An Ameliorated Methodology for the Control and Data flow of a Legacy Program for Multicore Machine.

Shoba M[1], Vinay T R[2]
Department of Computer Science & Engineering
Sri Venkateshwara College of Engineering
Bangalore, India
shoba0708@hotmail.com, tr.vinay@gmail.com

Abstract

Hardware and software are not going hand in glove with each other; hardware is growing at a faster rate. Now we are having multiprocessor machines compared to having a single processor machines few years ago. The legacy programs which were developed on this single processor machine are not copying with the advancement of hardware technology, i.e., if we run this programs on multicore machines, it would make use of only primary core and not utilizing the other cores. This leads to under utilization of multi-core machines. The paper proposes a methodology for restructuring procedure oriented programs in the same relative abstraction level in the form of automatic and semi-automatic techniques. The program is divided into chunks of smaller programs called slices, and these are converted into threads which can be executed in parallel on different cores on a multicore machine. This leads to better utilization of available cores and efficiency of the program also increases.

Key words: Restructuring, Multicore machines, Program Slicing, Control Flow Graph, Data Flow Table, Dependencies.

1. INTRODUCTION

Multi-processor workstations have been available for many years. However, until recently systems with more than one execution core have been beyond the price range of the average home user. The Intel(R) Core(TM) Duo processor is helping to change this. Now parallel processing capability is within the reach of every user. The question that naturally comes up is "Now that I have two cores, what can I do with them"? The short answer to this question is that you either:

1. Run more programs simultaneously (multi-tasking)
2. Parallelize your applications (multi-threading or parallel programming).

In this paper we will take a look at these two methods to take advantage of the additional processing power provided by a dual-core processor. We will emphasize parallel programming models as this is perhaps most of interest to software developers interested in high-performance, scalable applications.

Multi-tasking parallelism is a model where two or more processes are created and run simultaneously by the OS. This follows: the creation of the processes is done either programmatically via fork ()/exec () function calls or often times by simply running multiple instances of the same program simultaneously. The processes created in either of these methods will be scheduled on the two cores by the OS.

This model of parallelism works well when the user is performing parametric studies, performing Monte Carlo simulations, statistical studies, or where a large number of job runs are used to characterize a solution space. Typically these types of problems use tens to thousands of input files with variations on the characteristic parameters. The same executable program is run as a separate job for each file. The goal with this model is throughput, or processing as many runs in a given wall clock time as possible.

This model of parallelism benefits from the multiple cores by treating each core as an execution unit for each of the separate tasks (jobs). However, having the extra processing core in the Intel(R) Core(TM) Duo processor benefits this model by providing the system with twice the throughput capability over a single-core system. This model should be straight forward and familiar.

The remainder of this paper focuses on multi-threaded programming techniques. These techniques enable a single program to create multiple threads of execution. Threads are scheduled equally on available executions cores providing increased application performance. The broad view of the methodology is depicted as a block diagram in (figure 1).

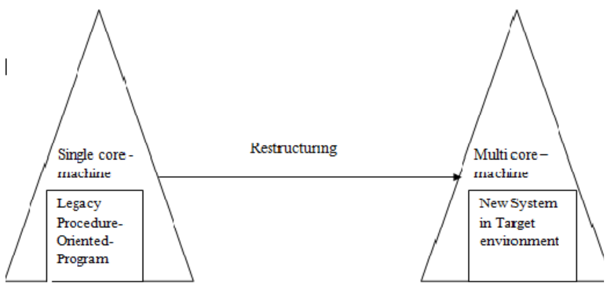
Single core - machine
Restructuring
Multi core – machine
Legacy Procedure-Oriented-Program
New System in Target environment

Figure 1: Specific Model for Re-engineering

2. MOTIVATION

A. *Parallelization via Auto-Parallelizing Compilers*

Auto-parallelizing compilers attempt to analyze programs and automatically generate parallel code. As far as ease of use for the programmer, there is nothing easier than taking an existing serial code and allowing an auto-parallelizing compiler to generate parallel code. There is an old saying that applies: If something seems too good to be true, it probably is. Creating an auto-parallelizing compiler that produces efficient code in all cases for a wide variety of applications has proven to be a very difficult task. Thus, auto-parallelization remains a hot research topic and the perfect auto-parallelizing compiler is yet to be produced. However, there are some cases where auto-parallelization is perfectly suited

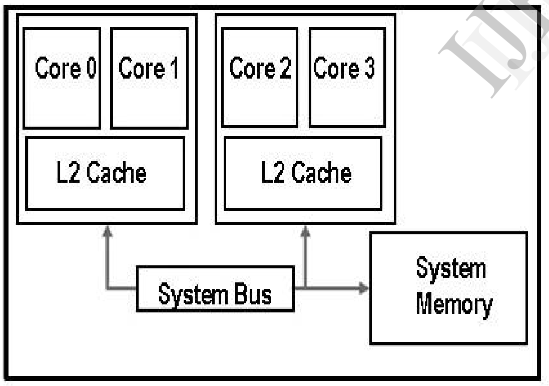

Figure 2: Multi-core processor showing 4 cores.

B. *Maintaining the Integrity of the Specifications*

Consider a lengthy industry program that may run up to thousands of lines. Dividing this program into smaller sub-programs based on functionality delivered, we can achieve parallelization via Program Slicing, which can be executed in parallel.

Program	**Slice 1**	**Slice 2**

1 2 . . . . . 999 1000	2 8 54 56 98 999	1 3 53 57 . . 997 1000

Figure 3: Slicing of a Large Program based on output variables at line number 999 & 1000.

Example showing slicing of a program based on output variables:

1. Read (n)
2. I:=1
3. Sum:=0
4. Product:=1
5. While I < n do
6. Sum := sum + I
7. Product := product * I
8. I := I + 1
9. Write (sum)
10. Write (product)

Figure 4: Program fragment used for Slicing

3. PROPOSED METHODOLOGY

This work proposes the methodology for restructuring procedure oriented programs say C [9] Language program in the same relative abstraction level in the form of sequence of automatic and semi-automatic techniques. The input program is divided into chunks of smaller programs called slices, & creating threads for each, so that these threads can be executed in parallel by giving them to a multiprocessor machine.

This work deals with the implementation of different intermediate graphical [6,7] representations and tabular representations for an input source program such as the Control flow Graph, the Data flow Graph and Data flow Table. Once a graphical representation and tabular representation of an input program is obtained, slicing is performed on the program using its processing element, which is the output of the program and applying Weiser's algorithm proposed by Weiser M, to obtain a static backward slice.

C. *Program Slice [3]*

A program slice consists of the parts of a program that (potentially) affect the values computed at some point of interest. Such a point of interest is referred to as a slicing criterion, and is typically specified by a pair (program point, set of variables). The parts of a program that have a direct or indirect effect on the values computed at a slicing criterion C constitute the *program slice with* respect to criterion C. The task of computing program slices is called program slicing. Weiser defined a program slice S as a reduced, executable program obtained from a program P by removing statements, such that S replicates part of the behavior of P.

D. Program Representations [8].

In this section, we study about the intermediate representation of a sample program and the methods followed to construct this representation.

1. Control Flow Graph (CFG) and Control Flow Table (CFT) [1,2]

A Control Flow Graph is a directed graph with a unique entry node START [3] and a unique exit node STOP, where each node is a statement in the program. There is a directed edge from node *P* to node *Q* in the control flow graph if control may flow from block *P* directly to block *Q*.

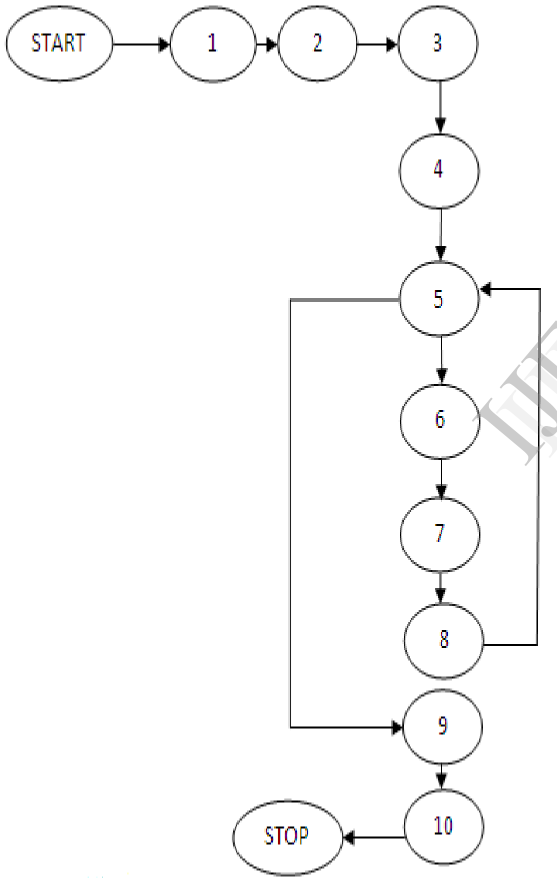

Figure 5: Control Flow Graph for the program given in Figure 4.

We propose a technique [1] for the formation of a four column Control Flow Table (CFT) to represent CFG of each program. The first and second column contains statement numbers of start and end statements of collapsible straight-line sequence of basic block. If the block contains the single statement, the first & second columns contain the same statement number. The third and fourth columns contain statement numbers of alternate control transits ex., IF-ELSE, WHILE and FOR.

Table 1. The Control Flow Table for the Program in Figure 3

START	**END**	**Transition 1 / Next Jump**	**Transition 2 / Alternate Jump**
1	5	6	9
6	8	5	
9	10E		

2. Data Flow Graph (DFT)

The data flow concept was first proposed by Dennis as a radically new way of organizing on large scale non sequential computations. While a control flow graph can be used to capture some control flow information about a program particularly, looping programming construct. Program slicing needs information about definitions and references of data items. Such information can be captured with data flow graph which holds information about the data items definitions and references.

The above designed CFG is used to design Data Flow Graph (DFG) in the form of Data Flow Table (DFT)[1]. Each entry in the DFT indicates the referenced and defined items of that statement. These entries are organized in the CFG order. The sample entries are shown in the **Table 2**. The data items in a “C” program are defined by scanf or read statements, the left side attribute of the arithmetic expressions. The data items in a “C” program are referenced by printf or write statements, the right side attributes of the arithmetic expressions and the attributes of control predicates.

Table 2. The Data Flow Table for the Program in Figure 3

Statement Number	Defined variable	Referenced variable
1	n	--
2	i	--
3	sum	--
4	product	--
5	--	i, n
6	sum	sum, i
7	product	product, i
8	i	i
9	--	sum
10	--	product

In program representation for static slicing, the data flow analysis is concerned with how data flows through a program as the execution of the program progresses i.e. how data flows along CFG. In slicing for behavior however, the interest is not in where the data is going, but in from where it comes from. To understand the behavior of a data items at some point, the affected statements from where the data comes from, need to be know. **This is the reason we traverse backward along CFG during the slicing process.**

3. Weiser's Program Slicing [8]

Weiser defined a program slice S as a reduced, executable program obtained from a program *P* by removing statements, such that *S* replicates part of the behaviour of *P*. The slice consists of parts of a program that potentially affect the values computed at some point of interest. Such a pair of interest is referred to as slicing criterion, and is physically specified by a pair:

<Program statement (n), set of Variables (V)>

We represent 'n' as the statement number along control flow path and *V* as the set of variables on which slicing is required. The parts of a program that have a direct or indirect effect on the values computed at a slicing criterion *C* constitute the program slice with respect to the criterion *C*. This program slicing is a static slicing as the process of obtaining slices does not concern with input or output values. With reference to the program given in Figure 3, we can slice it into 2 parts based on Output Variables

1st Slice pair: <9,Sum>	**2nd Slice pair: <10,Product>**
(1) read(n);	(1) read(n);
(2) i := 1;	(2) i := 1;
(3) sum := 1;	(4) product := 1;
(5) while (i<=n) do	(5) while (i<=n) do
{	**{**
(6) sum := sum + i;	(7) product := product * i;
(8) i := i + 1	(8) i := i + 1
}	**}**
(09) write(sum);	(10) write(product);
A slice of the program w.r.t. criterion (09, sum).	**A slice of the program .t. criterion 0,Product).**

Figure 6: Program in Figure 3, is sliced into two Parts, which are independent to each other.

These slices are independent of each other, so they can be made to run parallel on multi-core architecture machines.

CONCLUSION AND FUTURE WORK

The objective of this work is to slice the given program and create multiple threads, so that when these threads are presented to multicore machine, where they can be executed simultaneously according to the flow of dependence. We are restructuring the given input program in the same relative abstraction level to suit the target environment. We make use of flow graph techniques such as Control flow graph and Data flow graph and used Weiser's algorithm for slicing the programs. The proposed methodology can be applied to procedure oriented programs which are highly structured. Also the above methodology can be applied to legacy programs and try to extract independent slices and make them to execute on multi-core machines.

The intermediate representation of the program can be used to select test cases in regression testing, for software debugging and in many other applications. The methodology can further be extended to handle object-oriented programs as well. The Control flow graph of the given program can be automated for handling complex programs as well. Also identifying variables for slicing can be studied in a novel way.

REFERENCES

[1] Dr.Shivanand M.Handigund. Reverse Engineering of Legacy COBOL Systems.Doctoral dissertation, IIT, Bombay.

[2] "An Ameliorated Methodology for the design of Object Structures from legacy 'C' Program" by Dr.Shivanand M.Handigund, Rajkumar N.Kulkarni, 2010 International Journal of Computer Applications (0975 – 8887) Volume 1 – No. 13

[3] Pankaj Jalote. An Integrated Approach to Software Engineering, Third Edition, Narosa Publishing House.

[4] G.A.Venkatesh. The semantic approach to program slicing. In Proceedings of the ACMSIGPLAN'91 Conference on Programming Language Design and Implementation, pages 107–119, 1991. SIGPLAN Notices 26(6).

[5] R. Gupta and M.L. Soffa. A framework for generalized slicing. Technical report TR-92-07, University of Pittsburgh, 1992.

[6] Ferrante J., Ottenstein K. J., Warren J. D., The Program Dependence Graph and its use in Optimization, ACM Transactions on Programming Languages and Systems, Vol. 9, No. 3, July 1987.

[7]Ottenstein K. J., Ottenstein L. M., The program dependence graph in a software development environment, Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pages 85-97, July 1995.

[8] Paul C. Jorgensen, A Craftsman's Approach - 2nd Edition.

[9] T. D. Brown Jr., "C for Basic Programmers", Tata McGraw Hill Publishing Company Limited, New Delhi, 1992