Paragraph 2 explains which block, **B**, the declaration **D** will belong to. It will belong to the block it is declared in if it does not have the **extern** storage class specifier. If it is part of a function parameter it will belong to the function's associated block, otherwise it will be a part of the **main** block.

- 3 In what follows, a pointer expression **E** is said to be *based* on object **P** if (at some sequence point in the execution of **B** prior to the evaluation of **E**) modifying **P** to point to a copy of the array object into which it formerly pointed would change the value of **E**). Note that "based" is defined only for expressions with pointer types.

The following note is presented on the same page paragraph 3 in the C99 standard - "In other words, **E** depends on the value of **P** itself rather than on the value of an object referenced indirectly through **P**. For example, if identifier **p** has type (`int **restrict`), then the pointer expressions **p** and **p+1** are based on the restricted pointer object designated by **p**, but the pointer expressions ***p** and **p[1]** are not."

Paragraph 3 might seem somewhat cryptic and unclear at first glance but a small example will help explaining the situation. Consider the code presented in Listing 2.4.

```

1 typedef struct {
2     int *data;
3 } example_t;
4
5 void foo4(example_t * restrict a) {
6     example_t *temp = malloc(sizeof(example_t));
7     memcpy(temp, a, sizeof(example_t));
8     a = temp;
9
10     /* pointer expressions with a are done here */
11 }

```

Listing 2.4: C99 standard paragraph 3 example.

In the code above a copy of the struct is made and then the copy is assigned to the pointer again. If a pointer expression such as `a` or `a+1` is made, the value of that expression will

change because of the copy that was made. Since the value of the said expression is an address, it will have changed now that the pointer points to a new memory location. When the value is changed when we make an assignment, a pointer expression is said to be based on the object. But the value of for instance `a->data`, will not have changed even when a copy was made, it stays the same.

4 During each execution of **B**, let **L** be any lvalue that has **&L** based on **P**. If **L** is used to access the value of the object **X** that it designates, and **X** is also modified (by any means), then the following requirements apply: **T** shall not be const-qualified. Every other lvalue used to access the value of **X** shall also have its address based on **P**. Every access that modifies **X** shall be considered also to modify **P**, for the purposes of this subclause. If **P** is assigned the value of a pointer expression **E** that is based on another restricted pointer object **P2**, associated with block **B2**, then either the execution of **B2** shall begin before the execution of **B**, or the execution of **B2** shall end prior to the assignment. If these requirements are not met, then the behavior is undefined.

Consider the following code: `a[i] = b[i] + c[i];`. What section 4 simply says is that `a[i]` is the only way to access that object and it may not be const.

5 Here an execution of **B** means that portion of the execution of the program that would correspond to the lifetime of an object with scalar type and automatic storage duration associated with **B**.

Paragraph 5 is rather straightforward, see 2.5. The variable `a` lives inside the function `func` and is destroyed when the function returns. The execution of the block associated with `foo5` means the execution of all code inside `foo5`.

```
1 void foo5 () {
2     int a;
3     /* ... */
4 }
```

Listing 2.5: C99 standard paragraph 5 example.

6 A translator is free to ignore any or all aliasing implications of uses of **restrict**.

This simply means that a compiler is free to completely ignore the usage of `restrict`. Just like the keyword `register`, `restrict` is just a hint to the compiler that a pointer is alias free. Whether or not the compiler does anything with the hint is up to the compiler writers. This means that a programs behavior should be exactly same with or without `restrict` (e.g. the output should not change). Note that the `register` keyword is not entirely just a hint, the following code is invalid : `register int a; int *b = &a;`.

Because of the somewhat complicated formal definition of `restrict` it might be hard to actually understand what it means and how it is applied so a couple of examples are in order. The following examples are all presented in the C99 standard to illustrate how `restrict` works.

```

1 int * restrict a;
2 int * restrict b;
3 extern int c[];

```

Listing 2.6: C99 standard restrict example 1.

In Listing 2.6 all declarations have file scope and they promise that an object which is being accessed through either a, b or c and is modified, then the other two will never access that object.

```

1 void f(int n, int * restrict p, int * restrict q) {
2     while (n-- > 0)
3         *p++ = *q++;
4 }

```

Listing 2.7: C99 standard restrict example 2.

Listing 2.7 shows a program in which the function parameters are declared with restrict. This promises the compiler that none of the objects accessed by q will be accessed by p. Calling the function f with parameters that do overlap results in undefined behavior.

```

1 void g(void) {
2     extern int d[100];
3     f(50, d + 50, d); // valid
4     f(50, d + 1, d); // undefined behavior
5 }

```

Listing 2.8: C99 standard restrict example 3.

The code in 2.8 calls the function presented in Listing 2.7. The first function call is defined since even though the parameters p and q will refer to the same object d, they will never modify each others part of the array since they are disjoint. In the second call however p and q will overlap and hence the behavior is undefined.

Consider now the following code presented in Listing 2.9.

```

1 #define SIZE 10
2 void foo5(int * restrict a, int * restrict b,
3     int * restrict c) {
4     for (int i = 0; i < SIZE; i++) {
5         a[i] = b[i] + c[i];
6     }
7 }
8
9 void foo6(void) {
10     extern int a[SIZE];
11     extern int b[SIZE];
12     extern int c[SIZE];
13
14     foo5(a, b, c); // valid
15     foo5(a, b, b); // valid
16 }

```

Listing 2.9: C99 restrict example 1.

Making a call to function `foo5` such as `foo5(a, b, b)` is perfectly legal since `b` in this case will never be modified. In order to make the previous function call one must know how the function works. If only a function prototype is presented and the programmer has no access to the source code, then a programmer can not know that such a call is in fact safe. A restrict pointer can also be a member of a struct. With it, the specified object can only be accessed through the restricted struct member pointer.

One of the real dangers with `restrict` is that it is up to the programmer to make sure that no actual alias is present in the code. Consider for instance the function call with undefined behavior in Listing 2.8. If the code is part of a program that is written in a compiler that ignores the `restrict` keyword nothing bad will happen. However, if say for instance 10 years later the program is compiled with another compiler that does in fact implement `restrict`, then the code is no longer valid and a serious bug has been introduced. This bug might be extremely hard to find and fix, especially in a large code base. Dennis Ritchie himself described `restrict` as "timebombs that are sure to explode in people's faces" [10].

2.3 The PowerPC 970

All performance related results presented in this thesis are based on an Apple Power Mac G5 Quad 2.5 GHz with 6 GB RAM which has been provided by Jonas Skeppstedt and the Department of Computer Science at LTH.

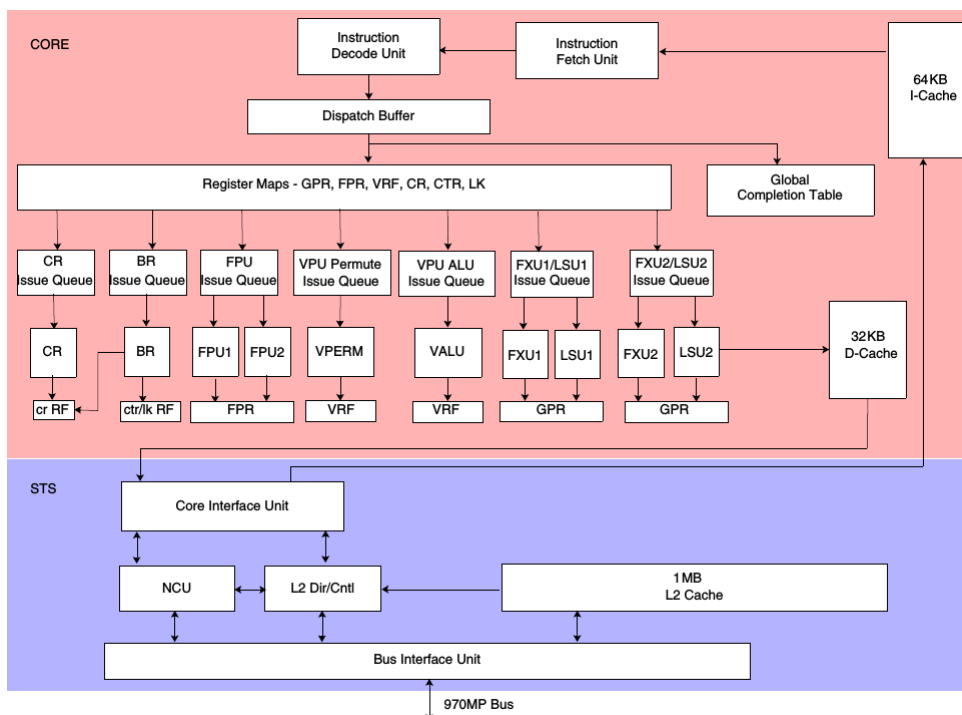


Figure 2.1: PowerPC 970 overview [6].

In 1991 the AIM (Apple, IBM, Motorola) alliance was formed and out of it the PowerPC architecture was born. It is a RISC instruction set architecture which was then almost identical to Power. PowerPC 970 was introduced in 2002 by IBM and it was released in 2003 as PowerPC G5 in Apple machines. The PowerPC 970 is a superscalar processor.

A superscalar processor can execute multiple instructions in parallel and to achieve this the processor has multiple execution units. For example when executing two integer instructions the processor can send each instruction to two different ALUs. It is a possibility that the two instructions are not able to execute in parallel. One reason for this might be that the first instruction writes to a register which the other instruction reads from. The processor must also be able to fetch multiple instructions at the same time. This adds some trouble when dealing with branch instructions, if the first instruction is a branch and proceeds to jump to some other code, the second instruction must be discarded. Although instructions are executed out of order they need to be completed in order and this is accomplished by a reorder buffer. The reorder buffer maintains an ordered list of instructions which are added when they are dispatched and later removed when completed. Note that even though a superscalar processor is a parallel machine it is important that it still looks like a sequential machine from the programmer's perspective.

An overview of PPC 970 can be seen in Figure 2.1. The instruction fetch unit can fetch up to 8 instructions per cycle from the I-Cache [11]. Once an instruction has been fetched it is sent to the decode stage. Instructions are translated into internal instructions called "IOPs". Some instructions are also cracked down into multiple smaller IOPs, a cracked instruction is split into two IOPs and a microcoded instruction is split into more than two IOPs. This allows the processor to schedule code more freely. The PPC 970 assembles instructions into dispatch groups and each dispatch group consists of up to 5 IOPs. The decoded IOPs are moved in order into a dispatch group and the group is dispatched once it is full. After a group has been dispatched, the individual IOPs enter their respective issue queues. Now the IOPs can be issued out of order and after execution they need to be placed in the right order with the help of the GCT (Group Completion table). The GCT is the PPC 970's equivalent of a reorder buffer with the difference being that a reorder buffer keeps track of individual instructions and the GCT keeps tracks of dispatch groups. So instead of keeping track of 100 in-flight instructions, PPC 970 only needs to keep track of 20 active groups which reduces some overhead. If the GCT is full no new groups can be dispatched but there are four dispatch buffers available which can hold four groups.

In order to assemble and dispatch a group there are some rules which must be followed. Assembling: A group must be populated with IOPs in program order. The last IOP must be a branch instruction otherwise the group can only have four slots. The CRU can only go into slot one and two. A cracked instruction must have both of its IOPs in the same group. A microcoded instruction always start a new group. Dispatch: The GCT has 20 entries and if it happens to be full then no group can be dispatched. If an instructions issue queue is full the group must wait. There must be register rename resources available [11].

Execution Units: the PPC 970 has two integer units which execute almost all integer instructions. They are slightly specialized in that one IU performs integer divisions and the other performs special purpose register operations. There are many different SPR instructions and a few of them are used for manipulating the LR (Link Register) and CTR (count register). The LR is used as a branch target address or holds a return address and the CTR is used for loop counts and branching.

The CRU handles operations related to the CR (Condition Register). Most of the arithmetic operations has the option of setting various flags on the CR to store information about their outcome (i.e the result is positive, negative or zero). The CRU is the unit that executes the instructions which performs logical operations on the 8 condition register fields.

On older PPC implementations these instructions were executed on the IU units. The two LSU (Load/Store Units) executes all of the load and store instructions and has dedicated hardware for address generation. The two FPUs are identical and executes floating point instructions.

Issue queues: as described earlier the execution units are fed instructions from the issue queues. However the issue queues are a bit more complicated than what can be seen in 2.1. The FPU issue queue consists of two 10-entry interleaved queues. Which queue an instruction enters depends on where it resides in the dispatch group. Slot 0 and slot 3 is attached to one queue and slot 1 and 2 to the other. Once an instructions all sources are available it is then issued to the connected execution unit. The FXU and LSU issue queues works in the same manner as the FPUs. One FXU and LSU pair shares one 18-entry issue queue. Slot 0 and 3 is connected to the first pair and slot 1 and 2 to the second pair. The CRU issue queue has 10 entries and is fed instructions from slot 0 and 1. The BR issue queue has 12 entries and is fed from slot 4.

The dispatch grouping scheme leads to some interesting performance implications with regards to code scheduling. For example whether FPU1 or FPU2 is used for a floating point IOP depends on where the IOP is placed in a group. It may happen that one FPU is underutilized. This simply means that code that does not take this into account will miss out on some performance improvements.

The pipeline stages for most instructions are [6]:

- 16 stages for fixed-point register-to-register operations.
- 18 stages for most load-and-store operations.
- 21 stages for most floating-point operations.
- 19 stages for (VALU) fixed-point, 22 stages for complex-fixed, and 25 stages for floating-point operations.
- 19 stages for VPU permute operations.

Branch prediction is very important for the PPC 970 since it has a very wide and deep pipeline [11]. If a miss prediction occurs then the whole pipeline has to be flushed and a costly pipeline stall will occur. Out of the 8 instructions fetched from the IFU, the branch prediction unit scans those and chooses up to two branches to predict. It uses two branching schemes for each branch and keeps track of which one is the most successful at predicting a specific branch.

The following is a short list of a few PowerPC assembly instructions:

li r3, 5 loads the value 5 into r3

lfd f12,8(r10) Loads a doubleword specified by the address in register r10 with offset 8 into floating point register f12.

lfd f0,8(r10) This a cracked instruction, it is split internally into two instructions. It loads the address specified by r10 with offset 8 into f0 and then updates the address in r10 with the new address.

stfdu f0,8(r8) Also a cracked instruction. It stores the content in floating point register f0 to the address specified in r8 with offset 8 and then update r8 with the new address.

add r3,r26,r14 add the content of r26 with the content in r14 and place the result in r3.

fmadd f12,f9,f12,f10 Fused floating point add and multiply instruction. It multiplies f9 with f12 and adds the result of the multiplication with f10 and then stores the result of the addition in f12.

fmr f31,f12 Copy the content of the floating point register f12 to f31.

cmpw cr7,r31,r15 The contents of r31 and r15 is compared and sets the bits in the condition field cr7 appropriately.

bne cr7,100026e0 jump to address 100026e0 if the condition register cr7 is set to false. Note that the instructions uses the first operand as the destination register.

2.4 GCC, Clang and XL C

There are many available compilers for the PowerPC architecture, however this thesis focus only on three of them - GCC, Clang and XL C. All of them are well known names in the software industry and are optimizing compilers that employ various optimization algorithms based on SSA form, such as dead code elimination and global value numbering just to name a few. The following compiler versions have been used in this thesis - GCC 4.8.2, Clang 3.3 and XL C 11.1, all running on Fedora 19, Linux.

The GNU Compiler Collection, or simply GCC, is an open source under the GPL-license compiler platform which is a part of the GNU Project [2]. The first official beta was released March of 1987 and has been in development ever since. The aim of GCC is to improve compilers used by the GNU systems. It provides several front ends including but not limited to C, C++, Java and Fortran. The C front end is more commonly known as gcc, and when GCC is mentioned in this thesis often what is really meant is the C front end gcc. A variety of different platforms are supported, including PowerPC, x86 and ARM. Due to its popularity it is considered to be the de facto standard of open source compilers and has been adopted as the standard compiler for the Unix-like operating system Linux and is usually included with most distributions of Linux. To compile the fictional C file main.c in Linux one can simply type "gcc main.c -std=c99" in the terminal. This will make GCC compile the C file with the C99 standard enabled. The version of gcc used in this thesis was 4.8.2.

Clang is an open source compiler front end under the BSD-license which supports the C language family [1]. Unlike GCC, Clang use LLVM for its the back end which is an open source compiler library that enable Clang to have many of the compiler optimizations present in LLVM. By using LLVM, the Clang front end can produce LLVM IR code, a sort of intermediate code, which then the LLVM optimizer can optimize with the usual optimization techniques. The optimizer then pass the tuned IR code to the back end which produces machine code for the specific processor architecture. This means that Clang supports all platforms that LLVM supports, which includes the PowerPC. By default Clang

already perform some pointer analysis which needs to be taken into account when evaluating restrict optimizations. To compile the fictional C file `main.c` in Linux one can simply type `"clang main.c -std=c99"` in the terminal. This will make Clang compile the C file with the C99 standard enabled.

XL C/C++ is an advanced optimizing compiler developed by IBM that supports C and the C++ language [4]. But since this thesis only covers the C language, XL C/C++ shall be referred to as XL C only. Unlike the other two compilers evaluated in this thesis, XL C is the only one that is proprietary licensed and is not available as open source. As of November of 2014 IBM charges at least \$1,420 per user license which includes a 12-month long support. A 60-day free trial is also available for anyone interested to evaluate the compiler. XL C supports the PowerPC family and work with several operating systems such as Linux. Due to the closed source nature of XL C it is difficult to know what and how the compiler optimizes code. To compile the fictional C file `main.c` in Linux one can simply type `"xlc main.c -qlanglvl=stdc99"` in the terminal. This will make XL C compile the C file with the C99 standard enabled.

2.5 The Livermore loops

In order to evaluate how well the three compilers are able to optimize code with restrict enabled a suitable benchmark is needed. In the beginning of the thesis various C code snippets were produced to investigate restrict based optimizations. While it is somewhat trivial to come up with some specific piece of code that can show performance gains with restrict, it is better to try to find some already established benchmark which can be evaluated with and without restrict.

The Livermore loops (also known as The Livermore Fortran Kernels) were developed in the 1970's and 1980's by the Lawrence Livermore National Laboratory. It is a Fortran benchmark consisting of a set of 24 kernels (functions) taken from real world scientific source code and was designed to measure floating point performance of computers and their compilers [3]. The loops are suitable as a benchmark for compilers since they provide many different optimization opportunities. Thanks to the usefulness of the benchmark it has been ported to various other languages besides Fortran. The loops are a perfect fit as a restrict benchmark since they all use different arrays which may alias. Consider the loop in Listing 2.10. The loop is the third Livermore loop and represents a simple inner product (dot product).

```
1 for (l=1; l <= loop; l++) {  
2     q = 0.0;  
3     for (k=0; k < n; k++) {  
4         q += z[k] * x[k];  
5     }  
6 }
```

Listing 2.10: Livermore loop 3 translated to C.

From a restrict point of view this loop is not particular interesting since even without the presence of alias no further optimization can be done. Out of the 24 available loops only 14 were used in this thesis since they were the only ones which were deemed to have any restrict based optimization opportunities.

2.6 Simulation tools

Three simulation tools have been used in this thesis - ScrollPipeViewer, run_timer and ASIM.

ASIM is a program written by Jonas Skeppstedt which he started developing 1997 (development is still ongoing). It is written in C and takes another runnable program as input. It then interprets one instruction at a time and follows all instruction and memory accesses done by the specified program. By doing this it can produce a .tt6e trace file (it simply contains all the memory accesses) which serves as input for the simulator run_timer. As of late 2014 ASIM currently only supports programs which use 32-bit instructions.

Run_timer is series of cycle accurate CPU simulators created by IBM. It supports a broad range of Power products, including Power 4, Power5, Power 6, Power 7 and PowerPC 970. With run_timer it is possible to accurately simulate the performance of a particular program on a given Power implementation. The run_timer programs are simply the hardware platform modeled in software. With them it is possible to simulate the hardware and the performance characteristics of the processor before the hardware actually exists. Because it is very expensive and difficult to produce a functioning prototype of a processor it is common to create a simulator of the processor first. The output produced by the simulator is a .pipe file based on the given .tt6e input. The .pipe file contains the entire pipeline simulated which can then be used as input to ScrollPipeViewer.

ScrollPipeViewer is a user interface program for displaying the simulated pipeline execution. As can be seen from Figure 2.2 in area 1, it displays the clock cycles at the x-axis and the IOPs at the y-axis. This makes it possible to see the pipeline stages of instructions for each clock cycles. Area 2 displays the IOP id which starts from 1 and increases for each executed IOP. Area 3 displays the IOP mnemonic which is the name of the IOP and the register/values it uses. Area 4 displays the instruction address and area 5 displays the data address for load and stores. With all this information available ScrollPipeViewer makes it easy to study the performance of a given program, without a pipeline viewer it is otherwise much more difficult to understand why a program for instance might be slower than expected.

The following list explains what a couple of the symbols mean in ScrollPipeViewer:

- B** Branch prediction occurred.
- C** Instruction complete.
- D** Instruction in one stage of decode.
- E** Instruction in execute cycle.
- F** Fetch initiated.
- f** Instruction finished.
- f** Dispatch hold due to issue queue is full.
- j** Load rejected due to load hit store, data not ready.
- I** Instruction is issued.

M Instruction is dispatched.

s Cannot issue, source is not ready.

u Cannot issue, unit is not free.

V Fetch data is back.

	2	3	4	5
	Top Id	Mnemonic	Inst Addr	Data Addr
FV.B...DD..DD..DMIIEf.....C.....	158953	addi R4,R4,8	100004d4	
FV.B...DD..DD..DMI6.f.....C.....	158954	bc 25,0,-40	100004d8	
...FV.B...D..DDD..DMI2E..f.....C.....	158955	lfd F0,8(R3)	100004b0	ffe3c9c8
...FV.B...D..DDD..DM.s..sI5EEEEf.....C.....	158956	fmul F4,F29,F0	100004b4	
...FV.B...D..DDD..DMsI3E..f.....CD...S.....	158957	stfd F5,-8(R6)	100004b8	ffe3c6e0
...FV.B...D..DDD..DM.s..s.s..sss.I5...f.C.....	158958	stfd F5,-8(R6)	100004b8	
...FV.B...D..DDD..DMI0Ef.....f.C.....	158959	addi R3,R3,8	100004bc	
...FV.B...DD..DD..MI2E..f.....C.....	158960	lfd F3,0(R4)	100004c0	ffe3c828
...FV.B...DDD..DD..M.s..s.sI5EEEEf.....C.....	158961	fmadd F5,F3,F1,F30	100004c4	
...FV.B...DDD..DD..MIIEf.....C.....	158962	addi R6,R6,8	100004c8	
...FV.B...DDD..DD..M.s.s.s.sI4EEEEf.....C.....	158963	fmadd F1,F31,F2,F4	100004cc	
...FV.B...DD..DD..DMsI4EEEEf.....C.....	158964	fmr F2,F0	100004d0	
...FV.B...DD..DD..DMIIEf.....C.....	158965	addi R4,R4,8	100004d4	
...FV.B...DD..DD..DMI6.f.....C.....	158966	bc 25,0,-40	100004d8	
...FV.B...D..DDD..DMI2E..f.....C.....	158967	lfd F0,8(R3)	100004b0	ffe3c9d0
...FV.B...D..DDD..DM.s..sI5EEEEf.....C.....	158968	fmul F4,F29,F0	100004b4	
...FV.B...D..DDD..DMsI3E..f.....CD...S.....	158969	stfd F5,-8(R6)	100004b8	
...FV.B...D..DDD..DM.s..s.s..sss.I5...f.C.....	158970	stfd F5,-8(R6)	100004b8	ffe3c6e8
...FV.B...D..DDD..DMI0Ef.....f.C.....	158971	addi R3,R3,8	100004bc	
...FV.B...DDD..DD..MI2E..f.....C.....	158972	lfd F3,0(R4)	100004c0	ffe3c830
...FV.B...DDD..DD..M.s..s.sI5EEEEf.....C.....	158973	fmadd F5,F3,F1,F30	100004c4	
...FV.B...DDD..DD..MIIEf.....C.....	158974	addi R6,R6,8	100004c8	
...FV.B...DDD..DD..M.s.s.s.sI4EEEEf.....C.....	158975	fmadd F1,F31,F2,F4	100004cc	
...FV.B...DD..DD..DMsI4EEEEf.....C.....	158976	fmr F2,F0	100004d0	
...FV.B...DD..DD..DMIIEf.....C.....	158977	addi R4,R4,8	100004d4	
...FV.B...DD..DD..DMI6.f.....C.....	158978	bc 25,0,-40	100004d8	
...FV.B...D..DDD..DMI2E..f.....C.....	158979	lfd F0,8(R3)	100004b0	ffe3c9d8
...FV.B...D..DDD..DM.s..sI5EEEEf.....C.....	158980	fmul F4,F29,F0	100004b4	
...FV.B...D..DDD..DMsI3E..f.....CD...S.....	158981	stfd F5,-8(R6)	100004b8	
...FV.B...D..DDD..DM.s..s.s..sss.I5...f.C.....	158982	stfd F5,-8(R6)	100004b8	ffe3c6f0
...FV.B...D..DDD..DMI0Ef.....f.C.....	158983	addi R3,R3,8	100004bc	

Figure 2.2: ScrollPipeViewer overview.

Chapter 3

Results

The results presented in this Chapter are all from a modified Livermore loop benchmark which have been running on the machine described in section 2.3. The benchmark is a selection of 14 Livermore loops out of the original 24. The code used is based on the original translated Livermore benchmark to C with some adjustments made to better suit this thesis. For instance, in the original benchmark there is a big global struct containing all the arrays in the benchmark and this has been removed. Instead, they have been split into separate local arrays in the main function. This make it harder for the compiler to analyze the code, while if all arrays are located in a giant struct they will not alias and the compiler can make more optimizations easier. The original code is just one .c file containing all the code, while the benchmark in this thesis is split into three files, which are compiled separately and linked together to make it once again harder for the compiler to analyze the code. Another difference is that the original benchmark only used floating point types. In our version both integer and floating point types are evaluated since they use different hardware units and there are also some specialized instructions for floating point types. The types of the functions parameters and whether or not restrict is allowed have been defined with a #define which is set by a compiler flag. This makes it easier to run the benchmark with different settings without having to change the code. The input data and loop iterations have also been increased compared with the original benchmark. To run the benchmark a couple of makefiles and scripts were written. These files allowed us to run the benchmark and then use ASIM and run_timer to further study the pipeline. There are in fact two benchmarks, one for GCC and Clang, and one for XL C. The only difference between these are that in order to make XL C run some of the loops an empty dummy function had to be added to some of the loops, otherwise XL C performed dead code elimination and never actually ran those loops. This adds a slight overhead cost to the XL C results which GCC and Clang does not have.

All the presented results are based on an average of ten runs for each loop and each loop runs a number of iterations internally which is set for each loop (i.e the outer loop counter for each loop might differ). In order to be able to use the simulators a lower loop

iteration count have been used because ScrollPipeViewer only seem to be able to open files which are at most around 120 MB large (the default settings for the benchmark generate .pipe files that are very big and take a long time to produce by the simulators). Generally speaking one often only need to study a couple iterations of a loop in order to understand the performance characteristics of it. Due to unknown reasons we were unable to make Clang perform loop unrolling, even with aggressive loop unrolling flags set. Because of this there are no unrolled results available for Clang. All presented results are based on the 32-bit instruction set, O3 optimization flag, no inlining allowed and static linkage unless stated otherwise.

3.1 General results

The following tables have been produced by compiling the data generated by the benchmark. All the data has been normalized against some other reference data. Which data it has been normalized against is specified in the caption text in each table. For instance, Figure 3.7 shows the execution time of each loop with double restrict (i.e the function's pointer parameters are double * restrict) for XL C and Clang against GCC's result. Figures 3.1-3.6 are normalized against the variant without restrict and no unrolling. The plotted data is the percentage change against the normalized data which is set as 100% in all the tables. Results under 100% means that the loop runs faster than what it is normalized against, while anything over 100% means it actually is slower.

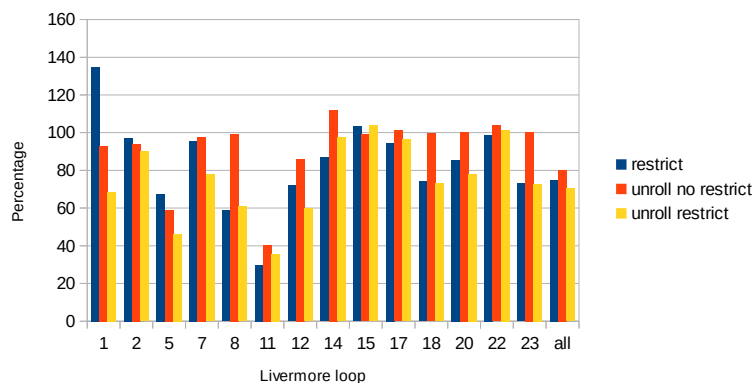


Figure 3.1: GCC's execution time normalized against double, no restrict.

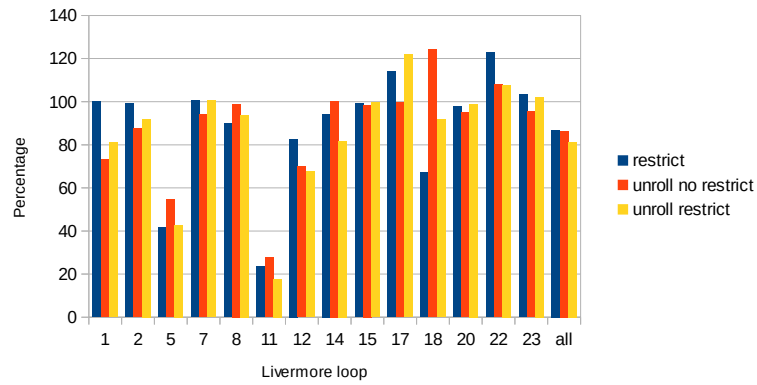


Figure 3.2: GCC's execution time normalized against int, no restrict.

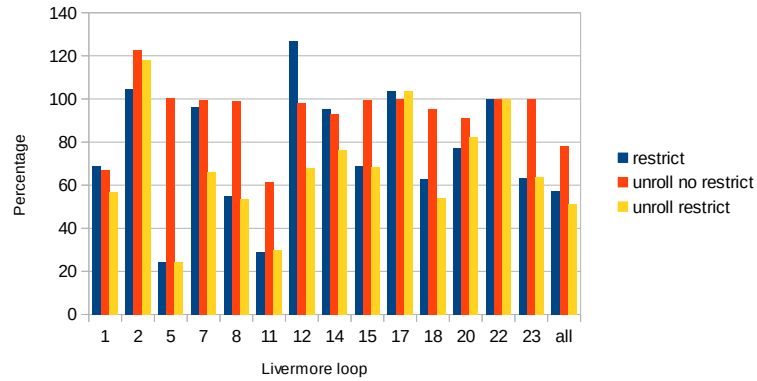


Figure 3.3: XL C's execution time normalized against double, no restrict.

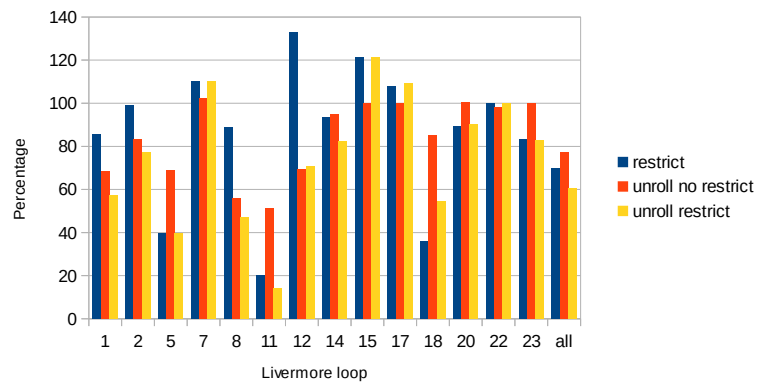


Figure 3.4: XL C's execution time normalized against int, no restrict.

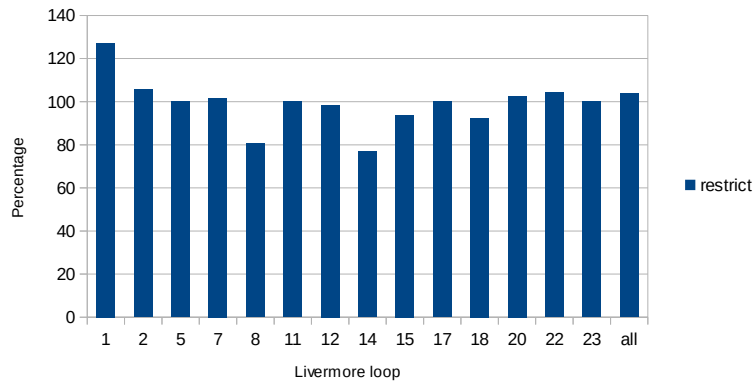


Figure 3.5: Clang's execution time normalized against double, no restrict.

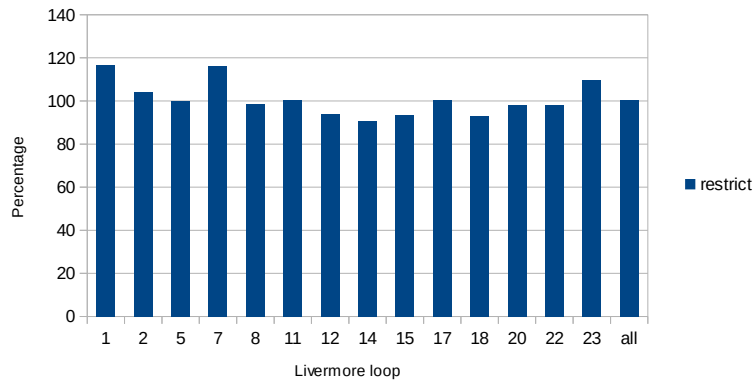


Figure 3.6: Clang's execution time normalized against int, no restrict.

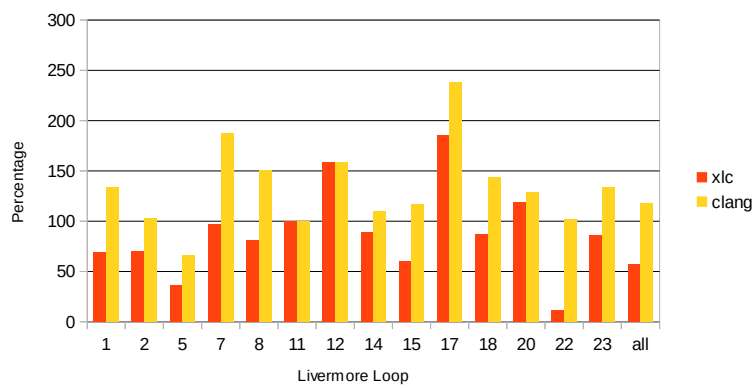


Figure 3.7: Execution time normalized against GCC (double with restrict).

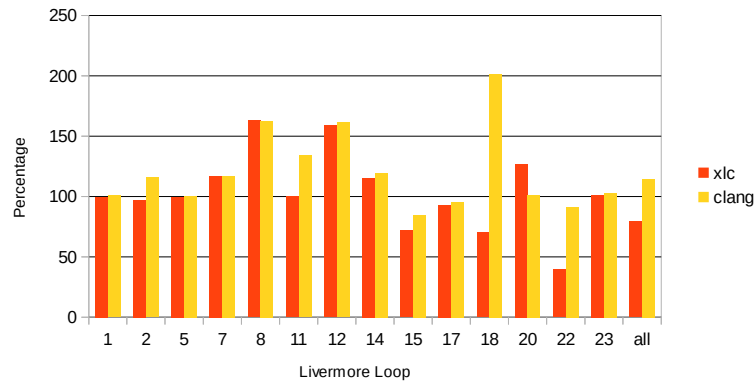


Figure 3.8: Execution time normalized against GCC (int with restrict).

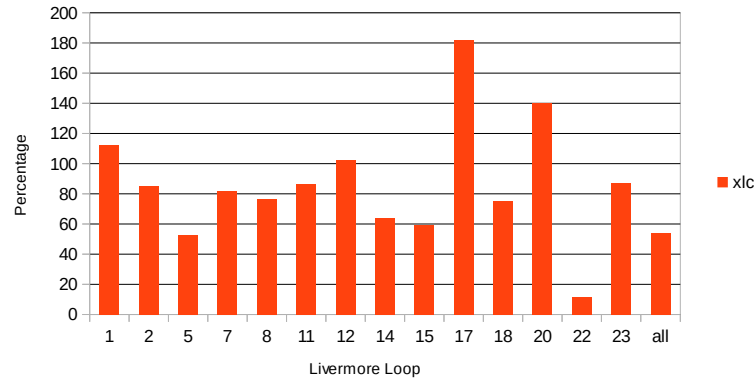


Figure 3.9: Execution time normalized against GCC (double, unrolling with restrict).

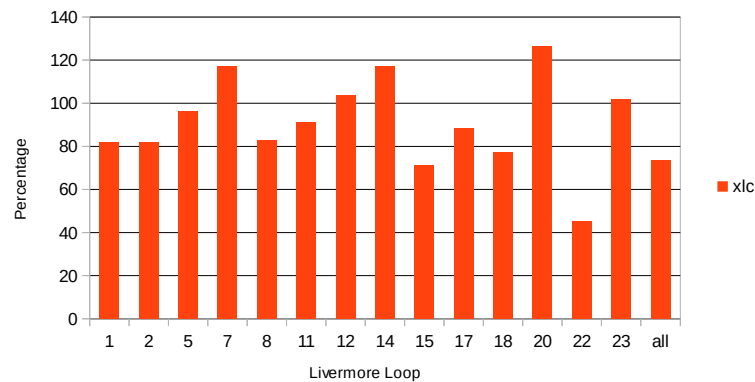


Figure 3.10: Execution time normalized against GCC (int, unrolling with restrict).

All the data in the previous tables are based on execution times. Because the loops in the benchmark take a variable amount of time (i.e the fastest might only take a second while the slowest might take over a minute to execute) it can be misleading to compare only execution times. For instance, if all the fast loops in the benchmark have worse performance with restrict enabled but one slow loop have an increase then the actual running time might be improved while the average relative performance gain is actually worse. Because of this looking at just the running time might lead to skew comparisons, especially between different compilers. To give a different perspective on the restrict performance of each compiler the table presented in Figure 3.11 show instead the total average execution time change relative to the compiler's non restrict type counterpart.

Type	Double			Int		
	Restrict	Unroll	Unroll restrict	Restrict	Unroll	Unroll restrict
GCC	16.31	8.15	24.14	11.73	12.35	14.32
XL C	23.18	5.21	31.04	13.71	15.78	24.50
Clang	1.16	-	-	-0.94	-	-

Figure 3.11: Average execution time change (in percent) relative no restrict. A positive value means the execution time was reduced while a negative value means the execution time increased.

We will now analyze a selected set of loops in more detail. The loops have been chosen to cover as many the different possible restrict optimization combined with the interesting performance characteristics as seen in the figures above.

3.2 Loop 1

Loop 1 is a hydrodynamics code fragment and the C code is shown in Listing 3.1. The two `#define` `TYPE` and `RESTRICT` are used to make it easy to switch between double and integer types and enabling or disabling the usage of restrict. How they are defined is shown in Listing 3.2. Looking at the C code it is clear that a compiler can save the value of `z[k+11]` in a register and use this value for the next loop iteration for the `z[k+10]` array access if `x` and `z` does not alias. But by default a compiler does not know this, however with restrict enabled we as programmers guarantee that this is the case. The compiler is then free to perform this optimization if it has been implemented by the compiler writers. Saving a value to a register in one iteration and reading it from the register in the next iteration in every loop iteration should result in a performance gain since it is much slower to read from memory (most of the time the value will likely be in the cache though). However looking at Figures 3.1, 3.3 and 3.5 in the previous section the execution time is, somewhat surprisingly, slower when using double restrict than without restrict.

The execution time of GCC and Clang increased by 34.83% and 27.17% respectively while the execution time of XL C is reduced by 31.30%.


```

1 void liver_loop1(TYPE*RESTRICT x, TYPE*RESTRICT y,
2                 TYPE*RESTRICT z, TYPE r,
3                 TYPE q, TYPE t, int n, int loop) {
4     for (int l = 0; l <= loop; l++ ) {
5         for (int k = 0; k < n; k++ ) {
6             x[k] = q + y[k]*( r*z[k+10] + t*z[k+11] );
7         }
8     }
9 }

```

Listing 3.1: Livermore loop 1 in C.

```

1 #ifdef RES
2 #define RESTRICT restrict
3 #else
4 #define RESTRICT
5 #endif
6 #ifdef TI
7 #define TYPE int
8 #else
9 #define TYPE double
10 #endif

```

Listing 3.2: #define for TYPE and RESTRICT used in the benchmark loops.

The assembly shown in Listing 3.3 corresponds to the inner loop in Listing 3.1. The lfd instruction on line 1 in Listing 3.3b has been replaced with a fmr on line 5 in Listing 3.3a which is in line with what can be expected by a restrict optimization. The assembly for

```

1 lfd    f0,8(r10)
2 lfdx  f10,r4,r9
3 fmul  f11,f0,f3
4 fmadd f11,f1,f12,f11
5 fmr   f12,f0
6 fmadd f0,f10,f11,f2
7 stfdx f0,r3,r9
8 addi  r9,r9,8
9 cmplw cr7,r9,r8
10 bne   cr7,10001e10

```

a: Restrict.

```

1 lfd    f12,0(r10)
2 lfdx  f11,r4,r9
3 lfd   f0,8(r10)
4 fmul  f0,f3,f0
5 fmadd f0,f12,f1,f0
6 fmadd f0,f11,f0,f2
7 stfdx f0,r3,r9
8 addi  r9,r9,8
9 cmplw cr7,r9,r8
10 bne   cr7,10001950

```

b: No restrict.

Listing 3.3: GCC loop 1 assembler (double).

Clang is shown in Listing 3.4. Clang has done the same optimization as GCC, i.e replaced the lfd on line 2 in 3.4b with a fmr on line 13 in 3.4a. It is worth remembering here that Clang perform alias analysis by default and it actually failed in this case since it did not replace the load instruction. The assembly for XL C is shown in Listing 3.5a and 3.5b and as seen XL C also replaced one lfd with a fmr but the code is better scheduled compared to the GCC and Clang versions. Comparing the instructions and which registers they write to in listing 3.3a, 3.4a and 3.5a it can be seen that there are less data dependencies in the XL C version. Studying the assembly code of GCC and Clang alone does however not explain why there is a decrease instead of an increase in performance.

```

1 lfd      f4,0(r11)
2 lfd      f6,0(r10)
3 fmul     f0,f0,f1
4 addi    r0,r12,8
5 addi    r10,r10,8
6 addi    r11,r11,8
7 fmul     f5,f4,f3
8 fadd     f0,f0,f5
9 fmul     f0,f6,f0
10 fadd     f0,f0,f2
11 stfd    f0,0(r12)
12 mr      r12,r0
13 fmr     f0,f4
14 bdnz    10000c98
    
```

a: Restrict.

```

1 lfd      f0,0(r10)
2 lfd      f4,-8(r10)
3 lfd      f5,0(r9)
4 addi    r12,r11,8
5 addi    r9,r9,8
6 addi    r10,r10,8
7 fmul     f0,f0,f3
8 fmul     f4,f4,f1
9 fadd     f0,f4,f0
10 fmul     f0,f5,f0
11 fadd     f0,f0,f2
12 stfd    f0,0(r11)
13 mr      r11,r12
14 bdnz    10000c94
    
```

b: No restrict.

Listing 3.4: Clang loop 1 assembler (double).

```

1 lfd      f0,8(r3)
2 fmul     f4,f29,f0
3 stfd    f5,-8(r6)
4 addi    r3,r3,8
5 lfd      f3,0(r4)
6 fmadd    f5,f3,f1,f30
7 addi    r6,r6,8
8 fmadd    f1,f31,f2,f4
9 fmr     f2,f0
10 addi    r4,r4,8
11 bdnz+   100004b0
    
```

a: Restrict.

```

1 stfd    f0,0(r3)
2 lfd      f1,8(r4)
3 fmul     f0,f29,f1
4 lfd      f1,0(r4)
5 fmadd    f0,f31,f1,f0
6 lfd      f2,8(r5)
7 fmadd    f0,f2,f0,f30
8 addi    r3,r3,8
9 addi    r4,r4,8
10 addi    r5,r5,8
11 bdnz+   10000500
    
```

b: No restrict.

Listing 3.5: XL C loop 1 assembler (double).

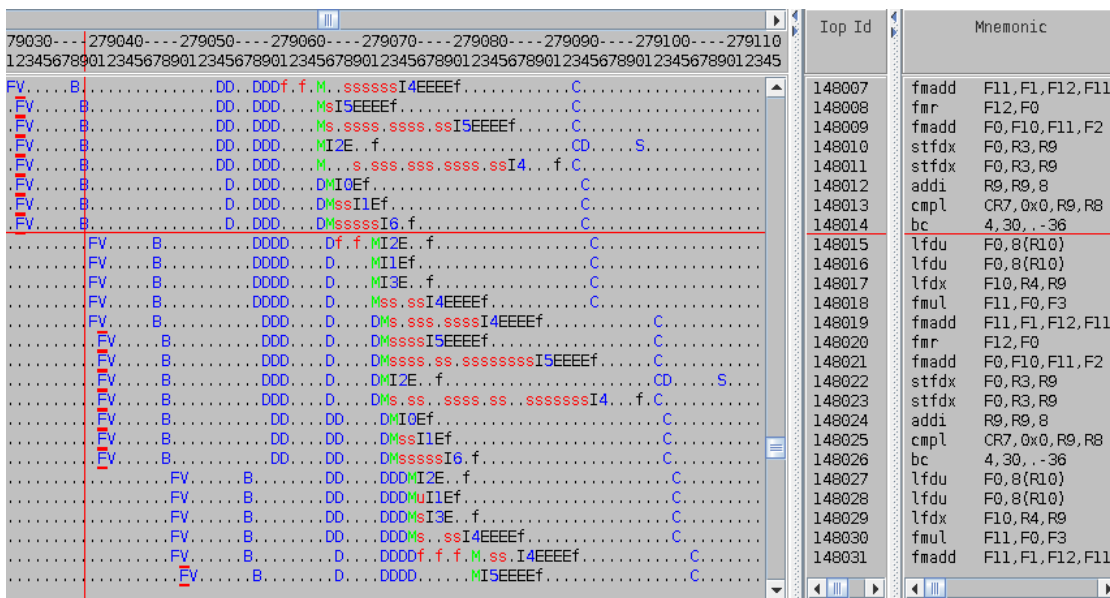


Figure 3.12: GCC loop 1 pipeline (restrict, double).

In order too understand the cause of the actual loss the pipeline has to be investigated which will reveal problems such as pipeline stalling. Figures 3.12 and 3.13 are both screenshots taken from ScrollPipeViewer showing the pipeline generated by GCC for the first livemore loop. As shown by both figures there are more stalls in the restrict version of the code. This is illustrated as all the empty space or dots between the pipeline stages, where

Iop Id	Mnemonic
148007	fmadd F0, F12, F1, F0
148008	fmadd F0, F11, F0, F2
148009	stf dx F0, R3, R9
148010	stf dx F0, R3, R9
148011	addi R9, R9, 8
148012	cmpl CR7, 0x0, R9, R8
148013	bc 4, 30, -, -36
148014	lfd F12, 0(R10)
148015	lfdx F11, R4, R9
148016	lfdx F0, 8(R10)
148017	lfdx F0, 8(R10)
148018	fmul F0, F3, F0
148019	fmadd F0, F12, F1, F0
148020	fmadd F0, F11, F0, F2
148021	stf dx F0, R3, R9
148022	stf dx F0, R3, R9
148023	addi R9, R9, 8
148024	cmpl CR7, 0x0, R9, R8
148025	bc 4, 30, -, -36
148026	lfd F12, 0(R10)
148027	lfdx F11, R4, R9
148028	lfdx F0, 8(R10)
148029	lfdx F0, 8(R10)
148030	fmul F0, F3, F0
148031	fmadd F0, F12, F1, F0

Figure 3.13: GCC loop 1 pipeline (no restrict, double).

each represents one clock cycle. The marking "s" means "waiting for sources" and the marking "f" means that an issue queue is full, in this case the floating point issue queue. This happens more frequently in the restrict version since it has one more floating point instruction: the fmr. The dispatch groups are also a cause of the increased amount of stalls, by having less than optimal scheduling new groups have to wait for the previous group before they can be dispatched. In Figure 3.12 the 4 IOPs below the red marking are in one dispatch group. It consists of 1 cracked lfdx, 1 lfdx, and 1 fmul(IOP 148018). When the floating point issue queue is full the dispatch of this group must wait. This means the lfd IOPs cannot be issued until the fmul has a spot in the issue queue. If the lfdx and fmul were in separate groups the lfdx could be issued while the fmul is waiting for the issue queue. The stalls in the pipeline for Clang are caused by the same reason as the GCC version, the floating point issue queue is full. Note that despite that there are 2 FPUs there is only issue queue that is full due to the way instructions enter the two issue queues for each FPU.

The XL C pipeline can be seen in Figures 3.14 and 3.15. In the restrict version there is less stalls due to waiting for sources and the floating point issue queue being full.

Figures 3.1-3.4 reveals something interesting, when XL C and GCC unrolls the loop there is an actual performance gain instead of a loss which happens without unrolling.

In the unrolled version with restrict there is no need to use fmr to save $z[k+11]$ in a register since it is already stored in a register for each unrolled iteration, this enables better throughput in the pipeline. The restrict version is unrolled 8 times and it has 8 stores and 16 loads. The version without restrict is unrolled 4 times and it has 4 stores and 12 loads. There are also more possibilities to reorder the instructions which can be seen in Listing 3.6a, most of loads are in the beginning of the loop and all the stores are at end. By comparing Figures 3.16 and 3.12 it is easy to see that the pipeline is better utilized with unrolling and thus leads to better performance.

Top Id	Mnemonic
154394	bc 25,0,-40
154395	lfd F0,8(R3)
154396	fmul F4,F29,F0
154397	stfd F5,-8(R6)
154398	stfd F5,-8(R6)
154399	addi R3,R3,8
154400	lfd F3,0(R4)
154401	fmadd F5,F3,F1,F30
154402	addi R6,R6,8
154403	fmadd F1,F31,F2,F4
154404	fnr F2,F0
154405	addi R4,R4,8
154406	bc 25,0,-40
154407	lfd F0,8(R3)
154408	fmul F4,F29,F0
154409	stfd F5,-8(R6)
154410	stfd F5,-8(R6)
154411	addi R3,R3,8
154412	lfd F3,0(R4)
154413	fmadd F5,F3,F1,F30
154414	addi R6,R6,8
154415	fmadd F1,F31,F2,F4
154416	fnr F2,F0
154417	addi R4,R4,8
154418	bc 25,0,-40

Figure 3.14: XL C loop 1 pipeline (restrict, double).

Top Id	Mnemonic
154445	bc 25,0,-40
154446	stfd F0,0(R3)
154447	stfd F0,0(R3)
154448	lfd F1,8(R4)
154449	fmul F0,F29,F1
154450	lfd F2,0(R4)
154451	fmadd F0,F31,F2,F0
154452	lfd F1,8(R5)
154453	fmadd F0,F1,F0,F30
154454	addi R3,R3,8
154455	addi R4,R4,8
154456	addi R5,R5,8
154457	bc 25,0,-40
154458	stfd F0,0(R3)
154459	stfd F0,0(R3)
154460	lfd F1,8(R4)
154461	fmul F0,F29,F1
154462	lfd F2,0(R4)
154463	fmadd F0,F31,F2,F0
154464	lfd F1,8(R5)
154465	fmadd F0,F1,F0,F30
154466	addi R3,R3,8
154467	addi R4,R4,8
154468	addi R5,R5,8
154469	bc 25,0,-40

Figure 3.15: XL C loop 1 pipeline (no restrict, double).

```

1 lfd      f7,24(r10)
2 lfd      f9,40(r10)
3 addi    r9,r9,64
4 ...
5 fmul    f29,f7,f3
6 fmul    f31,f9,f3
7 fmul    f28,f6,f3
8 ...
9 fmadd   f9,f1,f9,f13
10 fmadd  f10,f1,f10,f4
11 lfdx   f13,r4,r31
12 fmadd  f11,f1,f11,f5
13 lfdx   f5,r4,r7
14 fmadd  f31,f30,f6,f2
15 ...
16 lfdx   f10,r4,r11
17 fmadd  f11,f10,f11,f2
18 stfdx  f28,r3,r8
19 stfdx  f31,r3,r6
20 stfdx  f4,r3,r31
21 ...
22 bne    cr7,10001,f50
    
```

```

1 lfd      f0,8(r10)
2 lfd      f5,0(r10)
3 addi    r12,r9,8
4 addi    r11,r9,16
5 lfdx   f7,r4,r9
6 addi    r8,r9,24
7 fmul    f4,f3,f0
8 fmadd   f6,f5,f1,f4
9 fmadd   f8,f7,f6,f2
10 stfdx  f8,r3,r9
11 ...
12 lfd     f10,24(r10)
13 lfdx   f0,r4,r8
14 lfd    f13,32(r10)
15 fmul    f11,f3,f13
16 fmadd   f12,f10,f1,f11
17 fmadd   f4,f0,f12,f2
18 stfdx  f4,r3,r8
19 bne    10001,eac
    
```

a: Restrict.

b: No restrict.

Listing 3.6: GCC loop 1 assembler (double, unroll).

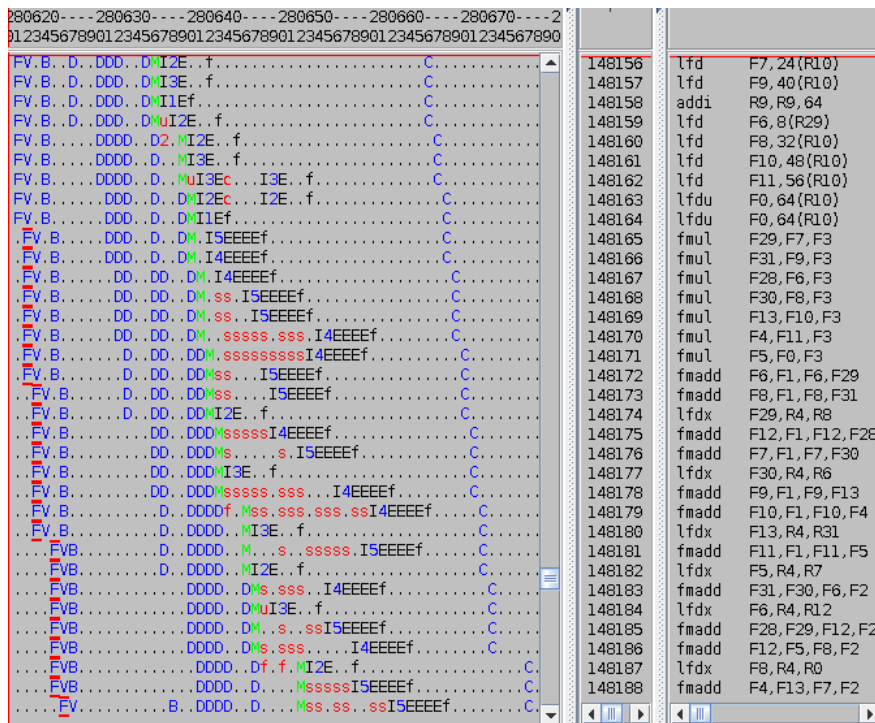


Figure 3.16: GCC loop 1 pipeline (restrict, double, unroll).

Address	OpCode	Operands
148114	lfd	F0, 8(R10)
148115	lfd	F5, 0(R10)
148116	addi	R12, R0, 8
148117	addi	R11, R0, 16
148118	lfdx	F7, R4, R9
148119	addi	R8, R9, 24
148120	fmul	F4, F3, F0
148121	fmadd	F6, F5, F1, F4
148122	fmadd	F8, F7, F6, F2
148123	stfdx	F8, R3, R9
148124	stfdx	F8, R3, R9
148125	addi	R0, R0, 32
148126	cmpl	CR0, 0x0, R9, R7
148127	lfd	F9, 16(R10)
148128	lfd	F13, 8(R10)
148129	lfdx	F12, R4, R12
148130	fmul	F10, F3, F9
148131	fmadd	F11, F13, F1, F10
148132	fmadd	F0, F12, F11, F2
148133	stfdx	F0, R3, R12
148134	stfdx	F0, R3, R12
148135	lfd	F5, 24(R10)
148136	lfd	F4, 16(R10)
148137	lfdx	F8, R4, R11
148138	fmul	F6, F3, F5

Figure 3.17: GCC loop 1 pipeline (no restrict, double, unroll).

Using int the compilers have done the same optimization with regards to restrict, i.e replaced a load with a move instruction (Listing 3.7-3.8). All three compilers run in the same amount of time using int and restrict (Figure 3.8). However the XLC version with no restrict is slower compared the same version of Clang and GCC, also Clang without restrict is the fastest. It so happens that the code is better scheduled in Clang without restrict. This was confirmed using the pipeline simulator. It can be seen that there are more "u"s (which means that the execution unit is not free) in 3.20 than in 3.21. We counted the clock cycles for a few iterations of the loop and most of the time each iteration in the restrict version takes more cycles to complete.

With unrolling the behaviour was equivalent with what was seen using double.

```

1  lwz    r27,0(r29)
2  mullw r0,r0,r6
3  lwz    r25,0(r30)
4  addi   r30,r30,4
5  addi   r29,r29,4
6  mullw r26,r27,r8
7  add    r0,r26,r0
8  mullw r0,r0,r25
9  add    r26,r0,r7
10 addi   r0,r28,4
11 stw    r26,0(r28)
12 mr     r28,r0
13 mr     r0,r27
14 bdnz   10000cb0 <liver_loop1+0x50>

```

a: Restrict.

```

1  lwz    r0,-4(r30)
2  lwz    r28,0(r30)
3  lwz    r27,0(r12)
4  addi   r12,r12,4
5  addi   r30,r30,4
6  mullw r0,r0,r6
7  mullw r28,r28,r8
8  add    r0,r28,r0
9  mullw r0,r0,r27
10 add    r28,r0,r7
11 addi   r0,r29,4
12 stw    r28,0(r29)
13 mr     r29,r0
14 bdnz   10000ca4 <liver_loop1+0x44>

```

b: No restrict.

Listing 3.7: Clang loop 1 assembler (int).

```

1  mr     r7,r8
2  lwz    r8,4(r4)
3  lwz    r0,4(r3)
4  addi   r4,r4,4
5  mullw r6,r28,r7
6  mullw r7,r26,r8
7  add    r9,r6,r7
8  addi   r3,r3,4
9  mullw r10,r9,r0
10 add    r9,r27,r10
11 stw    r9,4(r5)
12 addi   r5,r5,4
13 bdnz+  100004a0

```

a: Restrict.

```

1  lwz    r0,0(r4)
2  addi   r3,r3,4
3  lwz    r6,4(r4)
4  lwz    r7,4(r5)
5  addi   r4,r4,4
6  mullw r0,r29,r0
7  mullw r6,r27,r6
8  add    r8,r0,r6
9  addi   r5,r5,4
10 mullw r7,r8,r7
11 add    r8,r28,r7
12 stw    r8,4(r3)
13 bdnz+  10000510

```

b: No restrict.

Listing 3.8: XLC loop 1 assembler (int).

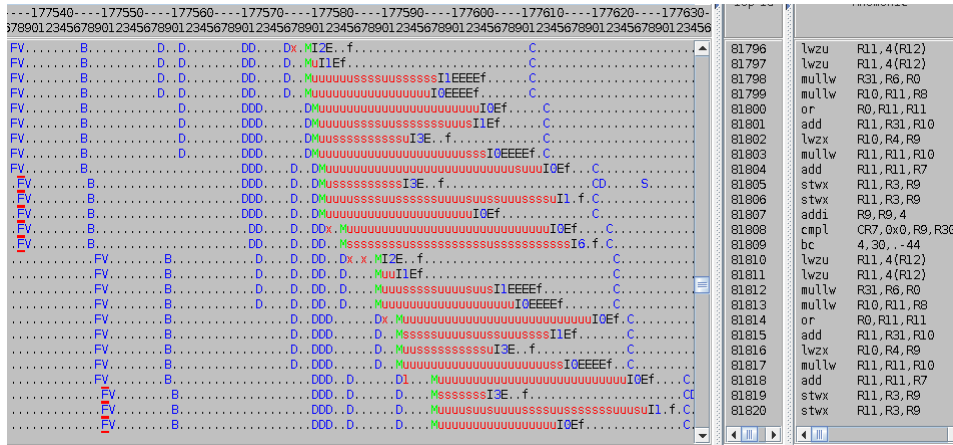


Figure 3.18: GCC loop 1 pipeline (restrict, int).

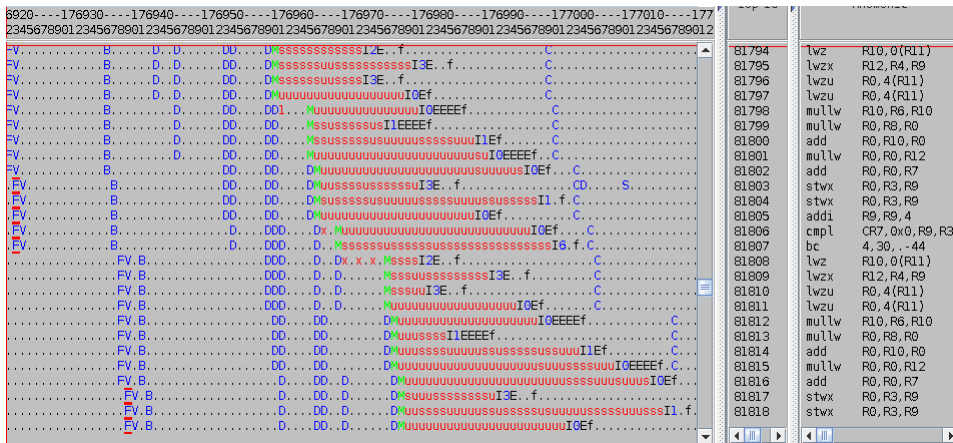


Figure 3.19: GCC loop 1 pipeline (no restrict, int).

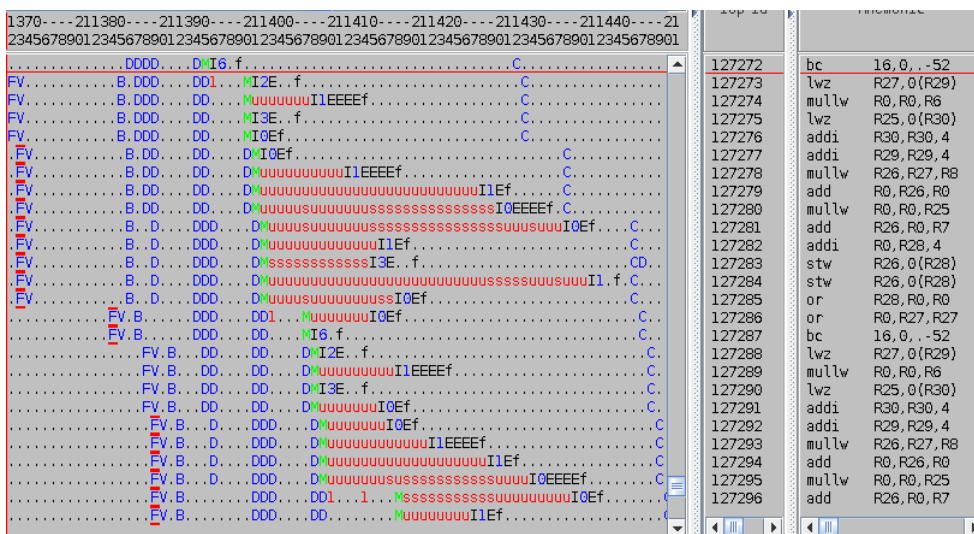


Figure 3.20: Clang loop 1 pipeline (restrict, int).

Address	OpCode	OpName	OpArg
127267	bc	bc	16, 0, -52
127268	lzw	lzw	R0, -4(R30)
127269	lzw	lzw	R28, 0(R30)
127270	lzw	lzw	R27, 0(R12)
127271	addi	addi	R12, R12, 4
127272	addi	addi	R30, R30, 4
127273	mullw	mullw	R0, R0, R6
127274	mullw	mullw	R28, R28, R8
127275	add	add	R0, R28, R0
127276	mullw	mullw	R0, R0, R27
127277	add	add	R28, R0, R7
127278	addi	addi	R0, R29, 4
127279	stw	stw	R28, 0(R29)
127280	stw	stw	R28, 0(R29)
127281	or	or	R29, R0, R0
127282	bc	bc	16, 0, -52
127283	lzw	lzw	R0, -4(R30)
127284	lzw	lzw	R28, 0(R30)
127285	lzw	lzw	R27, 0(R12)
127286	addi	addi	R12, R12, 4
127287	addi	addi	R30, R30, 4
127288	mullw	mullw	R0, R0, R6
127289	mullw	mullw	R28, R28, R8
127290	add	add	R0, R28, R0
127291	mullw	mullw	R0, R0, R27

Figure 3.21: Clang loop 1 pipeline (no restrict, int).

3.3 Loop 5

Loop 5, as seen in Listing 3.9, is a function which performs tri-diagonal elimination on linear systems. Just like in the first Livermore loop it is clear that an optimization opportunity in this loop would be to exploit the fact that once $x[i]$ is calculated it will be used in the next loop iteration as $x[i-1]$. Unlike the previous loop however the benefit of restrict is less obvious here since x is the source for the optimization, both the reads and writes are done to x whereas in the previous loop x was only written to and never read from. Looking again at Figure 3.1 we see this time much more promising numbers, the execution of time of GCC is reduced by 32.63% with restrict and for XL C it is reduced by an impressive 75.57%.

Clang on the other hand show no improvements with restrict. It is obvious that restrict had an impact on performance but one might wonder why it even made a difference. The simple answer is that a compiler may see potential data hazards which a regular software engineer might not see. These potential problems, which may turn out to be imaginary, hinders the compiler from performing optimization opportunities as described above.

```

1 void liver_loop5 (TYPE*RESTRICT x, TYPE*RESTRICT y,
2                   TYPE*RESTRICT z, int n, int loop) {
3     for ( int l=1 ; l<=loop ; l++ ) {
4         for ( int i=1 ; i < n ; i++ ) {
5             x[i] = z[i]*( y[i] - x[i-1] );
6         }
7     }
8 }

```

Listing 3.9: Livermore loop 5 in C.

GCC has replaced one load with one fmr as seen in Listing 3.10a and 3.10b. Clang has removed one load in both versions which means that Clang’s alias analysis has managed to determine that there is no alias present in the version without restrict. This also explains why Clang shows no performance improvement in Figure 3.1. XL C has removed one load and scheduled the code better than GCC and Clang. Looking at the instructions on lines 4-6 in Listing 3.10a and lines 6-8 in Listing 3.11a they all write to the same register. By comparing this to the code in Listing 3.12a it becomes evident that XL C has managed to schedule the code better which results in fewer data dependencies between the instructions.

```

1 lfd  f11,8(r9)
2 lfd  f0,8(r8)
3 cmplw cr7,r9,r7
4 fsub  f0,f0,f12
5 fmul  f0,f0,f11
6 stfdu f0,8(r10)
7 fmr   f12,f0
8 bne  cr7,10001f80

```

a: Restrict.

```

1 lfd  f12,8(r9)
2 lfd  f11,0(r10)
3 lfd  f0,8(r8)
4 cmplw cr7,r9,r7
5 fsub  f0,f0,f11
6 fmul  f0,f0,f12
7 stfdu f0,8(r10)
8 bne  cr7,10001f60

```

b: No restrict.

Listing 3.10: GCC loop 5 assembler (double).

```

1 lfd    f1,0(r11)
2 lfd    f2,0(r10)
3 addi   r0,r12,8
4 addi   r10,r10,8
5 addi   r11,r11,8
6 fsub   f0,f1,f0
7 fmul   f0,f2,f0
8 stfd   f0,0(r12)
9 mr     r12,r0
10 bdnz  10000e00

```

a: Restrict

```

1 lfd    f1,0(r11)
2 lfd    f2,0(r10)
3 addi   r0,r12,8
4 addi   r10,r10,8
5 addi   r11,r11,8
6 fsub   f0,f1,f0
7 fmul   f0,f2,f0
8 stfd   f0,0(r12)
9 mr     r12,r0
10 bdnz  10000dfc

```

b: No restrict.**Listing 3.11:** Clang loop 5 assembler (double).

```

1 lfd    f1,8(r9)
2 fmul   f2,f1,f0
3 lfd    f3,8(r4)
4 fnmsub f0,f1,f0,f3
5 stfd   f2,8(r6)
6 addi   r4,r4,8
7 addi   r9,r9,8
8 addi   r6,r6,8
9 bdnz+  100006c0

```

a: Restrict.

```

1 lfd    f0,8(r9)
2 lfd    f1,0(r6)
3 fsub   f2,f0,f1
4 lfd    f0,8(r10)
5 fmul   f1,f0,f2
6 stfd   f1,8(r6)
7 addi   r6,r6,8
8 addi   r9,r9,8
9 addi   r10,r10,8
10 bdnz+ 100007f0

```

b: No restrict.**Listing 3.12:** XL C loop 5 assembler (double).

From the pipeline in Figure 3.23 it can be seen that the lfd F11,0(R10) instruction at the red marker has to wait for the previous store at IOP 148005 to finish. This causes stalling which gets worse and as worse for each iteration, even the global completion table becomes full which can be seen by the marker "c". The same behaviour is present in the X LC version (Figure 3.25). In the restrict version of GCC (Figure 3.22) and X LC (Figure 3.24) this behaviour disappears. It can also be seen that there are less "waiting for sources" stalls in the X LC version. This corresponds well with the execution time in Figure 3.7

Using unrolling the restrict version of X LC and GCC unrolled the loop 8 times. Unrolling did not yield any performance benefit for X LC (Figure 3.3). But for GCC unrolling improved the performance for the same reasons as in loop 1, since the fmr was not removed in the version without unrolling.

The int version of the loop does the same optimizations and exhibits the same behaviour in the pipeline as the double version.

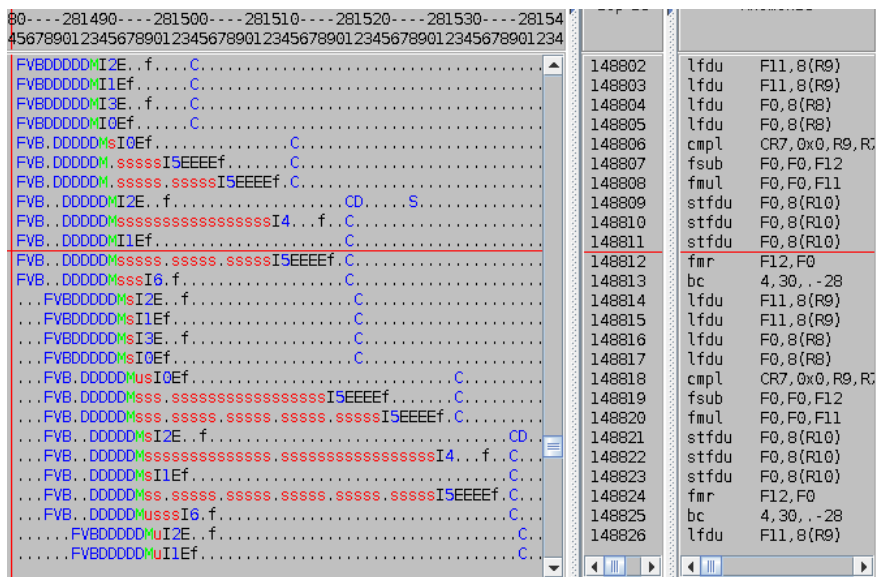


Figure 3.22: GCC loop 5 pipeline (restrict, double).

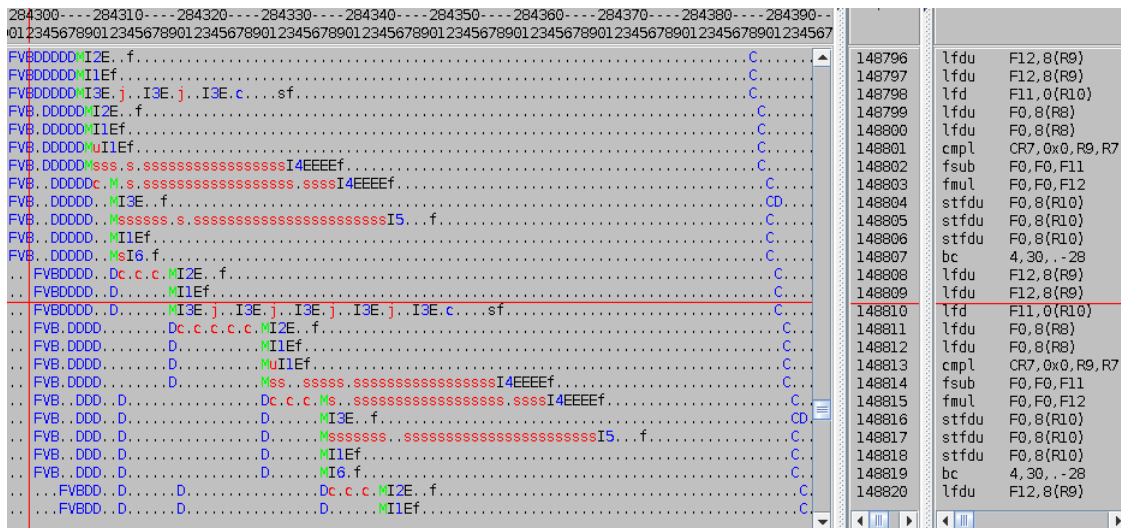


Figure 3.23: GCC loop 5 pipeline (no restrict, double).

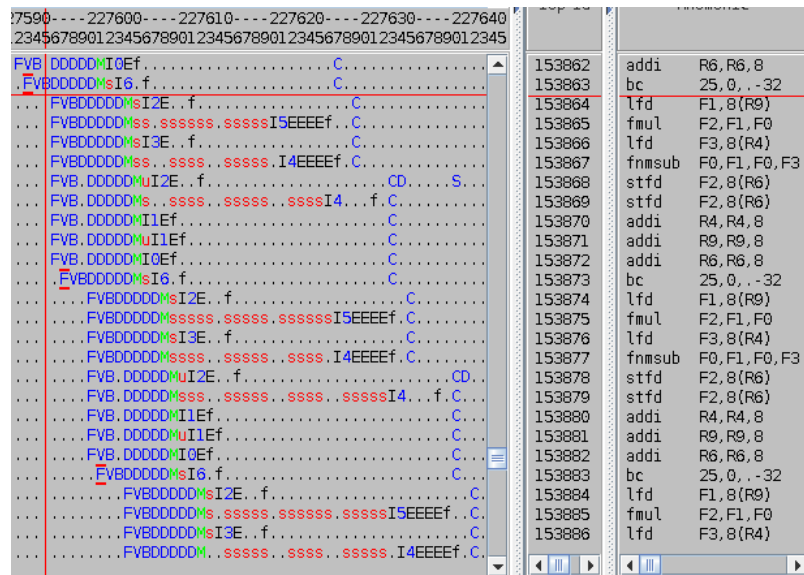


Figure 3.24: XL C loop 5 pipeline (restrict, double).

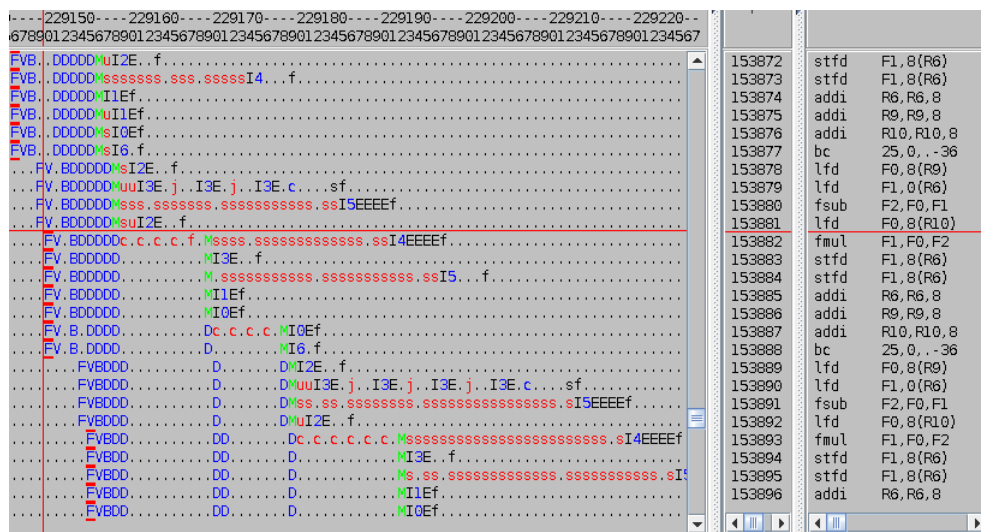


Figure 3.25: XL C loop 5 pipeline (no restrict, double).

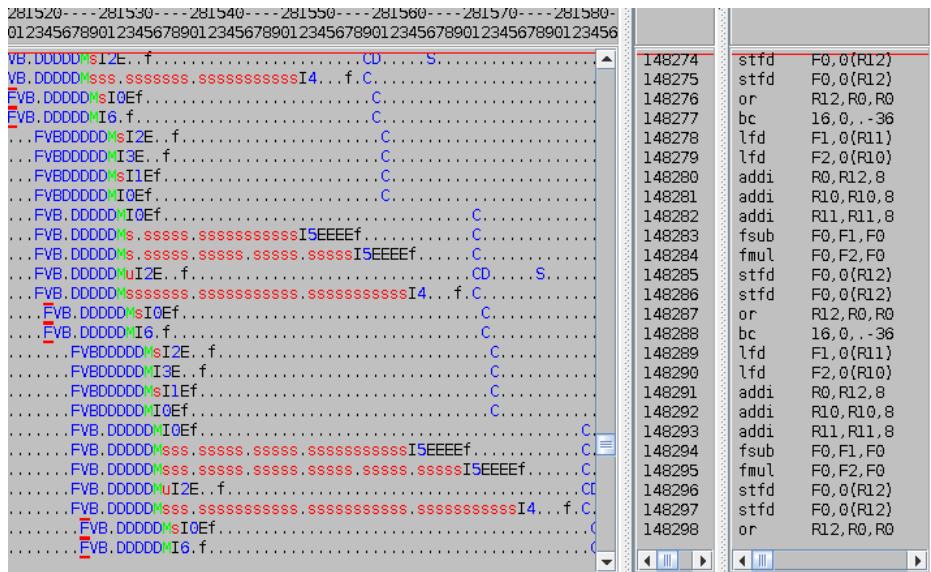


Figure 3.26: Clang loop 5 pipeline (restrict, double).

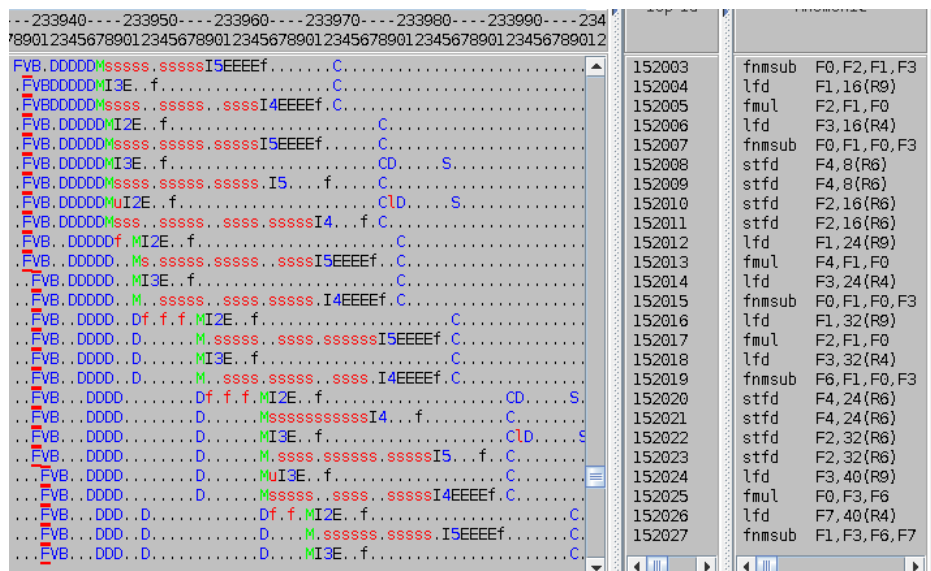


Figure 3.27: XL C loop 5 pipeline (restrict, double, unroll).

3.4 Loop 8

Listing 3.13 show Livermore loop 8 which is an ADI integration code fragment that can be used to solve partial differential equations. This loop presents many opportunities to remove or replace load instructions. The number of loads for each iteration in the regular version with double is 23 for GCC, Clang and XL C. Using restrict XL C managed to reduce the number of loads to 15, GCC and Clang reduced it to 9 loads and 6 move instructions. XLC have also made use of the fnmsub instruction which saves many clock cycles while both GCC and Clang failed to do so.

GCC did not unroll the loop when compiled with unrolling, although GCC scheduled the code slightly differently, which can explain the small decrease in performance in Figure 3.1. XL C unrolled once and made a loop split which in turn resulted in a small performance improvement when unrolling. Looking at Figure 3.1-3.6 it is clear that using restrict benefited the performance of this loop greatly.

```

1 void liver_loop8 (TYPE *RESTRICT du1,TYPE *RESTRICT du2 ,
2     TYPE *RESTRICT du3, TYPE u1[RESTRICT][101][5],
3     TYPE u2[RESTRICT][101][5],TYPE u3[RESTRICT][101][5],
4     TYPE a11, TYPE a12, TYPE a13, TYPE a21,
5     TYPE a22, TYPE a23, TYPE a31, TYPE a32, TYPE a33,
6     TYPE sig, int n, int loop) {
7
8     for (int l=1 ; l<=loop ; l++ ) {
9         int n11 = 0;
10        int n12 = 1;
11        for (int kx=1 ; kx<3 ; kx++ ) {
12            for (int ky=1 ; ky<n ; ky++ ) {
13                du1[ky] = u1[n11][ky+1][kx] - u1[n11][ky-1][kx];
14                du2[ky] = u2[n11][ky+1][kx] - u2[n11][ky-1][kx];
15                du3[ky] = u3[n11][ky+1][kx] - u3[n11][ky-1][kx];
16                u1[n12][ky][kx]=
17                    u1[n11][ky][kx]+a11*du1[ky]+a12*du2[ky]
18                    + a13*du3[ky] + sig*(u1[n11][ky][kx+1]
19                    -2.0*u1[n11][ky][kx]+u1[n11][ky][kx-1]);
20                u2[n12][ky][kx]=
21                    u2[n11][ky][kx]+a21*du1[ky]+a22*du2[ky]
22                    + a23*du3[ky] + sig*(u2[n11][ky][kx+1]
23                    -2.0*u2[n11][ky][kx]+u2[n11][ky][kx-1]);
24                u3[n12][ky][kx]=
25                    u3[n11][ky][kx]+a31*du1[ky]+a32*du2[ky]
26                    + a33*du3[ky] + sig*(u3[n11][ky][kx+1]
27                    -2.0*u3[n11][ky][kx]+u3[n11][ky][kx-1]);
28            }
29        }
30    }
31 }

```

Listing 3.13: Livermore loop 8 in C.

3.5 Loop 11

Loop 11 is shown in Listing 3.14 and is used to calculate partial sums of an array. The optimization situation is the same as described in section 3.3 - the compiler is unable by default to reuse the value of `x[k]` in the next iteration as `x[k-1]`, but with `restrict` enabled the compiler is free to perform this optimization. Unlike the previous loops there is also another important difference, there is no need to save the `x[k]` value to a separate register. Instead the register that contains the value of `x[k]` can be reused whereas in previous loops a separate extra register was needed.

```
1 void liver_loop11 (TYPE *RESTRICT x, TYPE *RESTRICT y ,
2   int n, int loop) {
3   for (int l=1 ; l<=loop ; l++ ) {
4     x[0] = y[0];
5     for (int k=1 ; k<n ; k++ ) {
6       x[k] = x[k-1] + y[k];
7     }
8   }
9 }
```

Listing 3.14: Livermore loop 11 in C.

The results presented in Figure 3.1-3.4 show once again that using `restrict` greatly improved the execution time of the loop with both GCC and XL C, especially when using `int` combined with `restrict` and unrolling (the execution time of GCC and XL C is reduced by 82.26% and 85.93% respectively). It should be noted that Clang's alias analysis determined correctly that the arrays do not alias which explains why Clang shows no improvement in Figure 3.5 and 3.6.

All three compilers have managed to remove 1 load in the `restrict` version as can be seen in Listings 3.15, 3.16 and 3.17. In the three versions without `restrict` the load `x[i-1]` has to wait for the previous store to finish which causes many pipeline stalls, this is the same situation as in Livermore loop 5 (section 3.3).

Even when just unrolling the loop without `restrict` a significant performance boost can be seen since for each unrolled iteration there is no need to load the `x[i-1]` value since it is already stored in a register. However `x[i-1]` must still be loaded for the next loop iteration. Using `restrict` this load can be removed.

Using `restrict` GCC unrolled the loop 8 times but the last `x[i-1]` load was replaced with a move instruction. XL C unrolled the loop 8 times and managed to schedule the code so it was not needed to use a move instruction for the last `x[i-1]` load.

The performance is as expected in Figures 3.1 and 3.3. An interesting note is that XL C has produced some rather strange code - if `n` happens to be less than `k` before the first loop iteration XL C will execute line 8-9 in Listing 3.17a and line 10-11 in 3.17b. This will run the `x[0] = y[0]` statement many times (value of the variable `loop` times to be precise) rather than just once. This is of course not wrong but is rather strange behavior which possibly might be a bug and is present in all the variants of the loop.


```

1 lfd    f12,8(r9)
2 fadd   f0,f0,f12
3 cmplw cr7,r9,r8
4 stfdu  f0,8(r10)
5 bne    cr7,100023e0

```

a: Restrict

```

1 lfd    f0,8(r9)
2 lfd    f12,0(r10)
3 fadd   f0,f12,f0
4 cmplw cr7,r9,r8
5 stfdu  f0,8(r10)
6 bne    cr7,10002320

```

b: No restrict.**Listing 3.15:** GCC loop 11 assembler (double).

```

1 lfd    f2,0(r10)
2 addi   r12,r11,8
3 addi   r5,r5,-1
4 addi   r10,r10,8
5 cmplwi cr1,r5,0
6 fadd   f1,f1,f2
7 stfd   f1,0(r11)
8 mr     r11,r12
9 bne    cr1,100011e0

```

a: Restrict

```

1 lfd    f1,0(r11)
2 addi   r0,r12,8
3 addi   r5,r5,-1
4 addi   r11,r11,8
5 cmplwi cr1,r5,0
6 fadd   f0,f0,f1
7 stfd   f0,0(r12)
8 mr     r12,r0
9 bne    cr1,10001184

```

b: No restrict.**Listing 3.16:** Clang loop 11 assembler (double).

```

1 lfd    f2,8(r8)
2 fadd   f1,f1,f2
3 stfd   f1,8(r5)
4 addi   r8,r8,8
5 addi   r5,r5,8
6 bdnz+ 10000ce0
7 ...
8 stfd   f0,0(r3)
9 bdnz+ 10000d10 # jump to line 8

```

a: Restrict

```

1 lfd    f0,0(r5)
2 lfd    f1,8(r8)
3 fadd   f2,f0,f1
4 stfd   f2,8(r5)
5 addi   r5,r5,8
6 addi   r8,r8,8
7 bdnz+ 10000f90
8 ...
9 lfd    f0,0(r4)
10 stfd   f0,0(r3)
11 bdnz+ 10000fd0 # jump to line 9

```

b: No restrict.**Listing 3.17:** XLC loop 11 assembler (double).

3.6 Loop 12

Livermore loop 12 as seen in 3.18 is used to calculate the forward difference. As seen in previous loops the compiler should be able to save $y[k+1]$ into a register use it as $y[k]$ in the next loop iteration if restrict is used. The interesting part about this loop is the execution time of XL C, it is actually worse with restrict for XL C as seen in Figures 3.3 and 3.4. GCC on the other hand has managed to produce better code and looking at Figures 3.7-3.8 we see that GCC is significantly faster than both XL C and Clang.

```

1 void liver_loop12(TYPE*RESTRICT x, TYPE*RESTRICT y,
2     int n, int loop) {
3     for (int l=1; l<=loop; l++) {
4         for (int k=0; k<n; k++) {
5             x[k] = y[k+1] - y[k];
6         }
7     }
8 }

```

Listing 3.18: Livermore loop 12 in C.

Even with Clang's own alias analysis restrict has managed to produce a little better code resulting in a small performance boost.

All three compilers have replaced one load with a move instruction. However when compiling GCC without the unroll flag set it actually still unrolled the loop once (only with restrict), this can be seen in Listing 3.19a. When compiled with the unroll flag it did a more aggressive unrolling. Clang's alias analysis failed this time which can be seen in Listing 3.20.

The reason why XL C is slower using type double with restrict is the same reason as why GCC is slower with restrict in section 3.2, the floating point issue queue becomes full as can be seen by Figure 3.28. There are also slightly more waiting for sources stalls compared to the version without restrict(Figure 3.29). This behaviour is not present in the GCC version since it unrolled the loop which leads to better utilization of the pipeline.

The reason why XL C is slower using type int with restrict is shown in Figure 3.30. The starting address of the loop (10001170) is near the end of an icache block boundary, this forces a second ifetch which can be seen by the red markers on the 'F's. This delays the branch predicted at the end of the iteration and in turn delays the fetch at the start of the next iteration. Compare this with Figure 3.31 where the starting address of the loop (100011520) is near the start of an icache block boundary, a whole iteration can be fetched in one cycle. Since this is a very small loop this causes a slowdown of around 30%.

It should be noted that in the version without restrict XL C has done something peculiar. It has generated three loops and depending on the input n, only one of them is selected to run. We believe the reasons for this is are as follows: when compiled with the optimization flag -O3 unrolling is enabled and with unrolling it is necessary generate multiple loops in order to handle different sizes of the input. However when disabling unrolling with the flag -qnounroll, XL C has generated the different loops but when recognizing the flag it keeps all three loops anyway instead of removing the unnecessary ones.

Unrolling provides the same benefits as explained in section 3.2.

```

1 lfd    f12,0(r10)
2 lfd    f0,8(r10)
3 addi   r8,r8,2
4 addi   r9,r9,16
5 cmpw   cr7,r8,r7
6 addi   r10,r10,16
7 fsub   f10,f12,f11
8 fsub   f12,f0,f12
9 fmr    f11,f0
10 stfd  f10,-16(r9)
11 stfd  f12,-8(r9)
12 bne    cr7,10002460

```

a: Restrict

```

1 lfd    f0,8(r9)
2 lfd    f12,-8(r9)
3 cmplw  cr7,r9,r8
4 fsub   f0,f0,f12
5 stfdu  f0,8(r10)
6 bne    cr7,10002380

```

b: No restrict.

Listing 3.19: GCC loop 12 assembler (double).

```

1 lfd    f1,0(r10)
2 addi  r11,r9,8
3 addi  r10,r10,8
4 fsub  f0,f1,f0
5 stfd  f0,0(r9)
6 mr    r9,r11
7 fmr   f0,f1
8 bdnz  10001254
    
```

a: Restrict

```

1 lfd    f0,-8(r8)
2 lfd    f1,0(r8)
3 addi  r11,r9,8
4 addi  r10,r10,-1
5 addi  r8,r8,8
6 cmplwi cr1,r10,0
7 fsub  f0,f1,f0
8 stfd  f0,0(r9)
9 mr    r9,r11
10 bne  cr1,100011f4
    
```

b: No restrict.

Listing 3.20: Clang loop 12 assembler (double).

```

1 lfd    f1,8(r4)
2 fsub  f2,f1,f0
3 fmr   f0,f1
4 stfd  f2,8(r3)
5 addi  r4,r4,8
6 addi  r3,r3,8
7 bdnz+ 10000d80
    
```

a: Restrict

```

1 lfd    f0,16(r3)
2 ...
3 bdnz+ 10001070 # jump to line 1
4 ...
5 lfd    f0,16(r3)
6 lfd    f1,8(r3)
7 fsub  f2,f0,f1
8 addi  r3,r3,8
9 stfd  f2,8(r4)
10 addi  r4,r4,8
11 bdnz+ 100010e0 # jump to line 5
12 ...
13 lfd    f0,16(r3)
14 ...
15 bdnz+ 10001140 # jump to line 13
    
```

b: No restrict.

Listing 3.21: X LC loop 12 assembler (double).

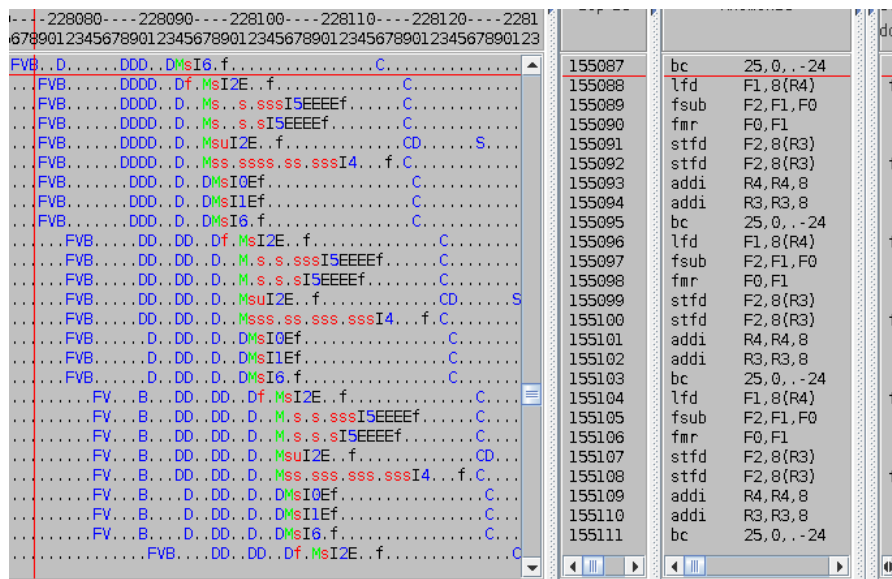


Figure 3.28: XL C loop 12 pipeline (restrict, double).

Inst Addr	Mnemonic	Data
155111	bc	25, 0, -24
155112	lfd	F0, 16(R3)
155113	lfd	F1, 8(R3)
155114	fsub	F2, F0, F1
155115	addi	R3, R3, 8
155116	stfd	F2, 8(R4)
155117	stfd	F2, 8(R4)
155118	addi	R4, R4, 8
155119	bc	25, 0, -24
155120	lfd	F0, 16(R3)
155121	lfd	F1, 8(R3)
155122	fsub	F2, F0, F1
155123	addi	R3, R3, 8
155124	stfd	F2, 8(R4)
155125	stfd	F2, 8(R4)
155126	addi	R4, R4, 8
155127	bc	25, 0, -24
155128	lfd	F0, 16(R3)
155129	lfd	F1, 8(R3)
155130	fsub	F2, F0, F1
155131	addi	R3, R3, 8
155132	stfd	F2, 8(R4)
155133	stfd	F2, 8(R4)
155134	addi	R4, R4, 8
155135	bc	25, 0, -24

Figure 3.29: XL C loop 12 pipeline (double).

Inst Addr	Mnemonic	Inst Addr	Data Addr
83462	stw	R0, 8(R3)	ffe15004
83463	stw	R0, 8(R3)	ffe15004
83464	addi	R3, R3, 4	ffe15004
83465	bc	25, 0, -24	ffe15004
83466	or	R0, R5, R5	ffe1528c
83467	lwz	R5, 8(R4)	ffe1528c
83468	addi	R4, R4, 4	ffe15008
83469	subf	R0, R0, R5	ffe15008
83470	stw	R0, 8(R3)	ffe15008
83471	stw	R0, 8(R3)	ffe15008
83472	addi	R3, R3, 4	ffe1500c
83473	bc	25, 0, -24	ffe1500c
83474	or	R0, R5, R5	ffe15290
83475	lwz	R5, 8(R4)	ffe15290
83476	addi	R4, R4, 4	ffe1500c
83477	subf	R0, R0, R5	ffe1500c
83478	stw	R0, 8(R3)	ffe1500c
83479	stw	R0, 8(R3)	ffe1500c
83480	addi	R3, R3, 4	ffe15294
83481	bc	25, 0, -24	ffe15294
83482	or	R0, R5, R5	ffe15294
83483	lwz	R5, 8(R4)	ffe15294
83484	addi	R4, R4, 4	ffe15294
83485	subf	R0, R0, R5	ffe15294
83486	stw	R0, 8(R3)	ffe15294

Figure 3.30: XL C loop 12 pipeline (restrict, int).

	Iop Id	Mnemonic	Inst Addr	Data Addr
3250----156260----156270----156280-- 234567890123456789012345678901234567				
.FV.BDDDDMsI2E..f..C.....	83468	lwz R0,8(R3)	10001520	ffe15280
.FV.BDDDDMsI3E..f..C.....	83469	lwz R5,4(R3)	10001524	ffe1527c
.FV.BDDDDMsuIIEf..C.....	83470	addi R3,R3,4	10001528	
.FV.BDDDDMsussIOEf..C.....	83471	subf R0,R5,R0	1000152c	
.FV.B.DDDDDMuI2E..f..CD.....S.....	83472	stw R0,4(R4)	10001530	
.FV.B.DDDDDMsussuIO.f..C.....	83473	stw R0,4(R4)	10001530	ffe14ffc
.FV.B.DDDDDMuIIef..C.....	83474	addi R4,R4,4	10001534	
.FV.B.DDDDDMsI6.f..C.....	83475	bc 25,0,-24	10001538	
.....FVBDDDDMsI2E..f..C.....	83476	lwz R0,8(R3)	10001520	ffe15284
.....FVBDDDDMsI3E..f..C.....	83477	lwz R5,4(R3)	10001524	ffe15280
.....FVBDDDDMuIIef..C.....	83478	addi R3,R3,4	10001528	
.....FVBDDDDMsusuIOEf..C.....	83479	subf R0,R5,R0	1000152c	
.....FVB.DDDDDMsI2E..f..CD.....S.....	83480	stw R0,4(R4)	10001530	
.....FVB.DDDDDMsusuIO.f..C.....	83481	stw R0,4(R4)	10001530	ffe15000
.....FVB.DDDDDMuIIef..C.....	83482	addi R4,R4,4	10001534	
.....FVB.DDDDDMsI6.f..C.....	83483	bc 25,0,-24	10001538	
.....FVBDDDDMuI2E..f..C.....	83484	lwz R0,8(R3)	10001520	ffe15288
.....FVBDDDDMsI3E..f..C.....	83485	lwz R5,4(R3)	10001524	ffe15284
.....FVBDDDDMuIIef..C.....	83486	addi R3,R3,4	10001528	
.....FVBDDDDMsusuIOEf..C.....	83487	subf R0,R5,R0	1000152c	
.....FVB.DDDDDMsI2E..f..CD.....	83488	stw R0,4(R4)	10001530	
.....FVB.DDDDDMsusuIO.f..C.....	83489	stw R0,4(R4)	10001530	ffe15004
.....FVB.DDDDDMuIIef..C.....	83490	addi R4,R4,4	10001534	
.....FVB.DDDDDMsI6.f..C.....	83491	bc 25,0,-24	10001538	
.....FVBDDDDMsI2E..f..C.....	83492	lwz R0,8(R3)	10001520	ffe1528c
.....FVBDDDDMsI3E..f..C.....				

Figure 3.31: XL C loop 12 pipeline (int).

3.7 Loop 17

Livermore loop 17 as seen in Listing 3.22 is a function that performs an implicit conditional computation. When the loop starts a jump to label l61 will always occur first. Afterwards a either a jump to l60 or l62 will occur. A jump to l62 will lead to the next loop iteration while l60 will either jump back to l61 or proceed to l62. If one follows the jumps it becomes clear that while the code does have several array accesses none of them can be saved away into a register for later use since each jump to a new label increases the *i* variable which is used for the access of pointers. However with restrict the compiler can potentially schedule the code better due to the guarantee that no alias is present between the pointers.

```

1 void liver_loop17 (TYPE*RESTRICT vsp, TYPE*RESTRICT vstp,
2   TYPE*RESTRICT vxne, TYPE*RESTRICT ve3,
3   TYPE*RESTRICT vlr, TYPE*RESTRICT vlin,
4   TYPE*RESTRICT vxnd, int n, int loop) {
5   TYPE scale, xnm, e6, e3, xnc, xnei;
6   int ink, i, j;
7
8   for (int l=1 ; l<=loop ; l++ ) {
9     i = n-1;
10    j = 0;
11    ink = -1;
12    scale = 5.0 / 3.0;
13    xnm = 1.0 / 3.0;
14    e6 = 1.03 / 3.07;
15    goto l61;
16 l60:   e6 = xnm*vsp[i] + vstp[i];
17        vxne[i] = e6;
18        xnm = e6;
19        ve3[i] = e6;
20        i += ink;
21        if ( i==j ) goto l62;
22 l61:   e3 = xnm*vlr[i] + vlin[i];
23        xnei = vxne[i];
24        vxnd[i] = e6;
25        xnc = scale*e3;
26        if ( xnm > xnc ) goto l60;
27        if ( xnei > xnc ) goto l60;
28        ve3[i] = e3;
29        e6 = e3 + e3 - xnm;
30        vxne[i] = e3 + e3 - xnei;
31        xnm = e6;
32        i += ink;
33        if ( i != j ) goto l61;
34 l62:;;
35    }
36 }

```

Listing 3.22: Livermore loop 17 in C.

Looking at the results in Figures 3.1-3.6 it can be seen that GCC is slightly faster for double but slower for int. XL C is slower in both cases and Clang is neither better or slower. These differences are due to code scheduling, the code scheduled by GCC and XL C is slightly different when compiled with restrict. Listing 3.23 shows a small excerpt from the assembly which displays the only difference in the GCC version with type double, the stfdu and add instruction has been moved.

```

1 stfdu    f11,-8(r4)
2 lfd     f12,-8(r5)
3 add     r3,r12,r9
4 add     r11,r0,r9
5 cmpwi   cr6,r8,0
6 addi    r9,r9,-8
7 fmadd   f12,f10,f0,f12
8 lfd     f10,-8(r10)
9 fmul    f11,f12,f9
10 fcmpu   cr7,f0,f11

```

a: Restrict.

```

1 add     r3,r12,r9
2 lfd     f12,-8(r5)
3 add     r11,r0,r9
4 cmpwi   cr6,r8,0
5 addi    r9,r9,-8
6 fmadd   f12,f10,f0,f12
7 lfd     f10,-8(r10)
8 stfdu   f11,-8(r4)
9 fmul    f11,f12,f9
10 fcmpu   cr7,f0,f11

```

b: No restrict.

Listing 3.23: GCC loop 17 assembler (double).

Neither GCC or XL C unrolled the loop when compiled with unrolling, but there are some minor differences in the code scheduled with unrolling. This loop shows how sensitive PowerPC 970 can be when it comes to code scheduling.

Chapter 4

Discussion

4.1 Compiler discussion

The results presented in sections 3.1-3.7 from the previous Chapter contains both expected and surprising results. What is clear is that all the evaluated compilers have implemented restrict based optimization and that restrict can have an impact on the performance of a program.

Looking at Figures 3.1-3.2 we see that that total running time for the entire benchmark for GCC was reduced by 25.13% with double restrict and 29.66% with double restrict with unrolling. For int restrict the running time was reduced by 13.19% and 18.66% for int restrict with unrolling. The greatly reduced running time clearly show that with the help of restrict GCC was able to to make optimizations which it otherwise was unable to perform.

For XL C we see even bigger improvements for the total running time than GCC as seen in Figures 3.3-3.4. XL C managed to reduce the running time by 42.50% with double restrict and 48.86% with double restrict and unrolling. For the int type the running time was reduced by 30.22% with restrict and 39.37% with restrict unrolling. Simply put the improvements made by XL C with restrict are very impressive. When this thesis was started we did not anticipate that such vast improvements were possible with restrict optimizations but the IBM's compiler team behind XL C have done an excellent job with their optimizing compiler.

Clang show quite different results as seen in Figures 3.5-3.6. The total running time was actually increased by 3.78% with double restrict and 0.38% with int restrict. As mentioned earlier in Chapter 3 we were unable to make Clang perform loop unrolling and thus there is no data presented for unrolling. Since Clang already perform pointer analysis by default the mentioned figures show how restrict affected the performance in regards to Clang's own analysis. The results indicate that Clang's own pointer analysis does a rather good job in optimizing the code that would otherwise require restrict. It is however not

perfect and misses some opportunities as seen in loop 1 and 8.

As seen in Figures 3.7-3.10 XL C is clearly the better compiler overall compared to GCC and Clang when comparing execution times when using restrict. This is not very surprising since IBM develops both the compiler and the CPU architecture which probably means they have a much deeper knowledge of the CPU and can thus produce a better compiler for the CPU than IBM's competitors. While it is speculation from our part we suspect that XL C internally models the CPU pipeline very accurately. This would allow it to schedule code which utilize the pipeline better than GCC and Clang. From the same figures it can also be seen that GCC performs overall better than Clang and that Clang practically perform worse than both GCC and XL C in every loop with the exception of loop 8 (note that small variations very likely exists in the figures since the data is based on an average of ten runs of each loop). It is also clear that all the evaluated compilers have several loops which they have trouble optimizing with restrict which results in an increased running time instead of a decreased one.

Lastly Figure 3.11 (which show the total average performance increase relative to the compiler's non restrict type counterpart) establishes that XL C has the best average performance gain with restrict. GCC does a fine job as well while Clang shows a slight improvement for double but a slight decrease with int. We see that on average the usage of restrict benefited the loops in a positive and quite noticeable way.

It should be noted that the Livermore loops are a collection of small loops. Being small loops which are executed for many iterations they are more susceptible to big performance changes when optimizing code. Just because a loop in this thesis became faster or slower with restrict it of course does not mean that some other program will see the exact same changes. For instance, consider a C program which is 100 000 lines of code long. In this program there might exist a small loop which becomes 25% faster with restrict but is very rarely called. This program will obviously not become 25% faster, the speedup might not even be noticeable because the other 99.99% of the execution time is spent elsewhere.

A few remarks regarding Clang's results are in order since it easy to get an impression that restrict is a bad thing due to the increased running times and degraded average performance. In several of the loops Clang's own alias analysis managed to produce code that already did restrict like optimizations while in loops such as loop 1 and 8 it failed to do so. Loop 1 as described in section 3.2 produced code with restrict which turned out to perform much worse and since the first loop is one of the more time consuming loops this greatly affected the total time running time. The other mentioned loops actually had their execution time reduced but due to them being much faster loops their impact was less noticeable. In other words, if Clang did not perform any alias analysis (as this is written we are unsure if there are any flags for the version of Clang used in this thesis which turns off the analysis) restrict would very likely reduce the overall running time of the entire benchmark. It is clear that Clang does an overall good job with it's own analysis but since it is not perfect there are still merits to actually use restrict as seen in loop 8. But in this particular benchmark restrict did overall more damage than good for Clang.

Overall we conclude that XL C is the compiler of choice for the PowerPC if money and the lack of open source code is not an issue. XL C was overall better at optimizing code and it shows in the total running time. If money is an issue or open source code is required to use then GCC is the way to go. Sadly we have a hard time recommending Clang because it simply performed worse in practically all cases and was generally harder

to work with (for instance no unrolling).

4.2 Issues and problems encountered

The biggest problem faced in this thesis was the sheer amount of loops to cover. The benchmark that was developed as described in Chapter 3 consists of 14 different loops. Each loop produce eight different variants to investigate with four variants for each double and int type. The variants produced was the regular version, the restrict version, the unrolled version and the unrolled restrict version. Investigating each variant for each of the 14 loops for each of the three compilers makes it a total of $14 * 8 * 2 + 14 * 4 = 280$ loops to investigate (note that if unrolling worked for Clang it would have been $14 * 8 * 3 = 336$ loops). Since the actual analysis of the loops did not start at the beginning of the thesis but rather at the middle it was simply not possible to cover all the different loops with a detailed analysis. In the beginning of the analysis phase it was especially difficult to understand the results since our understanding of the PowerPC architecture and instruction set was limited which resulted in slower progress than expected. With that said it should be noted that we feel that all the interesting loops and deviations have been investigated and we in general feel confident that most of the loop variants can be explained by a couple of common cases which have been described in the specific loop analysis section.

Another issue was the benchmark running time. The difference between the fastest and the slowest loop was too big and it would have been more preferable if all the loops took roughly the same amount of time to execute. This also made the entire total running time for the benchmark too long, to get a good average time we ran each loop ten times which made the benchmark even further time consuming. Running the entire benchmark suite took over 12 hours. It is also unfortunate that we were unable to make Clang unroll loops, several different compiler flags were tried but none of them made Clang unroll loops.

Chapter 5

Conclusions

5.1 Recommendations regarding restrict

The data from the benchmark results in this thesis shows that restrict is potentially a powerful optimization which can make a program run faster but it can also make it run slower. Because of this it is unwise to use restrict blindly - adding restrict blindly everywhere in the code is potentially a recipe for disaster. As mentioned earlier in section 2.2 if code is compiled on a compiler which ignores restrict and the code is then compiled on a newer version of the same compiler or a different one which implements restrict it is possible that the program might run slower or even no longer be correct due to undefined behavior caused by wrong usage of restrict pointers. But when restrict actually does work as intended (as a performance speedup of course) it can greatly improve the running time of a small loop. If usage of restrict is desired it is recommended to try it in conjunction with loop unrolling since this can potentially reduce some of the pipeline problems caused by for instance a replacement of a load instruction.

When adding restrict to an existing or new codebase we recommend adding it on a smaller scale first. Identify a couple of interesting small loops which run often that can potentially be improved with restrict and run some tests and gradually expanded to more. Look at the assembly code and try to understand what the compiler actually has done with the newly optimized code. If the code became slower, if the possibility exists run the code in a pipeline simulator to understand what has happened and if there is anything you as a programmer can improve. For instance, making small changes in the C code could potentially schedule the code differently which might improve the performance.

For compiler developers restrict presents somewhat of a dilemma. It is our understanding that implementing restrict optimizations is not an easy task and as seen in this thesis the situation becomes even more problematic because of the possibility of degraded performance. To just replace a load instruction is not always a good optimization if the code is scheduled poorly as seen in Livermore loop 1 in section 3.2. Greater care must be taken

when scheduling code that has been optimized with restrict which most likely means that compiler developers must spend more time studying the PowerPC architecture to better schedule code to avoid pipeline problems. The payoff of investing that much time on for instance a better pipeline model in order to schedule restrict code better might simply not be worth the effort.

5.2 Further work

This thesis provides many interesting opportunities which are worth investigating further. One of the more obvious things would be to try to get benchmark results from Clang when it comes to unrolling. Because of the lack of unrolled data it is currently harder to effectively compare Clang with both GCC and XL C. Investigating additional compilers might give further insight on how well restrict based optimization are implemented.

A test by the authors running the benchmark with 64-bit instructions reveal some differences between certain loops with their 32-bit counterpart. It would be interesting to investigate why they differ, is it caused by specific 64-bit instructions or is the code scheduled differently?

Finally, running a quick pipeline simulation of a newer PowerPC implementation such as Power7 showed some improvements to loops that had degraded performance with restrict in this thesis. A future study could thus be to investigate if for instance a bigger issue queue solved some of the problems with degraded restrict performance.

Bibliography

- [1] Clang homepage. <http://clang.llvm.org>.
- [2] GCC homepage. <https://gcc.gnu.org/>.
- [3] Livermore loop homepage, original Fortran version. <http://www.netlib.org/benchmark/livermore>.
- [4] XL C homepage. <http://www-03.ibm.com/software/products/en/xlcpp-linux>.
- [5] D Callahan, S. Carr, and K. Kennedy. Improving Register Allocation for Subscripted Variables. 1990.
- [6] IBM. IBM PowerPC User's Manual Version 2.3 970MPRISC Microprocessor. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/DC3D43B729FDAD2C00257419006FB955/\protect\T1\textdollarfile/970MP_um.2008MAR07_pub.pdf.
- [7] International Organization for Standardization. C99 standard draft. <http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf>.
- [8] M. Mock. Why Programmer-specified Aliasing is a Bad Idea. 2004.
- [9] G. Ramalingam. The Undecidability of Aliasing. 1994.
- [10] Richie D. Dennis Ritchie on C99 restrict (no alias). <http://www.lysator.liu.se/c/dmr-on-noalias.html>.
- [11] J. Stokes. *Inside the Machine*. No Starch Press, 2007. Chapter 10 has been used as reference.

STUDENTER Anton Botvalde, Andreas LarssonHANDLEDARE Jonas Skeppstedt (LTH)EXAMINATOR Per Andersson (LTH)

Prestandaevaluering av ISO C restrict på Power-arkitekturen

POPULÄRVETENSKAPLIG SAMMANFATTNING **Anton Botvalde och Andreas Larsson**

Introduktion

Moderna C-kompilatorer kan utföra avancerade optimeringar som gör att ett C-program blir mycket snabbare. I C så tillåts det att flera olika pekare pekar på samma objekt i minnet, vilket är känt som alias. Detta försvårar möjligheterna för en kompilator att spara ett värde som en pekare pekar på i ett register då kompilatorn har svårt att avgöra om någon annan pekare eventuellt pekar på samma minnesplats. Vårt arbete har visat att optimeringen kan ge en prestandaökning men också, något oväntat, en prestandaförlust.

Bakgrund

För att underlätta pekaroptimeringar introducerades ett nytt nyckelord, restrict. Med restrict garanterar programmeraren både kompilatorn och andra programmerare att om en restrict-pekare ändrar värdet på det pekade objektet så är det den enda pekaren som kommer åt detta objekt. Med en sådan garanti blir det lättare för kompilatorn att utföra optimeringar som t.ex. spara undan värdet i ett register, vilket ofta leder till en prestandaökning. En läsning från ett register är avsevärt snabbare än en läsning från RAM-minnet.

I det här examensarbetet har vi undersökt om C-kompilatorerna GCC, Clang och XL C har implementerat optimeringar med användning av restrict på IBMs Power-arkitektur.

För att undersöka prestandapåverkan av restrict så konverterade vi ett Fortran-prestandamättningsprogram, mer känt som Livemore loops, till C. Looparna passar utmärkt som grund till utvärdering av restrict då de innehåller många möjligheter för restrict-baserade optimeringar.

Prestandaresultat

Totalt sett så gav restrict en prestandaökning men i vissa enskilda loopar så gav det en prestandaminskning. I loop 1 med GCC visade det sig att optimeringen bytte

en minnesläsning till en registerläsning vilket ledde till en sämre schemaläggning av koden. Restrict i Clang hade liten påverkan på prestandan då Clang utför en egen aliasanalys av pekare men det fanns även fall där analysen inte räckte till. I de fallen gjorde restrict både positiva och negativa förändringar.

Type	Double			Int		
	Restrict	Unroll	Unroll restrict	Restrict	Unroll	Unroll restrict
GCC	16.31	8.15	24.14	11.73	12.35	14.32
XL C	23.18	5.21	31.04	13.71	15.78	24.50
Clang	1.16	-	-	-0.94	-	-

Tabell Genomsnittlig förändring av exekveringstid mot ingen restrict i procent. Positiva värden innebär en minskning av exekveringstiden medan negativ innebär en ökning.

Slutsats

Alla tre kompilatorer har implementerat pekaroptimeringar som utförs vid användningen av restrict. IBM har generellt lyckats bäst när det kommer till prestandaökningen, vilket inte var oväntat då de har mest kännedom om arkitekturen.

Eftersom restrict kan försämra prestanda så är det inte rekommenderat som programmerare att lägga till restrict i sin kod hur som helst. Vid arbete med en befintlig kodbas rekommenderar vi att identifiera loopar som ser ut att vara lämpliga kandidater för restrict och undersöka hur prestandan påverkas.

För kompilatorskrivare är restrict lite av ett dilemma. Prestandan kan påverkas negativt och själva optimeringen kan vara svår att implementera. Eftersom det inte alltid lönar sig att ersätta en läsning från minnet till en registerläsning behöver kompilatorn antagligen en modell av arkitekturen för att kunna schemalägga koden effektivt. Att som kompilatorskrivare behöva lägga mycket tid på en optimering är inte en självklarhet och bör därför noggrant övervägas innan eventuell implementering. ■