$\mathrm{CS4202}$ P02 - Compiler Optimisations

150015673

16 November 2018

Contents

1	Task	4
2	Setup	4
3	Compiler Optimisations Chosen3.1 Loop unrolling (-funroll-loops)3.2 Tail recursion (-foptimize-sibling-calls)3.3 Loop interchange (-floop-interchange)3.4 Tree vectorisation (-ftree-vectorize)	4 4 4 5
4	Test-Suite 4.1 2d_mmult.c 4.2 1d_mmult.c 4.3 arr_incr.c 4.4 bst.c 4.5 mymap.c 4.6 ptest.c 4.7 reduce.c 4.8 sum.c 4.9 trfact.c	5 5 5 5 5 5 6 6 6 6 6
5	Manually Implemented Optimisations 5.1 Inline Functions 5.2 Loop Unrolling Data Collection	6 6 6 7
7	Results and Analysis 7.1 Compiler Optimisations 7.1.1 Loop Interchange 7.1.2 Vectorisation 7.1.3 Loop Unrolling 7.1.4 Sibling and Tail Recursive Calls 7.2 Manual Optimisations 7.2.1 Inline functions 7.2.2 Loop Unrolling	7 7 9 11 13 14 14 16
8	Given More Time	17
9	Conclusion	18
Re	eferences	18
Α	Assembler Code A.1 Vectorised A.2 Sibling and Tail Recursive Calls	19 19 22

В	Plot	ts	2	4
	B.1	Compiler Optimisations	2	4
	B.2	Manual Optimisations	2	7

1 Task

The aim of this practical was to examine and evaluate 4-5 GCC optimisations using a personally developed test-suite in order to gain a better understanding of computer architecture concepts and the effect of compiler optimisations.

2 Setup

- CPU: Intel Core i5-6500 @ 3.20GHz
- RAM: 8GiB DDR4
- OS: Fedora 28 (64-bit, kernel 4.18.16-200.fc28.x86_64)
- GCC: 8.2.1 20181105

3 Compiler Optimisations Chosen

I chose the optimisations below because they each are fairly simple, yet exploit different things, e.g. sequential execution or vector registers. I also chose them because although they are simple, they should allow for some speedup, e.g. loop unrolling should speed up a loop-heavy program, and vectorisation should speed up array-heavy programs.

Optimisation flags require a value for the -0 flag in order to work¹. I chose to use -0 to avoid any additional optimisations the compiler had.

3.1 Loop unrolling (-funroll-loops)

Branching and branch-prediction is expensive. Since loops repeat for a given condition, e.g. i < 10, we can limit the number of times that condition has to be checked by unrolling the loop. This leads to speedups both because of the fewer branches, but also because there are more sequential steps to execute, which is something processors are good at. The main downside is that the code size increases.

3.2 Tail recursion (-foptimize-sibling-calls)

Tail-recursive functions differ from recursive functions by doing the accumulation of the recursion as part of the parameters, instead of when the recursive call returns. This means that the compiler can replace the tail-recursive call with a GOTO statement or similar, thereby saving stack space since there is no need for recursively calling the function.

3.3 Loop interchange (-floop-interchange)

Loop interchange is swapping the inner and outer loops around. Doing this can improve exploitation of spatial locality and might also expose other optimisations, e.g. vectorisation.

¹(Using the GNU Compiler Collection (GCC): Optimize Options, 2018)

3.4 Tree vectorisation (-ftree-vectorize)

GCC uses trees as an internal representation of the compiled programs. This flag allows GCC to use SIMD vector instructions to optimise the program code. This should speed up code which does a lot of array manipulation.

4 Test-Suite

I wrote a variety of programs. Not all of them target the same optimisation, and it is possible that some of them cannot be optimised using one of the chosen optimisations. In order to achieve a measurable run-time, I defined the 'loops.h' file which contains the number of iteration each program should run the function to measure. Apart from the matrix multiplication, I set this to 100,000 iterations. To ensure that the compiler would not simply optimise away the program, the output is printed in programs where there is one output, and an element of the array is printed in programs whose result is an array.

4.1 2d_mmult.c

A simple, naïve 2d-array matrix multiplication of 1024×1024 matrices. This program should be optimisable through loop unrolling, loop interchange, and vectorisation. The dimension of $1024 = 2^{10}$ was chosen with the idea that it might fit nicer into vector registers.

Since matrix multiplication of matrices this large is already computationally expensive, only 5 iterations were used instead of the default 100,000.

4.2 1d_mmult.c

A simple, naïve matrix multiplication of 1024×1024 matrices, this time using 1d-arrays instead of 2d-arrays. This was done to examine if the compiler is better at optimising 1d-arrays than 2d-arrays.

Since matrix multiplication of matrices this large is already computationally expensive, only 5 iterations were used instead of the default 100,000.

4.3 arr_incr.c

Increments all the numbers in an array by a given constant. The idea of this program was mainly to target vectorisation although loop unrolling should improve it as well.

4.4 bst.c

A small, simple Binary Search Tree (BST) implementation. Values are inserted into the BST and it is traversed in-order. This was done to target recursive function optimisations.

4.5 mymap.c

An implementation of a 'map' function over integers. This program's purpose was to examine if/how well the compiler can optimise function pointers.

4.6 ptest.c

A simple primality test². The while loop might be unrollable. However, I mainly chose to include a primality test because it should be difficult to optimise automatically and so I wanted to see if the compiler could optimise it.

4.7 reduce.c

The 'reduction' of an array, i.e. the sum of all its elements. The array in this file is an integer array with $2^{14} = 16384$ elements. This size was, similar to the matrices' dimension, chosen to potentially be easily divisible into vector registers.

4.8 sum.c

Sums all the numbers from 1 to 10,000 inclusive. The main optimisation target of this program was loop unrolling.

4.9 trfact.c

Calculates the factorial of the given number using tail-recursion. To achieve the maximum number of calls 20!, the largest factorial that fits in an unsigned long long, was used. This program targets tail-recursive optimisations.

5 Manually Implemented Optimisations

I manually implemented two optimisations: inline functions and loop unrolling. The source for these can be found in man-inline-src and man-funroll-src respectively. To measure how effective the manual implementations were, I compiled the files without any optimisations.

5.1 Inline Functions

All code that could be moved to main was moved there. For the tail-recursive factorial, I also transformed the tail-recursive function to a loop. This is an additional optimisation, however it was something I did because it seemed logical: you can achieve the tail-recursive function inline by converting it into a loop. Apart from this, and the BST implementation, all function code was moved to main where possible.

The advantage of this should be that there are fewer operations due to avoiding the overhead of calling and returning from functions. The disadvantage is that the code might be less legible, and will likely not adhere to the "Don't Repeat Yourself" (DRY) principle.

5.2 Loop Unrolling

Where applicable, I manually unrolled loops in my programs using the generalised version of Duff's Device³. This increases the code size and is almost the exact opposite of DRY, but it might increase execution speed due to there being less branching. The code size and readability suffers especially heavily when the two matrix multiplication programs are unrolled.

 3 (Holly, 2005)

 $^{^{2}}$ taken from (*Primality test - Wikipedia*, 2018)

6 Data Collection

All programs were compiled and run on the scratch space of the PC used, to remove any delay there might be by having the files on the synchronised user drive. A slight variation in timing was observed through a couple of manual runs of the programs. To compensate for this variation, each program was run 10 times. The data from each flag was put in separate CSV-files with the program name indicating whether the flag was en- or disabled.

For timing I initially used /usr/bin/time. However, this times in lower resolution than the builtin time shell keyword. Therefore, I decided to use the latter. To output to CSV, TIMEFORMAT was set to "\${prog_name},%4U". Additionally, displaying outputs takes time and completely clutters the terminal when running 100,000 iterations of the same function. To combat this, stdout was piped to /dev/null. The benchmark.sh script takes around 45-60 minutes to complete.

7 Results and Analysis

All plots include error bars representing the standard error of the mean (SEM).

7.1 Compiler Optimisations

7.1.1 Loop Interchange

For loop interchange, the main difference was observed in terms of matrix multiplication. This was expected, as loop interchange is a common matrix multiplication optimisation. For the other programs, it had no effect⁴.

 4 see plot B.1: 13



Figure 1: Loop-interchange on matrix multiplication.

For the 2d-array matrix multiplication, the time seems to have increased slightly with the flag enabled. However, this seems to be a measurement error: looking at the assembler code reveals that the two files are identical, meaning that the compiler did not find a way to interchange the loops.

	2d.	_mmult_fno_loop_interchange.s ×			2d_mmult_floop_interchange.s	×
mmult:			mmult:		*	
.LFB25:			.LFB25:			
	.cfi_st	artproc		.cfi_s	tartproc	
	testl	%ecx, %ecx		testl	%ecx, %ecx	
	jle	.L17		jle	.L17	
	pushq	%r12		pushq	%r12	
	.cfi_de	ef_cfa_offset 16		.cfi de	ef_cfa_offset 16	
	.cfi_of	fset 12, -16		.cfi o	ffset 12, -16	
	pushq	%rbp		pushq	%rbp	
	.cfi_de	ef_cfa_offset 24		.cfi de	ef_cfa_offset 24	
	.cfi_of	fset 6, -24		.cfi o	ffset 6, -24	
	pushq	%rbx		pushq	%rbx	
	.cfi_de	ef_cfa_offset 32		.cfi de	ef cfa offset 32	
	.cfi_of	fset 3, -32		.cfi_o	ffset 3, -32	
	movq	%rdi, %r10		movq	%rdi, %r10	
	movq	%rsi, %r11		movq	%rsi, %r11	
	leal	-1(%rcx), %r9d		leal	-1(%rcx), %r9d	
	leaq	8(%rdi,%r9,8), %r12		leaq	8(%rdi,%r9,8), %r12	
	leaq	4(,%r9,4), %rbp		leag	4(,%r9,4), %rbp	
	movl	\$0, %ebx		movl	\$0, %ebx	
	jmp	.L11		imp	.L11	
.L15:			.L15:	5 1		
	movq	%rcx, %rax		mova	%rcx. %rax	
.L12:			.L12:		,	
	movq	(%rdx,%rax,8), %rcx		mova	(%rdx.%rax.8), %rcx	
	movl	(%rcx,%rdi), %ecx		movl	(%rcx.%rdi). %ecx	
	imull	(%r8,%rax,4), %ecx		imull	(%r8.%rax.4). %ecx	
	addl	%ecx, %esi		addl	%ecx. %esi	
	leaq	1(%rax), %rcx		lead	1(%rax). %rcx	
	cmpq	%r9, %rax		cmpa	%r9. %rax	
	jne	.L15		ine	.L15	
	movq	(%r10), %rax		mova	(%r10). %rax	
	movĺ	%esi, (%rax,%rdi)		movl	%esi, (%rax.%rdi)	
	addq	\$4, %rdi		addg	\$4, %rdi	
	cmpq	%rbp, %rdi		cmpa	%rbp. %rdi	
	je	.L13		ie	.L13	
.L14:			.L14:	2		
	movq	(%r11), %r8		movq	(%r11), %r8	
	movq	%rbx, %rax		movq	%rbx, %rax	
	movl	\$0, %esi		movl	\$0, %esi	
	jmp	.L12		jmp	.L12	
.L13:			.L13:	5 1		
	addq	\$8, %r10		addq	\$8, %r10	
	addq	\$8, %r11		addq	\$8, %r11	
	cmpq	%r12, %r10		cmpq	%r12, %r10	
	je	.L9		ie	.L9	1
.L11:	-		.L11:	,		
	movl	\$0, %edi		movl	\$0, %edi	
	jmp	.L14		jmp	.L14	
.L9:			.L9:	2 ····P		
	popq	%rbx		popq	%rbx	
	.cfi_de	ef_cfa_offset 24		.cfi de	ef cfa offset 24	

Figure 2: The assembler code for 2d_mmult with and without optimisation (right and left respectively) is identical.

This seems to indicate that the compiler does not only find it easier to optimise 1d-arrays, it might find it impossible to optimise 2d-arrays. Looking at vectorisation further strengthens this hypothesis.

7.1.2 Vectorisation

For vectorisation, several programs are sped up. The most drastic improvements are on the **reduce** and **arr_incr** programs where the execution time is almost halved. 1d-matrix multiplication





Figure 3: Both reduce and arr_incr are sped up significantly by vectorisation.

It makes sense that reduce and arr_incr are the programs on which vectorisation has the greatest effect: arr_incr is adding a constant to each element of a vector, and reduce can be split

⁵see plot B.1: 14

⁶see assembler A: 10

into multiple vectors which can then be added together until all that remains is a vector register with the total sum in it⁷. When examining vectorisation, I came across an interesting feature of x86 vector instructions: in both reduce, but surprisingly also in bst, 0 is derived by using pxor on the same register⁸ instead of simply using the constant \$0. By searching online, I found that this achieves greater performance, as the pxor is evaluated at the register rename stage⁹. However, as zeroing it is not the bulk of the bst program, there is no speed-up to be gained from doing so.

An optimisation which affected similar programs to the ones sped up with vectorisation is loop unrolling.

7.1.3 Loop Unrolling

With loop unrolling, the programs that were affected by vectorisation were also affected by loop unrolling, with sum and my_map being affected as well. This is likely due to the fact that vectorisation targets large arrays, which will typically be iterated over in big loops, hence being suitable for loop unrolling as well. My theory for why my_map was optimised here and not through vectorisation is that the compiler cannot know that the pointer to the function won't change and so it does not want to replace it with the function. For sum my theory is that vectorising it would require vectors whose elements were the numbers of the loop and that these would be more expensive to construct than the speedup of having them would win back. When comparing the results in Figure 4 and Figure 3, it becomes clear that although loop unrolling helps, it is not as fast as hardware tools like vector registers. Programs that optimised well using vectorisation. Since the sum program is purely loop-based, its execution time is greatly decreased: from $\approx 0.65s$ to $\approx 0.30s$. The program my_map does not improve its execution time much. However, this is still an improvement compared to no optimisation with vectorisation.

As expected, the past three optimisations have not affected the recursive programs, i.e. **bst** and **trfact**.

⁷see assembler A: 11

 $^{^{8}}$ see assembler A: 11 and 12

 $^{^9(\}mathrm{Fog},\,2018)$ and (Cordes & balajimc55, 2017)





Figure 4: Loop unrolling speeds up the same programs as vectorisation, and some other programs as well.

7.1.4 Sibling and Tail Recursive Calls

When optimising sibling and tail recursive calls, both bst and trfact improve their execution time. Other programs remain unaffected¹⁰ which was expected as they do not use any recursive calls.

•	tr	fact_foptimize_sibling_calls.s × >	•	tr	fact_fno_optimize_sibling_calls.s ×
	.file	"trfact.c"	1	.file	"trfact.c"
	.text			.text	
	.globl	fact	1	.globl	fact
	.type	fact, @function	1	.type	fact, @function
fact:			fact:		
.LFB24:			.LFB24:		
	.cfi_st	artproc		.cfi_sta	artproc
	movq	%rsi, %rax		movq	%rsi, %rax
	testq	%rdi, %rdi	1	testq	%rdi, %rdi
	je	.L4	1	jne	.L7
.L2:				ret	
	imulq	%rdi, %rax	.L7:		
	subq	\$1, %rdi		subq	\$8, %rsp
	jne	.L2		.cfi_de	f_cfa_offset 16
.L4:			1	imulq	%rdi, %rax
	ret		1	movq	%rax, %rsi
	.cfi_en	dproc		subq	\$1, %rdi
.LFE24:		e		call	fact
	.size	fact,fact		addq	\$8, %rsp
	.sectio	n .rodata.str1.1,"aMS",@progb		.cfi_de	f_cfa_offset 8
1			1	ret	
.LC0:		10]]	1.5524	.cti_en	aproc
	.string	"%llu\n"	.LFE24:		6+ 6+
	.lext	main		.size	Tacl,Tacl redete etrl 1 HeMCH Oprochite
	.globi	main ofunction	1	.sectio	n .rodata.stri.i, "ams",@progbits,
moin.	.type	main, @iunction	1.00.		
IIId III:			. LC0:	string	"%]]u\p"
.LFDZJ:	ofi of	artarac	1	.string	%ctu\II
	nucha	%rhv		alobl	main
	cfi de	f cfa offset 16		type	main Ofunction
	.cir_de	_cla_oliset to	d	. cype	main, eranction

Figure 5: The tail-recursive function in trfact has been optimised into a loop by the compiler.

Examining the assembler code reveals that the compiler has optimised trfact as expected: the tail-recursive function has been transformed into a loop. Looking at the assembler code for bst reveals that the speedup is gained by re-ordering some of the instructions¹¹. There are still recursive calls to the function, they just happen later in the function.

 $^{^{10}}$ see plots in B.1: 15

¹¹see assembler A: 12



Figure 6: Both trfact and bst improve their execution time when using -foptimize-sibling-calls.

As can be seen in the figure above, the reduction in execution time is not big: less than 0.01s. Despite this, since no other optimisation has improved the execution time of these programs, I would claim that a speedup is still significant. And this indicates that recursive function calls are difficult to optimise.

The compiler optimisations were effective in varying ways. For comparison, a couple of manual optimisations were implemented.

7.2 Manual Optimisations

7.2.1 Inline functions

For inline functions, some programs were more affected than others. Most programs were unaffected¹². The program that improved the most was mymap. This is likely because I knew, contrary to the compiler, that the function pointer would not change and as such could substitute its definition into the main loop.

 $^{^{12}\}mathrm{see}$ plots B.2: 16



Figure 7: By using knowledge unavailable to the compiler, mymap has improved execution time.

The tail-recursive factorial implementation is also improved. However, this is because of the manual 'inline' transformation to a loop. It shows that this improvement is fairly simple to do and does improve performance, but it is not simply moving the function inline.

The readability of the code was not too heavily affected by moving the functions, that could be moved, inline.

7.2.2 Loop Unrolling

When comparing the results for mmult, it initially seems like the manual optimisation was better. However, taking a look at the y-ticks reveals that while the proportional increase might be slightly better, the time-scale has tripled. This is likely due to the compiler unrolling the loops in assembler compared to the unrolling in C which is a higher level and hence might be slower. The C unrolled version is also mostly impossible to read and much longer than the original program. What is interesting is that the 2d-array implementation is still not affected. This suggests that the problem lies elsewhere, perhaps in the memory access of 2d-arrays.



Figure 8: Comparing manual loop unrolling to compiler unrolling for mmult.

The time-scale changing remains true for the other programs as well. And for mymap, the manual loop unrolling even increased the execution time. For most of the manually unrolled programs, it is also the case that the new proportion is close to, but not quite as good as, the compiler's optimisation.



Figure 9: Comparing manual loop unrolling to compiler unrolling for various programs.

8 Given More Time

Given more time, I would have liked look further into why the 2d-array matrix multiplication is not being optimised. I would start off by examining if this was due to multiple optimisations being required on top of each other, or whether the problem was the 2d-arrays themselves.

I would also have liked to examine the impact of my chosen optimisation in "the grand scheme of things", i.e. enabled a general optimisation level, specifically disabled a single optimisation, and then compared the timings between all optimisations being enabled and all optimisations except from one being enabled. This would have given an insight into how much the chosen optimisations

affect, and are affected by, other optimisations being present.

9 Conclusion

In this practical, I constructed various small programs for testing compiler optimisations and evaluated a subset of these. For comparison, I then implemented some optimisations by hand and compared the results with the original code, and with the compiler. By doing so, I have confirmed that compilers are usually better at optimising programs than humans, discovered some peculiarities about computer architecture, and confirmed that primality testing is hard to optimise (I did not manage to manually find a method for unrolling the loop in the primality test).

This practical has helped me gain a better and more thorough understanding of compiler optimisations, the need for these, and which programs can be optimised how and why/why not.

References

- Cordes, P., & balajimc55. (2017, May 23). What is the best way to set a register to zero in x86 assembly: xor, mov or and? Retrieved from https://stackoverflow.com/questions/33666617/ what-is-the-best-way-to-set-a-register-to-zero-in-x86-assembly-xor-mov-or-and ([Online; accessed Nov. 2018])
- Fog, A. (2018). The microarchitecture of intel, and and via cpus: An optimization guide for assembly programmers and compiler makers. Technical University of Denmark. Retrieved from https://www.agner.org/optimize/microarchitecture.pdf
- Holly, R. (2005, Aug 1). A Reusable Duff Device. Retrieved from http://www.drdobbs.com/ a-reusable-duff-device/184406208?queryText=%2522duff%2527s%2Bdevice%2522 ([Online; accessed Nov. 2018])
- Primality test Wikipedia. (2018, Oct). Retrieved from https://en.wikipedia.org/wiki/ Primality_test#Pseudocode ([Online; accessed Nov. 2018])
- Using the GNU Compiler Collection (GCC): Optimize Options. (2018, Jul). Retrieved from https://gcc.gnu.org/onlinedocs/gcc-8.2.0/gcc/Optimize-Options.html# Optimize-Options ([Online; accessed Nov. 2018])

A Assembler Code

A.1 Vectorised

mmult:			mmult:		
.LFB25:			.LFB25:		
	.cfi_sta	artproc		.cfi_st	artproc
	testl	%ecx, %ecx		testl	%ecx, %ecx
	jle	.L17		jle	.L17
	pushq	%r12		pushq	%r12
	.cfi_de	f_cfa_offset 16		.cfi_de	f_cfa_offset 16
	.cfi_of	fset 12, -16		.cfi_of	fset 12, -16
	pushq	%rbp		pushq	%rbp
	.cfi_de	f_cfa_offset 24		.cfi_de	f_cfa_offset 24
	.cfi_of	fset 6, -24		.cfi_of	fset 6, -24
	pushq	%rbx		pushq	%rbx
	.cfi_de	f_cfa_offset 32		.cfi_de	f_cfa_offset 32
	.cfi_of	fset 3, -32		.cfi_of	fset 3, -32
	movq	%rdi, %r10		movq	%rdi, %r10
	movq	%rsi, %rll		movq	%rsi, %r11
	leal	-1(%rcx), %r9d		leal	-1(%rcx), %r9d
	leaq	8(%rdi,%r9,8), %r12		leaq	8(%rdi,%r9,8), %r12
	leaq	4(,%r9,4), %rbp		leaq	4(,%r9,4), %rbp
	movl	\$0, %ebx		movl	\$0, %ebx
	jmp	.L11		jmp	.L11
.L15:			.L15:		
	movq	%rcx, %rax		movq	%rcx, %rax
.L12:			.L12:		
	movq	(%rdx,%rax,8), %rcx		movq	(%rdx,%rax,8), %rcx
	movl	(%rcx,%rdi), %ecx		movl	(%rcx,%rdi), %ecx
	imull	(%r8,%rax,4), %ecx		imull	(%r8,%rax,4), %ecx
	addl	%ecx, %esi		addl	%ecx, %esi
	leaq	1(%rax), %rcx		leaq	1(%rax), %rcx
	cmpq	%r9, %rax		cmpq	%r9, %rax
	jne	.L15		jne	.L15
	movq	(%r10), %rax		movq	(%r10), %rax
	movl	%esi, (%rax,%rdi)		movl	%esi, (%rax,%rdi)
	addq	\$4, %rdi		addq	\$4, %rdi
	cmpq	%rbp, %rdi		cmpq	%rbp, %rdi
	je	.L13		je	.L13
.L14:			.L14:		
	movq	(%r11), %r8		movq	(%r11), %r8
	movq	%rbx, %rax		movq	%rbx, %rax
	movl	\$0, %esi		movl	\$0, %esi
	jmp	.L12		jmp	.L12
.L13:			.L13:		
	addq	\$8, %r10		addq	\$8, %r10
	addq	\$8, %r11		addq	\$8, %r11
	cmpq	%r12, %r10		cmpq	%r12, %r10
	je	.L9		je	.L9
.L11:			.L11:		
	movl	\$0, %edi		movl	\$0, %edi
	jmp	.L14		jmp	.L14
.L9:			.L9:		
	popq	%rbx		popq	%rbx
	.cfi_de	f_cfa_offset 24		.cfi_de	t_cta_offset 24

Figure 10: 2d-array matrix multiplication is unaffected by vectorisation.

		reduce_ftree_vectorize.s	×
main:			
.LFB24:	- 6		
	.CT1_ST	artproc	
	pusnq	%rbp f ofo offoot 16	
	.cri_de	feet 6 -16	
	.cri_01	sec 0, -10 seen sehn	
	cfi de	f cfa register 6	
	nusha	%r14	
	pusha	%r13	
	pusha	%r12	
	pusha	%rbx	
	.cfi of	fset 14, -24	
	.cfi_of	fset 13, -32	
	.cfi_of	fset 12, -40	
	.cfi_of	fset 3, -48	
	subq	\$65536, %rsp	
	movq	%rsp, %r14	
	movl	\$100000, %r13d	
	movl	\$0, %r12d	
	leaq	65536(%r14), %rbx	
.L2:			
	movq	%r14, %rax	
	pxor	%xmm1, %xmm1	
	pxor	%xmm4, %xmm4	
.L3:	moudau	(9-may) 9-ymm0	
	movaqu	(%rax), %xmm0 %xmm4_%xmm2	
	ncmpatd	3XIIIII4, 3XIIIII∠ %vmmΩ %vmmϽ	
	movdga	∞xmmΩ %xmm3	
	nunnckl	da %xmm2 %xmm3	
	padda	%xmm3, %xmm1	
	punpckh	da %xmm2.%xmm0	
	paddq	%xmm0, %xmm1	
	addq	\$16, %rax	
	cmpq	%rax, %rbx	
	jne	.L3	
	movdqa	%xmm1, %xmm0	
	psrldq	\$8, %×mm0	
	paddq	%xmm1, %xmm0	
	movq	%xmm0, %rax	
	addq	%rax, %r12	
	movq	%r12, %rsi	
	movl	\$.LCO, %edi	
	movl	\$⊍, %eax	
	call		
	subl	\$1, %F130	
	Jne	•LZ	
	lear	au, acda 22(Sebb) Seco	
	nong	- յշ (ծլ նթ), ծլ sp %rhx	
	popa	%r12	
	popq	%r12	

Figure 11: reduce optimised using vector registers.



Figure 12: Using pxor to set left and right child to 0 instead of using the constant \$0.

A.2 Sibling and Tail Recursive Calls

ture court of mation	E 23	25	search:
search:	24	26	.CFB23:
.LFB25:	26	28 «	testq %rdi, %rdi
.cfi_startproc	27	29 «	je .L6
movq %rdi, %rax	> 28	30	subq \$8, %rsp
ie .L8	> 30	31	movl (%rdi). %edx
movl (%rdi), %edx	31	33 «	movq %rdi, %rax
cmpl %esi, %edx	≫ 32	34	cmpl %esi, %edx
jne .L5	> 33	35 «	je .L3
.16:	> 35	30	mova 8(%rdi). %rdi
movq 16(%rax), %rax	36	38	call search
.L7:	37	39	.L3:
testq %rax, %rax	38	40	addq \$8, %rsp
]e .L8 mov] (%rax) %edx	39	41	.cti_remember_state
cmpl %esi, %edx	40	43	ret
je .L8	42	44 «	.L5:
.L5:	43	45 «	.cfi_restore_state
cmpl %edx, %esi	» 44 45	46	movq 16(%rd1), %rd1
movg 8(%rax), %rax	45	47	imp L3
jmp .L7	47	49	.L6:
.L8:	48	50	.cfi_def_cfa_offset 8
ret	49	51	movq %rdi, %rax
.CT1_endproc	50	52	ret officendarios
.size search search	52	54	LFE25:
.globl insert	53	55	.size search,search
.type insert, @function	54	56	.globl insert
insert:	55	57	.type insert, @function
.LFB26:	56	58	insert:
	(a)	
insert	55	5.8	insert
.LFB26:	56	59	.LFB26:
ofi startares			
.cri_startproc	57	60	.cfi_startproc
testq %rdi, %rdi	57 ≫ 58	60 61 «	.cfi_startproc pushq_%rbx
.ci <u>i Startproc</u> testq %rdi, %rdi je .L16 pusha %rby	57 >> 58 59 60	60 61 « 62	.cfi_startproc pushq %rbx .cfi_def_cfa_offset 16 .cfi_offcet 316
.cii_startproc testq %rdi, %rdi je .L16 pushq %rbx .cfi def cfa offset 16	57 >> 58 59 60 61	60 61 « 62 63 64 «	.cfi startproc pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi, %rdi
.cii_Startproc testq %rdi, %rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16	57 >> 58 59 60 61 62	60 61 « 62 63 64 « 65	.cfi startproc pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi, %rdi je .L17
.cii_Startproc testq %rdi, %rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movg %rdi, %rbx	57 >> 58 59 60 61 62 >> 63	60 61 « 62 63 64 « 65 66	.cfi startproc pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi,%rdi je .L17 movg %rdi,%rbx
.cli_Startproc testq %rdi, %rdi je .ll6 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -l6 movq %rdi, %rbx cmpl %esi, (%rdi) iee Ll2	57 >> 58 59 60 61 62 >> 63 64 >> 65	60 61 « 62 63 64 « 65 66 67 67	.cfi startproc pushq %rbx .cfi def cfa offset 16 .cfi_offset 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) ice L15
.cli_Startproc testq %rdi, %rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L12 mova &(%rdi). %rdi	57 58 59 60 61 62 53 64 59 60 61 62 56 66	60 61 « 62 63 64 « 65 66 67 68 « 69	.cfi startproc pushq %rbx .cfi def cfa offset 16 .cfi_offset 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L15 mova 8(%rdi). %rdi
.cii_startproc testq %rdi, %rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L12 movq 8(%rdi), %rdi call insert	57 58 59 60 61 62 53 64 59 63 64 54 56 67	60 61 « 62 63 64 « 65 66 67 68 « 69 70	.cfi startproc pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L15 movq 8(%rdi), %rdi call insert
.cii_Startproc testq %rdi, %rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi, %rbx cmpl %esi, (%rdi) jge.L12 movq 8(%rdi), %rdi call insert movq %rax, 8(%rbx)	57 ≫ 58 59 60 61 62 ≫ 63 64 ≫ 65 66 67 68	60 61 « 62 63 64 « 65 66 67 68 « 68 70 70 71	.cfi startproc pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L15 movq 8(%rdi), %rdi call insert movq %rax, 8(%rbx)
.cii_Startproc testq %rdi, %rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi, %rbx cmpl %esi, (%rdi) jge.L12 movq 8(%rdi), %rdi call insert movq %rax, 8(%rbx) movq %rbx, %rax	57 58 59 60 61 62 63 64 55 66 67 68 69 20 20 20 20 20 20 20 20 20 20	60 61 « 62 63 64 « 65 66 67 68 « 69 70 71 72 73 «	.cfi startproc pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge.L15 movq 8(%rdi), %rdi call insert movq %rax, 8(%rbx) movq %rbx, %rax
.cli_startproc testq %rdi,%rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi,%rbx cmpl %esi,(%rdi) jge .L12 movq &(%rdi),%rdi call insert movq %rax,8(%rbx) movq %rbx,%rax .L10: popg %rbx	57 58 59 60 61 62 63 64 56 66 67 68 69 70 71	60 61 « 62 63 64 « 65 66 67 68 « 69 70 71 72 73 «	.cfi startproc pushq %rbx .cfi def cfa offset 16 .cfi offset 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge.L15 movq %(%rdi), %rdi call insert movq %rbx, %rax .L12: pong %rbx
.cli_startproc testq %rdi, %rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L12 movq &(%rdi), %rdi call insert movq %rax, 8(%rbx) movq %rbx, %rax .L10: popq %rbx .cfi_def_cfa_offset 8	57 ≫ 58 59 60 61 62 ≫ 63 64 ≫ 65 66 67 68 69 70 71 ≫ 72	60 61 « 62 63 64 « 65 66 67 68 « 69 70 70 71 72 73 « 74 75 «	.cfi startproc pusha %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L15 movq %r(di), %rdi call insert movq %rbx, %rax .L12: popq %rbx .cfi_remember state
.cii_startproc testq %rdi, %rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L12 movq &(%rdi), %rdi call insert movq %rax, 8(%rbx) movq %rbx, %rax .L10: popq %rbx .cfi_def_cfa_offset 8 ret	>> 57 >> 58 59 60 61 62 64 >> 65 66 67 68 69 71 >> 72 73	60 61 « 62 63 64 « 65 66 67 68 « 69 70 71 72 73 « 74 75 « 5	.cfi startproc pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L15 movq &(%rdi), %rdi call insert movq %rax, 8(%rbx) movq %rax, 8(%rbx) .cl12: popq %rbx .cfi_remember_state .cfi_def_cfa_offset 8
<pre>.cii_startproc testq %rdi, %rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L12 movq 8(%rdi), %rdi call insert movq %rax, 8(%rbx) movq %rbx, %rax .L10: popq %rbx .cfi_def_cfa_offset 8 ret .L16: fi seteme 2</pre>	\$57 >> 58 59 60 61 62 >> 63 64 >> 65 67 68 9 70 71 >> 70 71 >> 74	60 61 « 62 63 64 « 65 66 66 67 68 « 69 71 72 73 « 74 74 75 76 76	.cfi startproc pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi, %rdi je .l17 movq %rdi, %rbx cmpl %esi, (%rdi) jge.L15 movq %(rdi), %rdi call insert movq %rax, 8(%rbx) movq %rbx, %rax .L12: popq %rbx .cfi_remember_state .cfi_def_cfa_offset 8 ret
<pre>.cii_startproc testq %rdi, %rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi, %rbx cmpl %esi, (%rdi) jge.L12 movq &(%rdi), %rdi call insert movq %rax, 8(%rbx) movq %rbx, %rax .L10: popq %rbx .cfi_def_cfa_offset 8 ret .L16: .cfi_restore 3 movl %esi %edi</pre>	\$57 >> 58 59 60 61 62 >> 63 64 >> 65 66 67 68 9 70 71 >> 72 73 >> 74 75	60 61 « 62 63 64 « 66 67 68 « 69 70 71 72 73 « 74 75 « 76 78 «	.cfi startproc pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge.L15 movq &(%rdi), %rdi call insert movq %rax, 8(%rbx) movq %rbx, %rax .L12: popq %rbx .cfi remember state .cfi def_cfa_offset 8 ret .L17: .L17:
.cli_startproc testq %rdi,%rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi,%rbx cmpl %esi,%rdi) jge .L12 movq &{%rdi},%rbx call insert movq %rbx,%rax .L10: popq %rbx .cfi_def_cfa_offset 8 ret .L16: .cfi_restore 3 movl %esi,%edi jmp makeNode	\$7 \$58 \$9 60 61 \$63 64 \$65 66 67 68 69 \$71 \$72 73 \$74 75 76 \$77	60 61 « 63 « 64 « 65 66 67 68 « 69 70 71 72 73 « 75 « 76 75 « 79 80	.cfi_startproc pushd %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L15 movq &rdx, 8(%rbx) movq %rbx, %rax .L12: popq %rbx .cfi_remember_state .cfi_def_cfa_offset 8 ret .L17: .cfi_restore_state movl %esi, %edi
.cli_startproc testq %rdi, %rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L12 movq &(%rdi), %rdi call insert movq %rbx, %rax .L10: popq %rbx, %rax .L10: .cfi_def_cfa_offset 8 ret .cfi_def_cfa_offset 8 ret .L16: .cfi_restore 3 movl %esi, %edi jmp makeNode .L12§	\$7 >> 58 59 60 61 62 >> 63 64 >> 65 66 67 68 69 >> 70 71 >> 72 73 >> 74 76 >> 77 78	60 61 « 63 64 « 65 66 67 68 « 69 70 71 72 73 « 74 75 « 78 « 79 80 81 «	.cfi startproc pushq %rbx .cfi_def_cfa offset 16 .cfi_offset 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L15 movq & 8(%rdi), %rdi call insert movq %rax, 8(%rbx) movq %rax, 8(%rbx) .cfi remember state .cfi_def_cfa_offset 8 ret .L17: .cfi restore state movl %esi, %edi call makeNode
.cli_startproc testq %rdi, %rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi, %rbx cmpl %esi, (%rdi) jge.L12 movq %(%rdi), %rdi call insert movq %rax, 8(%rbx) movq %rbx, %rax .L10: popq %rbx .cfi_def_cfa_offset 8 ret .L16: .cfi_restore 3 movl %esi, %edi jmp makeNode .L12: .cfi_def_cfa_offset 16	\$7 \$58 \$9 60 61 62 \$63 64 \$65 66 67 70 71 \$70 71 \$75 76 \$77 79	60 61 « 62 63 64 « 65 66 67 68 « 69 70 71 72 73 « 74 75 « 76 77 78 « 79 80 81 «	.cfi startproc pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %sesi, (%rdi) jge .L15 movq %rAx, 8(%rbx) movq %rbx .cfi_remember_state .cfi_cfi_cf_cf_offset 8 ret .L12: .cfi_restore_state movl %sesi, %edi call makeNode .L12
<pre>.cii_startproc testq %rdi, %rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L12 movq %(%rdi), %rdi call insert movq %rax, 8(%rbx) movq %rbx, %rax .L10: </pre>	\$7 \$58 \$9 60 61 62 \$63 64 \$65 66 67 88 99 70 71 \$72 73 \$74 75 76 \$79 80 91	60 61 « 62 63 64 « 65 66 66 67 68 « 69 70 71 72 73 « 74 74 75 « 76 77 78 « 79 80 81 « 82 83	
<pre>.cii_startproc testq %rdi,%rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi,%rbx cmpl %esi,(%rdi) jge .L12 movq &{%rdi},%rdi call insert movq %rax,8(%rbx) movq %rbx,%rax .L10: popq %rbx .cfi_def_cfa_offset 8 ret .L16: .cfi restore 3 movl %esi,%edi jmp makeNode .L12: .cfi def_cfa_offset 16 .cfi_offset 3, -16 movq 16(%rdi),%rdi call insert</pre>	57 > 58 59 60 61 62 > 63 64 > 65 66 67 68 69 > 70 71 > 72 73 > 74 75 76 > 77 78 79 80 81 82	60 61 « 63 « 65 66 67 68 « 69 70 70 71 72 73 « 75 « 76 77 78 « 79 80 81 « 82 83 84 85	.cfi_startproc pusha %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L15 movq &rdi, %rbx movq %rdi, %rbx call insert movq %rbx, %rax .L12: popq %rbx .cfi_remember_state .cfi_restore_state .cfi_restore_state movi %esi, %edi call makeNode jmp .L12 .L15: movq 16(%rdi), %rdi call insert
.cli_startproc testq %rdi, %rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L12 movq &(%rdi), %rdi call insert movq %rbx, %rax .L10: popq %rbx .cfi_def_cfa_offset 8 ret .L16: .cfi_restore 3 movl %esi, %edi jmp makeNode .L122; .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rax, 16(%rbx)	\$7 >> 58 59 60 61 62 >> 63 64 >> 65 66 67 68 9 > 70 71 >> 72 73 >> 74 76 >> 77 78 79 80 81 82	60 61 « 63 « 65 66 67 68 « 69 70 71 72 73 « 74 75 « 77 78 « 79 80 81 « 82 83 84 85	.cfi startproc pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi, %rbx cmpl %esi, (%rdi) jg .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L15 movq & 8(%rbi), %rdi call insert movq %rax, 8(%rbx) movq %rax, 8(%rbx) .cfi remember state .cfi remember state .cfi remember state .cfi restore state movq %esi, %edi call makeNode jmp .L12 .L15: movq 16(%rdi), %rdi call insert movq %rax, 16(%rbx)
.cli_startproc testq %rdi, %rdi je .L16 pushq %rbx .cfi_def_cfa offset 16 .cfi_offset 3, -16 movq %rdi, %rbx cmpl %esi, %rdi) jge .L12 movq %rdi, %rbx call insert movq %rdx, %rbx) movq %rbx, %rax .L10: popq %rbx .cfi_def_cfa_offset 8 ret .L16: .cfi_restore 3 movl %esi, %edi jmp makeNode .L12; .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq 16(%rbx) movq %rax, 16(%rbx) movq %rax, 16(%rbx) movq %rbx, %rax	\$7 >> 58 59 60 61 62 >> 63 64 >> 65 66 67 68 69 >> 70 71 >> 72 73 74 76 77 70 80 81 82 84	60 61 « 63 65 66 67 68 « 69 70 70 71 71 72 73 « 74 75 « 78 « 79 80 81 « 83 84 83 84 85 86 87	<pre>.cfi startproc pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L15 movq %(%rdi), %rdi call insert movq %rax, 8(%rbx) movq %rax, 8(%rbx) movq %rax, 8(%rbx) .cfi remember state .cfi_def_cfa_offset 8 ret .L17: .cfi restore state movl %esi, %edi call makeNode jmp.L12 .L15: movq 16(%rdi), %rdi call insert movq %rax, 16(%rbx) movq %rax, %rax</pre>
<pre>.cii_startproc testq %rdi, %rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L12 movq &(%rdi), %rdi call insert movq %rax, 8(%rbx) movq %rax, 8(%rbx) movq %rbx, %rax .L10: popq %rbx .cfi_def_cfa_offset 8 ret .L16: .cfi_restore 3 movl %esi, %edi jmp makeNode .L12: .cfi def_cfa_offset 16 .cfi_offset 3, -16 movq %rbx, %rax jmp .L10 movq %rbx, %rax jmp .L10 ret</pre>	\$7 >> 58 59 60 61 62 >> 63 64 >> 65 66 67 70 71 >> 70 71 >> 70 71 >> 70 73 >> 74 75 76 >> 77 79 80 82 83 >> 85	60 61 « 62 63 64 « 65 67 68 « 67 70 71 72 73 « 74 75 « 76 77 78 « 79 80 81 « 82 83 84 85 86 87 88 «	.cfi startproc pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L15 movq %rAX, 8(%rbx) movq %rAX, 8(%rbx) movq %rAX, 8(%rbx) movq %rAX, 8(%rbx) movq %rbx, %rax .cfi_remember_state .cfi_restore_state movl %esi, %edi call makeNode jmp .L12 .L12: movq %rbx, %rax insert movq %rbx, %rax jmp .L12 .L12: movq %rbx, %rax
<pre>.cii_startproc testq %rdi,%rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_def_sta_, -16 movq %rdi,%rbx cmpl %esi,(%rdi) jge .L12 movq &(%rdi),%rdi call insert movq %rax,8(%rbx) movq %rax,8(%rbx) movq %rbx,%rax .L10: popq %rbx .cfi_def_cfa_offset 8 ret .L16: .cfi_restore 3 movl %esi,%edi jmp makeNode .L12: .cfi_offset 3, -16 movq 16(%rdi),%rdi call insert movq %rax,16(%rbx) movq %rax, 16(%rbx) movq %rax, 16(%rbx) movq %rbx,%rax jmp .L10 .cfi_endproc</pre>	57 > 58 59 60 61 62 > 63 64 > 65 66 67 68 69 > 70 71 > 72 73 > 74 75 76 77 77 78 79 80 81 82 83 84 > 85 86 85 86 87 85 86 86 87 88 88 88 88 88 86 87 88 88 88 88 88 88 88 88 88	60 61 « 63 63 66 66 67 68 « 69 70 71 72 73 « 74 74 75 « 74 75 « 78 80 81 « 82 83 84 85 86 85 88 88 88 88 88 88 88 88 88 88 88 88	
<pre>.cii_startproc testq %rdi,%rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi,%rbx cmpl %esi,%rdi) jge .L12 movq &{%rdi},%rbx call insert movq %rax,8(%rbx) movq %rbx,%rax .L10: popq %rbx .cfi_def_cfa_offset 8 ret .L16: .cfi_restore 3 movl %esi,%edi jmp makeNode .L12: .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rax,16(%rbx) movq %rax,16(%rbx) movq %rax,16(%rbx) movq %rax,16(%rbx) movq %rax,10(%rbx) movq %rbx) movq %rbx) movq %rbx,10(%rbx) movq %rbx) movq %rbx,10(%rbx) movq %rbx) movq %rbx) movq %rbx,10(%rbx) movq %rbx) movq %r</pre>	> 57 >> 58 59 60 61 62 > 63 64 > 65 66 67 68 9 > 70 71 > 72 73 > 74 75 76 > 77 78 79 80 81 823 844 >> 85 86 87 88	60 61 « 63 « 65 66 67 68 « 69 70 71 72 73 « 74 75 « 76 77 78 « 79 80 81 « 82 83 84 85 86 87 85 86 87 89 90 91	<pre>fi startproc pushq %rbx fi_def_cfa_offset 16 fi_offset 3, -16 testq %rdi, %rbi cmpl %esi, (%rdi) je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L15 movq %r(%rdi), %rdi call insert movq %rax, 8(%rbx) movq %rax, 8(%rbx) movq %rax, 8(%rbx) fi remember state fi remember state fi remember state fi remember state fi restore state movq %esi, %edi call makeNode jmp .L12 .L15: movq 16(%rdi), %rdi call insert movq %rax, 16(%rbx) movq %rax, 16(%rbx) movq %rbx, %rax jmp L12 fi_endproc .LFE26: .size insert,insert</pre>
<pre>.cli_startproc testq %rdi,%rdi je .L16 pushq %rbx .cfi_def_cfa offset 16 .cfi_offset 3, -16 movq %rdi,%rbx cmpl %esi,(%rdi) jge .L12 movq &{%rdi,%rbx call insert movq %rax, 8(%rbx) movq %rbx,%rax .L10: popq %rbx .cfi_def_cfa_offset 8 ret .L16: .cfi_def_cfa_offset 8 ret .L16: .cfi_def_cfa_offset 8 ret .L16: .cfi_def_cfa_offset 16 .cfi_def_cfa_offset 16 .cfi_def_cfa_offset 16 .cfi_def_cfa_offset 16 .cfi_def_cfa_offset 16 .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rax, 16(%rbx) movq %rbx,%rax jmp .L10 .cfi_endproc .LFE26: .size insert,insert .section .rodata.strl.1,"aMS",@progbits,1</pre>	>> 58 >> 58 59 60 61 62 >> 63 64 >> 65 66 67 68 970 70 773 773 774 76 777 78 79 80 81 82 84 >> 85 86 87 88 89	60 61 « 63 « 65 66 67 68 « 69 70 70 71 72 73 « 74 75 « 76 77 78 « 79 80 81 « 82 83 84 85 83 84 85 86 87 88 86 87 89 99 90 91 92	<pre>.cfi startproc pushq %rbx .cfi_def_cfa offset 16 .cfi_offset 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L15 movq & 8(%rdi), %rdi call insert movq %rax, 8(%rbx) movq %rax, 8(%rbx) movq %rax, 8(%rbx) movq %rbx, %rax .L12: popq %rbx .cfi remember state .cfi_def_cfa_offset 8 ret .L17: .cfi restore state movl %esi, %edi call makeNode jmp .L12 .L15: movq 16(%rdi), %rdi call insert movq %rax, 16(%rbx) movq %rbx, %rax jmp .L12 .cfi_endproc .LFE26: .size insert,insert .section .rodata.strl.1,"aMS",@progbits,1</pre>
<pre>.cli_startproc testq %rdi,%rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi,%rbx cmpl %esi,%rdi) jge .L12 movq %rdi,%rbx call insert movq %rax, 8(%rbx) movq %rax, 8(%rbx) movq %rax, 8(%rbx) movq %rbx,%rax .L10: .cfi_def_cfa_offset 8 ret .cfi_def_cfa_offset 8 ret .cfi_restore 3 movl %esi,%edi jmp makeNode .L12: .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rbx,%rax jmp .L10 .cfi_endproc .LFE26: .size insert,insert .section .rodata.strl.1,"aMS",@progbits,1 .LC0:</pre>	\$77 >> 58 59 60 61 62 >> 63 64 >> 65 66 67 68 69 >> 70 71 >> 72 73 >> 74 75 76 >> 77 79 80 81 82 83 >> 85 68 87 88 89 90	60 61 « 63 64 « 65 66 67 68 « 70 71 72 73 « 74 75 77 77 78 « 79 80 81 « 82 83 81 « 82 83 84 85 86 87 88 « 82 83 84 85 86 87 99 90 91 92 93	<pre>fi startproc pushq %rbx fi_def cfa offset 16 fi_def start, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L15 movq %(rdi), %rdi call insert movq %rax, 8(%rbx) movq %rax, 8(%rbx) movq %rbx, %rax fi remember state fi_def_cfa_offset 8 ret .L12: popq %rbx cfi restore_state movl %esi, %edi call makeNode jmp .L12 .L15: movq 16(%rdi), %rdi call insert movq %rbx, %rax jmp .L12 .L15: movq %rbx, %rax jmp .L12 .L15: movq %rbx, %rax jmp .L12 fi endproc .LFE26: size insert,insert .section .rodata.strl.1,"aMS",@progbits,1 .LC0:</pre>
<pre>.cii_startproc testq %rdi,%rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_def_sfa_, -16 movq %rdi,%rbx cmpl %esi,%rbx cmpl %esi,%rbx movq %rax,8(%rbx) movq %rax,8(%rbx) movq %rbx,%rax .L10: popq %rbx .cfi_def_cfa_offset 8 ret .L16: .cfi_restore 3 movl %esi,%edi jmp makeNode .L12: .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rax, 16(%rbx) movq %rbx,%rax jmp .L10 .cfi_endproc .LFE26: .size insert,insert .section .rodata.strl.1,"aMS",@progbits,1 .LC0: .string "%d\n" test</pre>	> 57 > 58 59 60 61 62 > 65 66 67 68 9 71 > 72 73 74 75 77 78 79 80 81 83 84 > 85 86 87 90 91 91	60 61 « 63 64 « 65 66 67 68 « 70 71 72 73 « 74 77 75 « 77 77 78 « 79 80 81 « 82 83 81 « 82 83 84 85 86 87 88 « 84 85 86 87 89 90 91 92 93 94	<pre>fi startproc pushq %rbx fi_def cfa offset 16 fi offset 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L15 movq %rkdi), %rdi call insert movq %rax, 8(%rbx) movq %rbx, %rax fi remember state fi def cfa_offset 8 ret .L12: fi restore state movl %esi, %edi call makeNode jmp .L12 .L15: movq %rbx, %rax jmp .L12 .L15: movq %rbx, %rax jmp .L12 .L15: movq %rbx, %rax jmp .L12 .L15: size insert,insert .section .rodata.strl.1,"aMS",@progbits,1 .LC0: string "%d\n" tavt</pre>
<pre>.cii_startproc testq %rdi,%rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi,%rbx cmpl %esi,%rdi) jge .L12 movq &{%rdi},%rbx cmpl %esi,%rdi call insert movq %rax,8(%rbx) movq %rbx,%rax .L10: popq %rbx .cfi_def_cfa_offset 8 ret .L16: .cfi_restore 3 movl %esi,%edi jmp makeNode .L12% .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rax,16(%rbx) movq %rbx,%rax jmp .L10 .cfi_endproc .LFE26: .size insert,insert .section .rodata.strl.1,"aMS",@progbits,1 .LC0: .string "%d\n" .text .dlobl inorder</pre>	> 57 >> 58 59 60 61 62 > 63 64 > 65 66 67 68 970 71 73 > 72 73 > 76 > 77 78 79 80 81 82 86 87 88 90 91 92	60 61 « 63 « 65 66 67 68 « 69 70 71 72 73 « 76 77 78 « 79 80 81 « 80 81 « 82 83 84 85 86 87 89 90 91 92 92 93 94 95 96	<pre>.cfi startproc pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi, %rbi cmpl %esi, (%rdi) je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L15 movq &f(%rdi), %rdi call insert movq %rbx, %rax .L12: popq %rbx .cfi_remember state .cfi_remember state .cfi_remember state .cfi_remember state .cfi_restore state .cfi_restore state .cfi_restore state .cfi_restore state .cfi_restore state .cfi_restore state .cfi_restore state .cfi_restore state .cfi_restore state .L17: .cfi_restore state movq %esi, %edi call makeNode jmp .L12 .L15: movq %rax, 10(%rbx) movq %rbx, %rax jmp .L12 .cfi_endproc .LFE26: .size insert,insert .section .rodata.strl.1,"aMS",@progbits,1 .LC0: .string "%d\n" .text</pre>
<pre>.cii_startproc testq %rdi,%rdi je .L16 pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rdi,%rbx cmpl %esi, (%rdi) jge .L12 movq & 8(%rdi), %rdi call insert movq %rax, 8(%rbx) movq %rbx,%rax .L10: popq %rbx .cfi_def_cfa_offset 8 ret .L16: .cfi_def_cfa_offset 8 ret .L16: .cfi_def_cfa_offset 8 ret .L16: .cfi_def_cfa_offset 16 .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rax, 16(%rbx) movq %rbx, %rax jmp .L10 .cfi_endproc .LFE26: .size insert,insert .section .rodata.strl.1,"aMS",@progbits,1 .L00: .string "%d\n" .text .globl inorder .type inorder, @function</pre>	>> 57 >> 58 59 60 61 62 >> 63 64 >> 65 66 67 68 9 712 73 74 75 76 77 78 79 80 81 82 84 > 85 86 90 91 92 92 93 94	60 61 « 63 « 65 66 67 68 « 69 70 71 72 73 « 76 77 73 « 76 78 « 79 80 81 « 82 80 81 « 83 84 85 86 87 87 88 « 89 90 91 92 93 91 92 93 94 95 96 97	<pre>.cfi startproc pushq %rbx .cfi_def_cfa_offset 16 .cfi_offset 3, -16 testq %rdi, %rbx cmpl %esi, (%rdi) je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jg .L15 movq %fordi), %rdi call insert movq %rax, 8(%rbx) movq %rax, 8(%rbx) movq %rbx, %rax .L12: popq %rbx .cfi remember state .cfi_def_cfa_offset 8 ret .L17: .cfi restore_state movl %esi, %edi call makeNode jmp .L12 .L15: movq 16(%rdi), %rdi call insert movq %rax, 16(%rbx) movq %rbx, %rax jmp .L12 .cfi_endproc .LEE26: .size_insert,insert .section .rodata.strl.1,"aMS",@progbits,1 .L00: .string "%d\n" .text .globl inorder .type inorder, @function</pre>
<pre>.cli_startproc testq %rdi,%rdi je .L16 pushq %rbx .cfi_def_cfa offset 16 .cfi_offset 3, -16 movq %rdi,%rbx cmpl %esi, (%rdi) jge .L12 movq & %rdi),%rdi call insert movq %rax, 8(%rbx) movq %rax, 8(%rbx) movq %rbx,%rax .L10: popq %rbx .cfi_def_cfa_offset 8 ret .L16: .cfi_def_cfa_offset 8 ret .L16: .cfi_def_cfa_offset 16 .cfi_def_cfa_offset 16 .cfi_def_cfa_offset 16 .cfi_def_cfa_offset 16 .cfi_def_cfa_offset 16 .cfi_offset 3, -16 movq %rax, 16(%rbx) movq %rax, 16(%rbx) movq %rbx,%rax jmp .L10 .cfi_endproc .LFE26: .size insert,insert .section .rodata.strl.1,"aMS",@progbits,1 .LC0: .string "%d\n" .text .globl inorder .type inorder, @function inorder:</pre>	>> 57 >> 58 59 60 61 62 >> 63 64 >> 65 66 67 68 69 >> 70 >> 72 >> 73 >> 74 76 >> 77 >> 78 79 80 812 82 844 >> 85 87 89 90 92 93 92 93 94	60 61 « 63 64 « 65 66 67 68 « 70 71 72 73 « 74 75 « 78 80 81 « 82 83 84 83 84 85 86 87 89 90 91 92 93 91 92 93 94 95 96	<pre>.cfi startproc pushq %rbx .cfi_def_cfa offset 16 .cfi_def_sfa 3, -16 testq %rdi, %rdi je .L17 movq %rdi, %rbx cmpl %esi, (%rdi) jge .L15 movq & 8(%rdi), %rdi call insert movq %rax, 8(%rbx) movq %rax, 8(%rbx) movq %rax, 8(%rbx) movq %rax, 8(%rbx) .cfi remember state .cfi_def_cfa_offset 8 ret .L17: .cfi restore state movl %esi, %edi call makeNode jmp .L12 .L15: movq %rax, 16(%rbx) movq %rax, 16(%rbx) movq %rbx, %rax jmp .L12 .cfi_endproc .LFE26: .size insert,insert .section .rodata.strl.1,"aMS",@progbits,1 .LC0: .string "%d\n" .text .globl inorder .type inorder, @function inorder:</pre>

(b)



Figure 12: The optimised vs. unoptimised bst assembler code, with the differences highlighted.

B Plots



B.1 Compiler Optimisations

Figure 13: Loop interchange does not make a difference on the non matrix multiplication programs.



Figure 14: 1d-array matrix multiplication is sped up slightly using vectorisation. 2d remains unchanged.



Figure 15: Several programs are unaffected by recursive call optimisations as they do not contain any.



B.2 Manual Optimisations

Figure 16: Most programs were unaffected by manually moving function definitions inline.