

Parallel Loop Detector in C Programs

Wei Wang ww6r@virginia.edu
Tanima Dey td8h@virginia.edu
University of Virginia, Charlottesville, VA 22904

Abstract

Parallelizing compilers is one of the hot topics in both Parallel Computing and Compiler world now-a-days. Given a serial program, it is still very challenging to design a compiler that produces another version of the same program that is appropriate for parallel architecture. In this paper, we discuss one of the techniques to detect portion of a source code written in C programming language that can be transformed into parallel code. Our system dynamically detects all the memory accesses done within a loop and based on this determines the independence of multiple loops. That is, our system is able to detect loops that can be parallelized based on data dependencies between them. We use a very powerful software dynamic translator, called Strata, to analyze the binary files of the corresponding serial source code.

1. Introduction

Parallel computing is one of the most prominent fields in Computer Science. Parallel computing and multi-processors are used to solve various kinds of computation intensive problems and data parallel applications. It is also used in modeling different natural phenomenon, such as, for implementing the ocean models, weather models etc. It has other applications regarding high throughput computing, computer graphics etc. Again, multi-core machines are the state of the art which is also used for high performance computing.

In spite of numerous applications of parallel computing, it is still very difficult to write parallel programs for various applications. One of the main challenges is that there is no standard parallel programming language or compiler that is widely accepted among the programmers. The programmers use C or FORTRAN programming language which is extended to support different parallel constructs for multiple processes or threads where the source code is compiled with the revised version of the compiler for supporting such extensions. Such as, *mpicc* is used for Message Passing Interface (MPI) used in C.

Another direction to parallelism is to design a compiler which automatically transforms a sequential code into multithreaded or vectorized code to utilize the underlying multi-processor architecture. This is known as Automatic Parallelization [1]. There are several challenges to overcome in order to write such compilers, such as, thorough analysis of the source program for data, loop or control dependencies. The dependency analysis is also hard for the source code that uses indirect addressing, handling pointers, calling recursive function etc. The first thing that these compilers should check is whether it is safe to parallelize a particular loop, i.e., to find out if

the loop is not dependent on the effect produced or variable updated by another loop. If it is safe then those two loops can be parallelized. The second issue to consider is that the analysis of the dependencies among the loops should not be of high overhead and the actual speedup obtained by this parallelism is worthwhile compared to the sequential execution of the code.

Motivated by the challenges faced for developing a compiler for automatic parallelization, we design and develop a parallel loop detector which is capable of detecting the loops those can be written using parallel constructs and executed on parallel architectures. This parallel loop detector analyzes the sequential source code and the binary file produced after the sequential code is compiled by a serial compiler, such as, GCC and produces the output that conveys the programmer which lines containing the loops can be made parallel. The converter statically analyzes the source code to know the line numbers containing *for*, *while*, *do-while* loops and dynamically analyses the memory accesses inside loops to determine the data dependency.

The paper is organized as follows: In Section 2, we describe the different part of our system, in Section 3 we discuss the experiments and results for small and simple example program, we discuss the limitations and future works in Section 4 and conclude in Section 5.

2. Program Implementation and Usage

In this section, we describe the various parts of our system that detect the parallelizable loops based on memory accesses. These are described as follows:

2.1 Usage

To use the instrumentation program, we need to change the program's source a little bit. We have to include the header file "strata.h", put "strata_start()" and "strata_init()" at the beginning of "main" function, and put "strata_stop" at the end of "main" function. There is a way to "stratify" a program without changing the source code by modifying original binary file. However, this method changes the PC value of assembly instructions in original binary file, making it hard to map the Program Counter (PC) back to source code line number.

In the second step, we set environment variable "STRATA_LOG" to "paraloop", compile and run the program normally. Then the system provides the output of all possible loops in PC addresses. Then we save these PC addresses into a file, name it "looplist.txt". We verify the PC addresses using "PC to source" converter (described in the later subsection) and make sure only loops generated by the program is left. We rerun the program. This time the detector reads in "looplist.txt", and instruments loops in the source file. Memory addresses accessed by the program are output to the screen. These memory addresses are sorted by their loop number and different iteration number within the loop. This information is then used in dependency analyzer to find whether several loops can be executed in parallel, or whether a loop can be broke down and executed in parallel.

2.2 Strata

Strata [4] is software dynamic translator (SDT) and a virtual machine (VM) developed here at UVa. The aim of Strata is to develop a cross-platform SDT to serve as the infrastructure for various kinds of research. Strata can be used for several purposes, such as, dynamic code instrumentation, dynamic software updates, ensuring safe execution of untrusted binaries, simulation of fast caches etc. We mainly use Strata for code instrumentation in our project. When a program is executed under the control of Strata, its instructions are translated one by one into a separate memory called *fragment cache* for execution. During this translation phase, many operations including software instrumentation can be performed which can provide valuable insights about the execution of the program.

Figure 1 shows the operations performed by the Strata VM. Strata mediate application execution by examining and translating instructions before they execute on the host CPU. The Strata VM is first entered by capturing

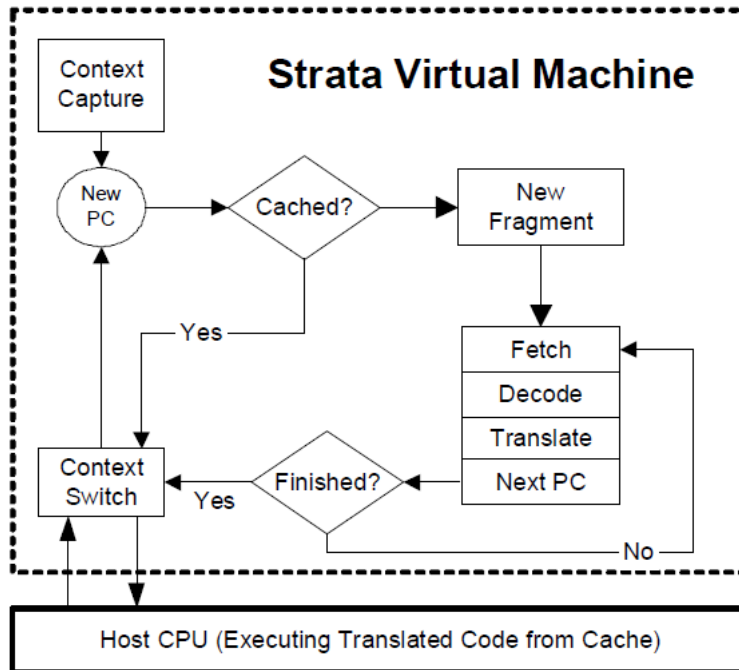


Figure 1. The main operations by Strata [4].

and saving the application context (e.g., PC, condition codes, registers.) The Strata VM begins processing the next application instruction. If a translation for this instruction has been cached, a context switch restores the application context and begins executing cached translated instructions on the host CPU. If there is no cached translation for the next application instruction, the Strata VM allocates storage in the cache for a new fragment of translated instructions. A fragment is a sequence of code in which branches may appear only at the end. The Strata VM then populates the fragment by fetching, decoding, and translating application instructions one-by-one until an end-of-fragment condition is met. A context switch restores the application context and the newly translated fragment is executed when the fragment ends, i.e., when the end-of-fragment condition is met.

2.3 Loop Detection

To find out parallel executable loops, the first thing we have to do is to automatically detect all loops within a program. Loop detection could be very tricky if compiler optimizations are involved. For simplicity, we only consider unoptimized code generated by ordinary compiler, like GCC. Usually, compiler generates a loop using backward jump instructions when optimizing flags are turned off. The following piece of codes illustrates how loops are usually generated in assembly code by GCC.

```

L1:
    insn1
    insn2
    insn3
    ...
    jmp L1

```

Since most loops use backward jumps, we use just this as an indicator of loop. In Strata, every instruction has to be fetched from ordinary text section of the memory and re-emitted into fragment cache. For each instruction fetch, we decide whether it is a short or near backward jump by its operation code (no matter it is a conditional jump or not). We don't consider long jumps since jumping between segments cannot be loops. The beginning

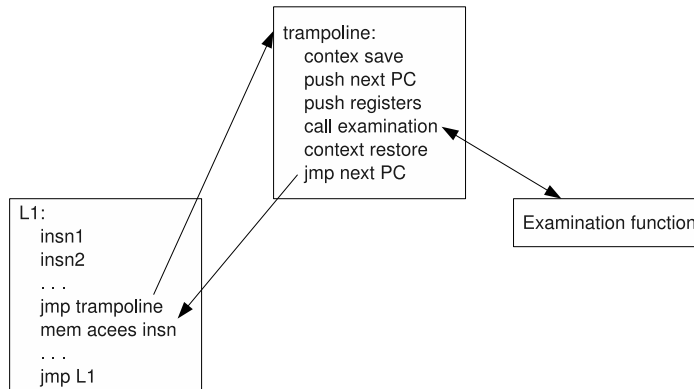


Figure 2. Trampolines Inserted into Application Code.

of a loop can also be calculated based on the operand of the jump instruction. Once we find a backward jump, we assume we find a potential loop; PC address of both backward jump and its target are considered as outputs. We call “PC to source” converter to examine if this is a real loop. If it is then we instrument it later. One reason of employing this verification step is because there are many loops within library codes, such as libc. Strata also translates these loops as well, so we need a way to exclude these loops from instrumentation. The other reason is that backward jumps are also used to implement “if” statements. We don’t want to instrument them either.

2.4 Loop Instrumentation

Once we find the loops, we start to instrument them. The basic idea is very simple. For each memory access instruction, we insert a trampoline before its execution. This trampoline is simply a context save and a jump to an instruction examination function. We also push the address of next memory access instruction and current register values onto the stack in the trampoline. We use this method to pass enough information to the instruction examination function as input parameters. Figure 2 illustrates how trampolines work.

In the examination function, we read in next memory access instruction and decode the target memory address using values from pertinent registers. Once we calculate an address successfully, we output the results, indicating which memory address is going to be accessed, how long memory is going to be accessed (8, 16, 32 or 64bits), and whether it is a read or a write. We insert one trampoline for each memory access operation because most instructions use indirect memory address. The real physical address has to be calculated based on correct register values. And we get the correct register values only right before that instruction is about to execute. This could reduce performance a lot, but it is the only way to get the true memory address. After the examination function finishes, context is restored, control is returned back to the application by jumping to next memory access instruction.

2.5 x86 Memory Access Instructions

A typical x86 instruction consists of 6 parts: instruction prefixes, opcode, ModR/M, SIB, displacement, immediate. Figure 3 illustrates x86 instruction format.

Opcode determines the type of instruction, and ModR/M determines whether memory access is involved. For memory access, memory addresses are usually calculated based on ModR/M, SIB and displacement. ModR/M indicates whether an instruction is using direct or indirect addressing mode. If indirect mode is used, it also tells us which register to use to compute the effective address as well as if SIB byte is used. SIB is usually used in addressing arrays. It contains scale index, the register that has the length of each array element; index, the index of the array element being accessed; and base, the base address of the arrays being accessed. Finally, displacement contains an offset to the base address which is usually the frame pointer. For example, ModR/M



Figure 3. x86 Instruction Format from Intel Manual [3].

with value $0x83$ means the effective address is $ebx + 32bits\ displacement$. SIB with value $0x95$ means the effective address is $edx \times 4 + esp + 8bits/32bits\ displacement$ (which displacement to use depends on ModR/M).

For ModR/M, there are 255 possible values, 192 of which have something to do with memory access. In these 192 possible values, 3 values need SIB. And for SIB, there are also 255 possible values. As a result there are 954 different memory addressing methods we have to deal with in our system. It is a big value and requires extreme care while handling those.

2.6 PC to Source Conversion

Mapping the PC value back to the source code information is very important in our system as it can notify the programmer the exact line numbers in which the loops can be made parallel. There can be a number of ways to do this reverse mapping. The main challenge of this reverse mapping is that when a source code, written in C programming language, is compiled, the source code information is removed by the compiler when it makes the executable binary file. So after the source code is compiled and linked, there is no way to find the source line number from the binary if no compiler flag or any other program is used.

The general way that debugging software works is that while compilation they use different flags, such as, for GDB we use '-g' flag during compiling the source code using GCC. This flag incorporates the debugging information in the form of stubs into the executable file which includes the symbol table, string table etc. data structures containing major source code information, such as, variable names, addresses etc. This enables the debugger to inform the programmers the exact location of a variable in the source code, setting break points etc.

Instead of using the debugger style reverse mapping, we have adopted a simpler approach to find the reverse mapping because we don't need large information of the source code but only the line numbers of different loop constructs of C that are used in the source code. The converter uses both the source and executable file. The source file must be compiled by the GCC compiler and executable file must have the same name as the source code file. For example, if we are analyzing the source file named "test.c" then the executable file's name should be "test" and both must be present in the same directory. In order to do the reverse mapping, the converter follows two steps. In the first step, the source is statically analyzed to find the different loop constructs. This is similar to the lexical analysis phase of a standard compiler which scans through the source code by each character. During this process, we save the line numbers into an array, named *lineNumber*, in which we find the keywords of the loop constructs, such as, *for*, *while* loop etc.

In the second step, we find the PC values for the loop constructs. In order to accomplish that we use *objdump* [2] program provided in standard Linux distribution. This program is mainly used to find out different information of the binary file for Linux. This program supports several options among which '-d' option provides the assembly code of the object or executable file of the program being analyzed. Basically it disassembles the executable file into the assembly language which contains the information that we need for the reverse mapping of the PC to source code line number. The output generated by *objdump* has a common structure in every line. Each line can be of two types, either a start of the function definition or the machine code followed by the associated assembly instruction. But each line starts with the corresponding PC value. So after this disassembled executable file is generated, we simply look for the main function (for the time being we are assuming, the source code contains only the main function). After we detect the starting of the main function, we start to save the PC values and opcode in each line in two separate arrays until we reach the end of this function. Then we analyze the opcode array to detect the assembly language keywords for loop constructs, such as, *jle*, *jl*, *jne*, *je*, *je*, *jge* etc. and save those PC values into a separate array, named *PCloop*, where we get such opcodes. Thus this array has an one-to-one mapping with the *lineNumber* array which contains the line number of the source file where we find the original C loop constructs (such as, for, while).

After these two arrays are generated, we simply read the "looplist.txt" file generated in loop detection phase where each line contains the starting and ending PC values of each loop. From these two PC values, the converter sends the loop detector -1 if the PC values are outside the range of main function for its verification part and the "looplist.txt" gets updated. It also maps the proper PC values into source code line numbers to convey information to the programmer.

3. Results and Evaluation

In this section, we describe how the loop detector is used to detect dependencies within and between loops. First, let's see the C Program 1 we are going to analyze as an example. In this small program, we have two loops, both write to array *a*, thus having write-after-write (WAW) dependency.

Program 1 A Small Sample Program to be Analyze

```
int main(){
    int i;
    int a[10];
    //loop 0
    for( i = 0; i < 10; i++)
        a[i] = i;
    //loop 1
    int j = 0;
    while(j<10){
        a[j] = 6;
        j++;
    }

    return 0;
}
```

After we compile the program with Strata, we run it for the first time to detect all loop address as demonstrated in Output 1. Each line in the output represents a loop. The first address in each line is the program counter of the first instruction of a loop; and the second address is the program counter of the last instruction of

a loop. As it is observed, there are more than 2 loops. Many loops are actually from system library code. After using "PC to Source" converter we learn that only the first two loops are actually from the program we wrote. As a result, in the "looplist.txt" file we only put only the first two loops.

Output 1 Loop Addresses Detected

```
8048dba,8048dcc
8048dd7,8048dea
f7faa600,f7faa623
f7faa2e4,f7faa310
f7fb7b18,f7fb7b22
f7fb7b18,f7fb7b22
f7faa568,f7faa581
f7faa0e2,f7faa111
...
```

Then we run the program again. This time the detector actually instruments the program and finds out all memory accesses in a loop. The output from our sample program is illustrated in Output 2. "Loop 0" indicates which loop it is executing now. Loops are number from 0, corresponding to the loop addresses in "looplist.txt" with the loop in the first line numbered 0. Each line "Loop 0: Begin loop iteration" indicates a new iteration of the loop body. Line "Loop 0: Write to memory ffda7340 32 bits, 0xffda7340W32" shows which memory is accessed. As literally written in the line, in the loop 0, 32 bit memory from address 0xffda7340 is written. The part after comma, "0xffda7340W32", basically says the same thing: address 0xffda7340, written (or R if read), 32 bits. This part is used for dependencies analysis later. Also, from this output we know that each loop has been executed for 10 times.

We take the memory access output and run it with the dependency analyzer. It gives the Output 3. As written in Output 3, Loop 0 and Loop 1 has WAW dependency at memory address from 0xffda7340 and 0xffda7364. So they cannot be executed in parallel. If there are no outputs from dependency analyzer, then the loops do not have dependencies and can be run in parallel. Besides WAW dependency, we can also detect read-after-write (RAW) and write-after-read (WAR) dependency. Read-after-read is considered as parallelizable. Actually using the memory access output, we can also tell if different iterations of the same loop have dependencies among them.

Finally, the "PC to Source converter" also produces the output shown in Output 4 so that the programmer knows the exact line number at which the loops are actually written after analyzing the "looplist.txt" file earlier.

We include two more examples each containing two independent loops. The examples are shown in Program 2 and Program 3 with the corresponding output from the loop instrumentation in Output 5 and Output 6 respectively. In both of the programs, loops are independent so there is no output from the loop detector.

4. Limitations and Future Work

Currently we implemented all addressing method and tested most 32 bits instructions. We also tested some 64 bits instructions. Although more tests have to be done, most of the untested instructions should be safe to use, since they shared the same instruction format. We cannot handle SIMD instructions now. These instructions include MMX instruction set, SSE1 3, SSSE3, and SSE4 instruction set. Implementing logic to handle these instructions should not be too hard. However testing them would be a little bit painful, since it is hard to write or find programs using it. We plan to put SIMD instruction handling into future work.

Output 2 Memory Accessed by both loops

Loop 0: Begin loop iteration
Loop 0: Write to memory ffda7340 32 bits, 0xffda7340W32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffda7344 32 bits, 0xffda7344W32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffda7348 32 bits, 0xffda7348W32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffda734c 32 bits, 0xffda734cW32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffda7350 32 bits, 0xffda7350W32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffda7354 32 bits, 0xffda7354W32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffda7358 32 bits, 0xffda7358W32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffda735c 32 bits, 0xffda735cW32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffda7360 32 bits, 0xffda7360W32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffda7364 32 bits, 0xffda7364W32
Loop 1: Begin loop iteration
Loop 1: Write to memory ffda7340 32 bits, 0xffda7340W32
Loop 1: Begin loop iteration
Loop 1: Write to memory ffda7344 32 bits, 0xffda7344W32
Loop 1: Begin loop iteration
Loop 1: Write to memory ffda7348 32 bits, 0xffda7348W32
Loop 1: Begin loop iteration
Loop 1: Write to memory ffda734c 32 bits, 0xffda734cW32
Loop 1: Begin loop iteration
Loop 1: Write to memory ffda7350 32 bits, 0xffda7350W32
Loop 1: Begin loop iteration
Loop 1: Write to memory ffda7354 32 bits, 0xffda7354W32
Loop 1: Begin loop iteration
Loop 1: Write to memory ffda7358 32 bits, 0xffda7358W32
Loop 1: Begin loop iteration
Loop 1: Write to memory ffda735c 32 bits, 0xffda735cW32
Loop 1: Begin loop iteration
Loop 1: Write to memory ffda7360 32 bits, 0xffda7360W32
Loop 1: Begin loop iteration
Loop 1: Write to memory ffda7364 32 bits, 0xffda7364W32

Output 3 Dependencies between loops

Loop 0 and Loop 1 has WRW dependency at memory 0xffda7340.
Loop 0 and Loop 1 has WRW dependency at memory 0xffda7344.
Loop 0 and Loop 1 has WRW dependency at memory 0xffda7348.
Loop 0 and Loop 1 has WRW dependency at memory 0xffda734c.
Loop 0 and Loop 1 has WRW dependency at memory 0xffda7350.
Loop 0 and Loop 1 has WRW dependency at memory 0xffda7354.
Loop 0 and Loop 1 has WRW dependency at memory 0xffda7358.
Loop 0 and Loop 1 has WRW dependency at memory 0xffda735c.
Loop 0 and Loop 1 has WRW dependency at memory 0xffda7360.
Loop 0 and Loop 1 has WRW dependency at memory 0xffda7364.

Output 4 Dependencies between loops

Loop 0 : Line 7
Loop 1 : Line 11

Program 2 Example Program 2

```
int main()
{
    int a[10];
    int b[5];

    int i;
    for(i=0;i<10;i++)
    {
        a[i]=10;
    }

    for(i=0;i<5;i++)
    {
        b[i]=i;
    }

    return 0;
}
```

Output 5 Memory Accessed by both loops for Example 2

```
Loop 0: Begin loop iteration
Loop 0: Write to memory ffb498c4 32 bits, 0xffb498c4W32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffb498c8 32 bits, 0xffb498c8W32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffb498cc 32 bits, 0xffb498ccW32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffb498d0 32 bits, 0xffb498d0W32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffb498d4 32 bits, 0xffb498d4W32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffb498d8 32 bits, 0xffb498d8W32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffb498dc 32 bits, 0xffb498dcW32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffb498e0 32 bits, 0xffb498e0W32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffb498e4 32 bits, 0xffb498e4W32
Loop 0: Begin loop iteration
Loop 0: Write to memory ffb498e8 32 bits, 0xffb498e8W32
Loop 1: Begin loop iteration
Loop 1: Write to memory ffb498ec 32 bits, 0xffb498ecW32
Loop 1: Begin loop iteration
Loop 1: Write to memory ffb498f0 32 bits, 0xffb498f0W32
Loop 1: Begin loop iteration
Loop 1: Write to memory ffb498f4 32 bits, 0xffb498f4W32
Loop 1: Begin loop iteration
Loop 1: Write to memory ffb498f8 32 bits, 0xffb498f8W32
Loop 1: Begin loop iteration
Loop 1: Write to memory ffb498fc 32 bits, 0xffb498fcW32
```

Nested loops can be processed by our program, however, we cannot separate two executions of the same loop. This is because Strata break fragment at each jump. So we are not sure if it is safe to emit another trampoline at the end of a fragment cache, right after the backward jump (the fail path). We believe we can enable this feature in the future.

We can detect loops and monitor their memory accesses in any functions in a program. However, the loops that are not inside main() function, we are not able to map their PC to source line number for the time being. We can add this feature by adding additional data structures to monitor all functions dumped by "objdump". Due to the time constraint, we decided to complete the basic requirements to detect loops having memory accesses that can be made parallel.

There can be another way to utilize the loop independence of the program. After we detect each of the independent loop, we can insert OpenMP pragma's into the source to produce the modified source code that can be compiled and executed directly on a parallel machine.

Program 3 Example Program 3

```
int main()
{
    int a[10];
    int i;

    for(i=0;i<5;i++)
    {
        a[i]=10;
    }

    for(i=5;i<10;i++)
    {
        a[i]=20;
    }

    return 0;
}
```

Output 6 Memory Accessed by both loops for Example 3

```
Loop 0: Begin loop iteration
Loop 0: Write to memory ff989718 32 bits, 0xff989718W32
Loop 0: Begin loop iteration
Loop 0: Write to memory ff98971c 32 bits, 0xff98971cW32
Loop 0: Begin loop iteration
Loop 0: Write to memory ff989720 32 bits, 0xff989720W32
Loop 0: Begin loop iteration
Loop 0: Write to memory ff989724 32 bits, 0xff989724W32
Loop 0: Begin loop iteration
Loop 0: Write to memory ff989728 32 bits, 0xff989728W32
Loop 1: Begin loop iteration
Loop 1: Write to memory ff98972c 32 bits, 0xff98972cW32
Loop 1: Begin loop iteration
Loop 1: Write to memory ff989730 32 bits, 0xff989730W32
Loop 1: Begin loop iteration
Loop 1: Write to memory ff989734 32 bits, 0xff989734W32
Loop 1: Begin loop iteration
Loop 1: Write to memory ff989738 32 bits, 0xff989738W32
Loop 1: Begin loop iteration
Loop 1: Write to memory ff98973c 32 bits, 0xff98973cW32
```

5. Conclusion

Automatic parallelization is still very challenging and one of the hardest problems in compilers. There can be a number of ways (both static and dynamic) to analyze the data dependency among loops and we present one of the approaches. Usually, to address one of the difficulties of detecting data independence, very high level abstraction, such as, dependency graphs are used. Such graphs contain numerous paths of execution and difficult analysis techniques. The nice thing about our approach is that we are translating the binary of the program dynamically, so we have more information to consider, especially we can examine the individual memory locations as accessed by the program inside. Again, since we are instrumenting the program binary dynamically, we can also add additional checking and functionalities to it without modifying Strata much. We are hopeful that SDT techniques can be fruitful for designing compilers for automatic parallelization.

References

- [1] Automatic parallelization, wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Automatic_parallelization.
- [2] Linux manual page for objdump. <http://linux.die.net/man/1/objdump>.
- [3] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 2A. Intel Corporation, 2009.
- [4] Kevin Scott and Jack Davidson. *Strata: A Software Dynamic Translation Infrastructure*. University of Virginia, Charlottesville, VA, USA, 2001.