Lab Notes Rachel Lew

Image Rotation – ppmtrans
***Note the times here reflect the program being run with ppm images individually and summing the times for 90 and 180 rather than using the run script**

Stage	**Benchmark**	**Time***	**Instruction Fetches**	**Rel to Start**	**Rel to Prev**
1. Initial	Big Image	20s		1.00	1.00
	Small Image	140ms	4.22×10^8	1.00	1.00
2. Compile with -O1 and link with –lcii40-O1	Big Image	12s		0.60	0.60
	Small Image	80ms	3.53×10^8	0.57	0.57
3. Compile with -O2 and link with –lcii40-O2	Big Image	12s		0.60	1.00
	Small Image	80ms	3.53×10^8	0.57	1.00
4. Eliminating UArrays as underlying structure in UArray2 in favor of c-arrays	Big Image	11s		0.55	0.91
	Small Image	80ms	3.10×10^8	0.57	1.00
5. Eliminating UArrays and UArray2s as underlying structure in UArray2b in favor of c-arrays	Big Image	10s		0.5	0.90
	Small Image	60ms	2.10×10^8	0.43	0.75
6. Storing height and width of original array in the closure rather than calling method for each pixel	Big Image	9.71s		0.49	0.97
	Small Image	60ms	1.88×10^8	0.43	1.00
7. Making the height, width, and methods global constants to eliminate closure dereferencing	Big Image	9.69		0.48	0.99
	Small Image	50ms	1.88×10^8	0.36	0.83

1 – 3. Following the specified steps to utilize various levels of compiler optimization

4. Eliminating Hansen's data structures in favor of simple C-Arrays will reduce the number of dereferences necessary to access the elements in the data structure. The number of function calls will also be reduced because each call to a UArray2 function will not require a subsequent call to the corresponding UArray function.

5. Similar to the above change, this significantly reduces the number of dereferences and function calls necessary to access the elements within the array. By representing the UArray2b as a single C-Array, we are removing the levels of indirection required by accessing a UArray2 and subsequently a UArray to retrieve an element.

6. Each time the apply function was called, extra dereferencing of the closure and methods were required to access the height and width information of the original array. This unnecessary dereferencing of a value that could be declared a constant was adding to each iteration of the for loops of the mapping function, and by removing them we saved at least one dereference for each element in the pixmap.

7. In order to avoid the dereferencing that occurs with each iteration of the map function and each call to the apply function, we have made the methods, height, and width of the original pixmap global variables that are accessible to the rotation functions. This did not result in any significant performance improvements because it was at such a low cost in comparison to the at function.

Universal Machine

Stage	**Benchmark**	**Time**	**Instruction Fetches**	**Rel to Start**	**Rel to Prev**
1. Initial	Midmark	9.26s	3.73×10^{10}	1.0	1.0
	Sandmark	230s	---	1.0	1.0
	Adventure	72s	---	1.0	1.0
2. Compile with -O1 and link with –lcii40-O1	Midmark	6.35s	2.99×10^{10}	0.69	0.69
	Sandmark	158s	---	0.68	0.68
	Adventure	51s	---	0.71	0.71
3. Compile with -O2 and link with –lcii40-O2	Midmark	6.41s	2.97×10^{10}	0.69	1.01
	Sandmark	158s	---	0.68	1.00
	Adventure	50s	---	0.69	0.98
4. Implementing mappedIDs as arrays instead of Seq	Midmark	4.93s	2.41×10^{10}	0.53	0.72
	Sandmark	123s	---	0.53	0.77
	Adventure	39s	---	0.54	0.78
5. Removing “unmappedIDs” and creating array of already	Midmark	4.86s	2.37×10^{10}	0.52	0.99
	Sandmark	121s		0.52	0.98

used IDs	Adventure	39s		0.54	1.00
6. Static inline functions for bitpack	Midmark	3.18s	1.23 x10^{10}	0.34	0.45
	Sandmark	79s		0.34	0.65
	Adventure	27s		0.38	0.69
7. Removing UArrays in place of UM_Word* for individual segments	Midmark	2.74s	8.84 x10^{9}	0.30	0.86
	Sandmark	68s		0.30	0.86
	Adventure	23s		0.32	0.85
8. Moving segmentLength computation out of for loop and creating a global variable for number of instructions	Midmark	2.59s	8.11 x10^{9}	0.28	0.95
	Sandmark	64s		0.28	0.94
	Adventure	23s		0.32	1.00
9. Making loadValue register function in um.c	Midmark	2.49s	7.43 x10^{9}	0.27	0.96
	Sandmark	62s		0.27	0.97
	Adventure	21s		0.29	0.91
10. Storing instructions separately to eliminate additional dereference	Midmark	2.47s	7.27 x10^{9}	0.27	0.99
	Sandmark	61s		0.27	0.98
	Adventure	21s		0.29	1.00
11. Inlining resize functions	Midmark	2.47s	7.27 x10^{9}	0.27	1.00
	Sandmark	61s		0.27	1.00
	Adventure	21s		0.29	1.00
12. Removing assert statements from segmentLoad and segmentStore	Midmark	2.32s	6.78 x10^{9}	0.25	0.94
	Sandmark	57s		0.25	0.93
	Adventure	20s		0.28	0.95
13. Making instructions a static constant instead of passing with memorySegments	Midmark	2.25s	6.69x10^{9}	0.24	0.97
	Sandmark	56s		0.24	0.98
	Adventure	19s		0.26	0.95

14. Static inline for execute_instruction	Midmark	830ms	3.83×10^9	0.09	0.37
	Sandmark	20s		0.09	0.36
	Adventure	7s		0.10	0.37

1 – 3. Using different compiler optimization levels significantly reduced our time

4 – 5. These steps encompassed both a change of abstraction and a change of underlying data structures. The latter change reflects the inherent speed up of removing Hansen's data structures in favor of C-Arrays that require fewer dereferences. The former implemented a stack using C-Arrays in order optimize the reuse of IDs. By only keeping track of the IDs that have been mapped and unmapped, we are reducing the number of additional items to store as well as optimizing the reuse of these IDs.

6. Because the Bitpack interface is required for every instruction in setting up the machine, we sacrificed a degree of modularity in favor of performance by moving these simple functions into the um where they are used. This eliminated four function calls for each instruction as they were read in as well as five function calls for each instruction that was executed and needed to be parsed. The significant reduction in instruction fetches as seen in the table is in line with this reasoning.

7. Although the outer structures for mappedIDs and unmappedIDs were updated to be C-Arrays, the underlying structure of mappedIDs, the individual segments themselves, were slowing down the program due to the additional dereferences and function calls required of the UArray interface. Eliminating UArrays in favor of C-Arrays saved immensely on instruction fetches because each function call to manipulate the segments did not necessitate an additional function call to access the element from the UArray. Because UArrays store their length, this step necessitated a change in abstraction as well: storing the length of the segment as the 0th element. We then used a static inline function to address the necessary offset changes for the accessing of elements in memory.

8. The length of the instruction segment required an additional dereference for each iteration of the loop. Because this value remains constant with the exception of when a new program is loaded, it can be stored and reset when said situation arises. This removed a dereference of the memory segments and an additional function call from every iteration of the instruction execution loop.

9. While abstraction is useful for several of the register functions (aka placing them in their own module), loadValue was called with such frequency that the additional overhead of function calling for this simple store within registers was hurting performance more than it was proving a useful abstraction.

10. Instructions being stored as the first element of the memory segments array necessitated two dereferences for each instruction fetch from memory. This could be easily reduced to a single dereference by moving the instructions outside of mappedIDs and making it a separate member of the memory struct. While the 0th element of the mappedIDs is still mapped when the program is loaded, the actual segment containing the instructions is only loaded into the separate instructions array. This achieves the optimization without interfering with the data structure invariants.

11. Inlining resize functions did not prove to speed up the program given the trivial number of times they were called and executed; however, we thought that this may have added savings depending on the original size of our mappedIDs, which affects the calling rate for this function.

12. The assert statements in each call to the segmentedLoad and segmentedStore was adding two function calls to each execution of the function. While these statements would provide useful error messages upon exiting the program, they are not required in the specs as the machine is allowed to segfault if the program asks to load and store memory in segments that are not yet mapped.

13. Moving the instructions entirely outside of the memory segments struct eliminated a dereference with each instruction load.

14. After examining kcachegrind, it was apparent that the actual calling of the function execute_instruction was presenting a greater cost than the actual contents of the function. Inlining it eliminated a function call from each loop iteration of instruction execution while preserving the separation of ideas created by having it as its own function.