Arrays v Pointers, Compiler Generated Code Optimization

Brandon Chin

CSC342 - Instructor: Prof. Izidor Gertner

4/13/2015

Objective:

We are going to create compiler generated assembly code for two separate functions. One which clears an array of some arbitrary size using an index approach, and the other which clears an array of some arbitrary size using a pointer approach. Then we will modify this compiler generated assembly code in order to optimize the duration of the function's execution. Finally, we will organize and compare our results before and after optimization of both methods.

Code Timer

Let's first test the code that will measure the duration of a specific segment of execution. The following code is provided to us:

```
#include <windows.h>
#include <iostream>
using namespace std;

int main()
{
    _int64 ctr1 = 0, ctr2 = 0, freq = 0;
    int acc = 0, i = 0;

    if (QueryPerformanceCounter((LARGE_INTEGER *)&ctr1) != 0)
    {
        // code segment being timed
        for (i = 0; i < 100; i++) acc++;

        // finish timing the code
        QueryPerformanceCounter((LARGE_INTEGER *)&ctr2);

        cout << "Start Value:" << ctr1 << endl;
        cout << "End Value:" << ctr2 << endl;

        QueryPerformanceFrequency((LARGE_INTEGER *)&freq);

        cout << "QueryPerformanceCounter minimum resolution: 1/" << freq << "Seconds," << endl;
        cout << "100 Incerement Time: " << ((ctr2 - ctr1)* 1.0 / freq) * 1000000 << "Microseconds." << endl;
    }
    else
    {
        DWORD dwError = GetLastError();
        cout << "Error value" << dwError << endl;
    }
    system("PAUSE");
    return 0;
}
```

(CodeTimer, main.cpp)

```
Start Value:3925786085259
End Value:3925786085260
QueryPerformanceCounter minimum resolution: 1/1753831Seconds,
100 Incerement Time: 0.57018Microseconds.
Press any key to continue . . .
```

(CodeTimer, main.cpp - output)

Clear Array Using Index Approach

Now we will utilize the code timer that was demonstrated above to analyze the duration of the following clear_array_index function. The function takes an array of any integer size, and overwrites all of its values to be 0. This is done by using another variable which will serve as both a counter to control the for loop and an index for each element in the array. At each iteration of the loop, the counter will increment by one, thus advancing to the next element in the array.

```
void clear_array_index(int ary[], int size);
```

(clear_array_index.h)

```
#include "clear_array_index.h"
void clear_array_index(int ary[], int size)
{
    for (int k = 0; k < size; k++)
        ary[k] = 0;
}
```

(clear_array_index.cpp)

```
#include "clear_array_index.h"
#include <windows.h>
#include <iostream>
using namespace std;

int main()
{
    _int64 ctr1 = 0, ctr2 = 0, freq = 0;
    const int num = 100;
    int i, a[num];

    for (i = 0; i < num; i++)
    {
        a[i] = i + 1;
    }

    if (QueryPerformanceCounter((LARGE_INTEGER *)&ctr1) != 0)
    {
        clear_array_index(a, num);
        QueryPerformanceCounter((LARGE_INTEGER *)&ctr2);

        cout << "Start Value:" << ctr1 << endl;
        cout << "End Value:" << ctr2 << endl;

        QueryPerformanceFrequency((LARGE_INTEGER *)&freq);

        cout << "QueryPerformanceCounter minimum resolution: 1/" << freq << "Seconds," << endl;
        cout << "Clear Array of 100 needs:" << ((ctr2 - ctr1)* 1.0 / freq) * 1000000 << "Microseconds." << endl;
    }

    else
    {
        DWORD dwError = GetLastError();
        cout << "Error value" << dwError << endl;
    }
    system("PAUSE");
    return 0;
}
```

(ClearArrayIndex, main.cpp)

Array Size: 100

```
Start Value:3940059025389
End Value:3940059025390
QueryPerformanceCounter minimum resolution: 1/1753831Seconds,
Clear Array of 100 needs:0.57018Microseconds.
Press any key to continue . . .
```

Array Size: 1000

```
Start Value:3943847949142
End Value:3943847949149
QueryPerformanceCounter minimum resolution: 1/1753831Seconds,
Clear Array of 1000 needs:3.99126Microseconds.
Press any key to continue . . .
```

Array Size: 10000

```
Start Value:3943984425050
End Value:3943984425107
QueryPerformanceCounter minimum resolution: 1/1753831Seconds,
Clear Array of 10000 needs:32.5003Microseconds.
Press any key to continue . . .
```

Array Size: 100000

```
Start Value:3944060355770
End Value:3944060356360
QueryPerformanceCounter minimum resolution: 1/1753831Seconds,
Clear Array of 100000 needs:336.406Microseconds.
Press any key to continue . . .
```

Clear Array Using Pointer Approach

This time, we will use the code timer to analyze the duration of the following clear_array_pointer function. The function once again takes an array of any integer size, and overwrites all of its values to be 0. However, it does so by using pointers instead of indexing over the course of the loop.

```
void clear_array_pointer(int arr[], int size);
```

(clear_array_pointer.h)

```
#include "clear_array_pointer.h"

void clear_array_pointer(int arr[], int size)
{
    for (int *p = &arr[0]; p < &arr[size]; p++)
        *p = 0;
}
```

(clear_array_pointer.cpp)

```
#include "clear_array_pointer.h"
#include <windows.h>
#include <iostream>
using namespace std;

int main()
{
    _int64 ctr1 = 0, ctr2 = 0, freq = 0;
    const int num = 100;
    int i, a[num];

    for (i = 0; i < num; i++)
    {
        a[i] = i + 1;
    }

    if (QueryPerformanceCounter((LARGE_INTEGER *)&ctr1) != 0)
    {
        clear_array_pointer(a, num);
        QueryPerformanceCounter((LARGE_INTEGER *)&ctr2);

        cout << "Start Value:" << ctr1 << endl;
        cout << "End Value:" << ctr2 << endl;

        QueryPerformanceFrequency((LARGE_INTEGER *)&freq);

        cout << "QueryPerformanceCounter minimum resolution: 1/" << freq << "Seconds," << endl;
        cout << "Clear Array of 100 needs:" << ((ctr2 - ctr1)* 1.0 / freq) * 1000000 << "Microseconds." << endl;
    }

    else
    {
        DWORD dwError = GetLastError();
        cout << "Error value" << dwError << endl;
    }
    system("PAUSE");
    return 0;
}
```

(ClearArrayPointer, main.cpp)

Array Size: 100

```
Start Value:3948904264405
End Value:3948904264406
QueryPerformanceCounter minimum resolution: 1/1753831Seconds,
Clear Array of 100 needs:0.57018Microseconds.
Press any key to continue . . .
```

Array Size: 1000

```
Start Value:3948987911614
End Value:3948987911619
QueryPerformanceCounter minimum resolution: 1/1753831Seconds,
Clear Array of 1000 needs:2.8509Microseconds.
Press any key to continue . . .
```

Array Size: 10000

```
Start Value:3949036472072
End Value:3949036472123
QueryPerformanceCounter minimum resolution: 1/1753831Seconds,
Clear Array of 10000 needs:29.0792Microseconds.
Press any key to continue . . .
```

Array Size: 100000

```
Start Value:3949096934273
End Value:3949096934731
QueryPerformanceCounter minimum resolution: 1/1753831Seconds,
Clear Array of 100000 needs:261.143Microseconds.
Press any key to continue . . .
```

Compiler Generated Assembly Code

Clear Array Index - Assembly

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01

    TITLE   C:\Users\brandon\Documents\Systems Organization\Classwork\4-13_Array_v_Pointer\ClearArrayIndex\ClearArrayIndex\clear_array_index.cpp
    .686P
    .XMM
    include listing.inc
    .model  flat

INCLUDELIB MSVCRTD
INCLUDELIB OLDNAMES

PUBLIC  ?clear_array_index@@YAXQAHH@Z           ; clear_array_index
EXTRN   __RTC_Shutdown:PROC
EXTRN   __RTC_InitBase:PROC
;   COMDAT rtc$TMZ
; File c:\users\brandon\documents\systems organization\classwork\4-13_array_v_pointer\cleararrayindex\cleararrayindex\clear_array_index.cpp
rtc$TMZ SEGMENT
;__RTC_Shutdown.rtc$TMZ DD FLAT:__RTC_Shutdown
rtc$TMZ ENDS
;   COMDAT rtc$IMZ
rtc$IMZ SEGMENT
;__RTC_InitBase.rtc$IMZ DD FLAT:__RTC_InitBase
; Function compile flags: /Odtp /RTCsu /ZI
rtc$IMZ ENDS
;   COMDAT ?clear_array_index@@YAXQAHH@Z
```

```
_TEXT   SEGMENT
_k$2537 = -8                        ; size = 4
_ary$ = 8                       ; size = 4
_size$ = 12                     ; size = 4
?clear_array_index@@YAXQAHH@Z PROC          ; clear_array_index, COMDAT

; 3    : {

    push    ebp
    mov ebp, esp
    sub esp, 204                ; 000000ccH
    push    ebx
    push    esi
    push    edi
    lea edi, DWORD PTR [ebp-204]
    mov ecx, 51                 ; 00000033H
    mov eax, -858993460             ; ccccccccH
    rep stosd

; 4    :    for (int k = 0; k < size; k++)

    mov DWORD PTR _k$2537[ebp], 0
    jmp SHORT $LN3@clear_arra
$LN2@clear_arra:
    mov eax, DWORD PTR _k$2537[ebp]

    add eax, 1
    mov DWORD PTR _k$2537[ebp], eax
$LN3@clear_arra:
    mov eax, DWORD PTR _k$2537[ebp]
    cmp eax, DWORD PTR _size$[ebp]
    jge SHORT $LN4@clear_arra

; 5    :        ary[k] = 0;

    mov eax, DWORD PTR _k$2537[ebp]
    mov ecx, DWORD PTR _ary$[ebp]
    mov DWORD PTR [ecx+eax*4], 0
    jmp SHORT $LN2@clear_arra
$LN4@clear_arra:

; 6    : }

    pop edi
    pop esi
    pop ebx
    mov esp, ebp
    pop ebp
    ret 0
?clear_array_index@@YAXQAHH@Z ENDP          ; clear_array_index
_TEXT   ENDS


END
```

(clear_array_index.asm)

Clear Array Pointer - Assembly

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01

    TITLE   C:\Users\brandon\Documents\Systems Organization\Classwork\4-13_Array_v_Pointer\ClearArrayPointer\ClearArrayPointer\clear_array_pointer.cpp
    .686P
    .XMM
    include listing.inc
    .model  flat

INCLUDELIB MSVCRTD
INCLUDELIB OLDNAMES

PUBLIC  ?clear_array_pointer@@YAXQAHH@Z         ; clear_array_pointer
EXTRN   __RTC_Shutdown:PROC
EXTRN   __RTC_InitBase:PROC
;   COMDAT rtc$TMZ
; File c:\users\brandon\documents\systems organization\classwork\4-13_array_v_pointer\cleararraypointer\cleararraypointer\clear_array_pointer.cpp
rtc$TMZ SEGMENT
;__RTC_Shutdown.rtc$TMZ DD FLAT:__RTC_Shutdown
rtc$TMZ ENDS
;   COMDAT rtc$IMZ
rtc$IMZ SEGMENT
;__RTC_InitBase.rtc$IMZ DD FLAT:__RTC_InitBase
; Function compile flags: /Odtp /RTCsu /ZI
rtc$IMZ ENDS
;   COMDAT ?clear_array_pointer@@YAXQAHH@Z
```

```
_TEXT   SEGMENT
_p$2537 = -8                        ; size = 4
_arr$ = 8                       ; size = 4
_size$ = 12                     ; size = 4
?clear_array_pointer@@YAXQAHH@Z PROC            ; clear_array_pointer, COMDAT

; 4    : {

    push    ebp
    mov ebp, esp
    sub esp, 204                ; 000000ccH
    push    ebx
    push    esi
    push    edi
    lea edi, DWORD PTR [ebp-204]
    mov ecx, 51                 ; 00000033H
    mov eax, -858993460             ; ccccccccH
    rep stosd

; 5    :    for (int *p = &arr[0]; p < &arr[size]; p++)

    mov eax, DWORD PTR _arr$[ebp]
    mov DWORD PTR _p$2537[ebp], eax
    jmp SHORT $LN3@clear_arra
$LN2@clear_arra:

    mov eax, DWORD PTR _p$2537[ebp]
    add eax, 4
    mov DWORD PTR _p$2537[ebp], eax
$LN3@clear_arra:
    mov eax, DWORD PTR _size$[ebp]
    mov ecx, DWORD PTR _arr$[ebp]
    lea edx, DWORD PTR [ecx+eax*4]
    cmp DWORD PTR _p$2537[ebp], edx
    jae SHORT $LN4@clear_arra

; 6    :        *p = 0;

    mov eax, DWORD PTR _p$2537[ebp]
    mov DWORD PTR [eax], 0
    jmp SHORT $LN2@clear_arra
$LN4@clear_arra:

; 7    : }

    pop edi
    pop esi
    pop ebx
    mov esp, ebp
    pop ebp
    ret 0

?clear_array_pointer@@YAXQAHH@Z ENDP            ; clear_array_pointer
_TEXT   ENDS
END
```

(clear_array_pointer.asm)

Clear Array Using Index Approach - Optimization

Before Optimization:

```
; 4    :    for (int k = 0; k < size; k++)

    mov DWORD PTR _k$2537[ebp], 0
    jmp SHORT $LN3@clear_arra
$LN2@clear_arra:
    mov eax, DWORD PTR _k$2537[ebp]
    add eax, 1
    mov DWORD PTR _k$2537[ebp], eax
$LN3@clear_arra:
    mov eax, DWORD PTR _k$2537[ebp]
    cmp eax, DWORD PTR _size$[ebp]
    jge SHORT $LN4@clear_arra

; 5    :        ary[k] = 0;

    mov eax, DWORD PTR _k$2537[ebp]
    mov ecx, DWORD PTR _ary$[ebp]
    mov DWORD PTR [ecx+eax*4], 0
    jmp SHORT $LN2@clear_arra
$LN4@clear_arra:
```

After Optimization:

```
; 4    :    for (int k = 0; k < size; k++)

    mov DWORD PTR _k$2537[ebp], 0
    mov ecx, DWORD PTR _ary$[ebp]   ; modified
    jmp SHORT $LN3@clear_arra
$LN2@clear_arra:
    mov eax, DWORD PTR _k$2537[ebp]
    add eax, 1
    mov DWORD PTR _k$2537[ebp], eax
$LN3@clear_arra:
    mov eax, DWORD PTR _k$2537[ebp]
    cmp eax, DWORD PTR _size$[ebp]
    jge SHORT $LN4@clear_arra

; 5    :        ary[k] = 0;

    mov eax, DWORD PTR _k$2537[ebp]

    mov DWORD PTR [ecx+eax*4], 0
    jmp SHORT $LN2@clear_arra
$LN4@clear_arra:
```

Originally, the register ECX is assigned the same address stored on stack at every instance of the loop. In order to optimize this, I have modified the code by initializing the register ECX before the loop begins. That way, this assignment only needs to be executed once.

Array Size: 100

```
Start Value:4093193646553
End Value:4093193646554
QueryPerformanceCounter minimum resolution: 1/1753831Seconds,
Clear Array of 100 needs:0.57018Microseconds.
Press any key to continue . . .
```

Array Size: 1000

```
Start Value:4093247968987
End Value:4093247968992
QueryPerformanceCounter minimum resolution: 1/1753831Seconds,
Clear Array of 1000 needs:2.8509Microseconds.
Press any key to continue . . .
```

Array Size: 10000

```
Start Value:4093550841141
End Value:4093550841190
QueryPerformanceCounter minimum resolution: 1/1753831Seconds,
Clear Array of 10000 needs:27.9388Microseconds.
Press any key to continue . . .
```

Array Size: 100000

```
Start Value:4093604713223
End Value:4093604713686
QueryPerformanceCounter minimum resolution: 1/1753831Seconds,
Clear Array of 100000 needs:263.994Microseconds.
Press any key to continue . . .
```

Clear Array Using Pointer Approach - Optimization

Before Optimization:

```
; 5    :    for (int *p = &arr[0]; p < &arr[size]; p++)

    mov eax, DWORD PTR _arr$[ebp]
    mov DWORD PTR _p$2537[ebp], eax
    jmp SHORT $LN3@clear_arra
$LN2@clear_arra:
    mov eax, DWORD PTR _p$2537[ebp]
    add eax, 4
    mov DWORD PTR _p$2537[ebp], eax
$LN3@clear_arra:
    mov eax, DWORD PTR _size$[ebp]
    mov ecx, DWORD PTR _arr$[ebp]
    lea edx, DWORD PTR [ecx+eax*4]
    cmp DWORD PTR _p$2537[ebp], edx
    jae SHORT $LN4@clear_arra

; 6    :        *p = 0;

    mov eax, DWORD PTR _p$2537[ebp]
    mov DWORD PTR [eax], 0
    jmp SHORT $LN2@clear_arra
$LN4@clear_arra:
```

After Optimization:

```
; 5    :    for (int *p = &arr[0]; p < &arr[size]; p++)

    mov eax, DWORD PTR _arr$[ebp]
    mov ebx, DWORD PTR _size$[ebp]      ; MODIFIED
    lea edx, DWORD PTR [eax+ebx*4]      ; MODIFIED
    jmp SHORT $LN3@clear_arra
$LN2@clear_arra:
    add eax, 4
$LN3@clear_arra:
    cmp eax, edx                        ; MODIFIED DWORD PTR _p$2537[ebp] to eax
    jae SHORT $LN4@clear_arra

; 6    :        *p = 0;

    mov DWORD PTR [eax], 0
    jmp SHORT $LN2@clear_arra
$LN4@clear_arra:
```

In order to optimize this code, a lot of statements are removed. Many registers are assigned values unnecessarily because these values do not change. There are also cases where the same value is written twice, once on stack and once again in registers. I have modified the code to accommodate for these inefficiencies.

Array Size: 100

```
Start Value:195303175208
End Value:195303175208
QueryPerformanceCounter minimum resolution: 1/1753829Seconds.
Clear Array of 100 needs:0Microseconds.
Press any key to continue . . .
```

Array Size: 1000

```
Start Value:195621576445
End Value:195621576447
QueryPerformanceCounter minimum resolution: 1/1753829Seconds.
Clear Array of 1000 needs:1.14036Microseconds.
Press any key to continue . . .
```

Array Size: 10000

```
Start Value:196101507054
End Value:196101507068
QueryPerformanceCounter minimum resolution: 1/1753829Seconds.
Clear Array of 10000 needs:7.98253Microseconds.
Press any key to continue . . .
```

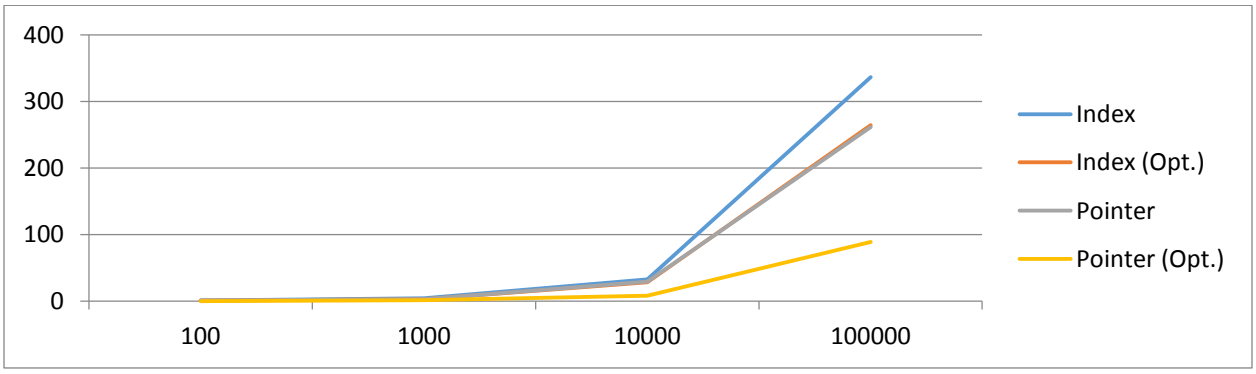
Array Size: 100000

```
Start Value:196022146312
End Value:196022146468
QueryPerformanceCounter minimum resolution: 1/1753829Seconds.
Clear Array of 100000 needs:88.9482Microseconds.
Press any key to continue . . .
```

Results

	Size N			
	100	**1000**	**10000**	**100000**
Index (Before Optimization)	0.57018	3.99126	32.5003	336.406
Index (After Optimization)	0.57018	2.85090	27.9388	263.994
Pointer (Before Optimization)	0.57018	2.85090	29.0792	261.143
Pointer (After Optimization)	0.00000	1.14036	7.98253	88.9482

(Table #1)



(Graph #1)