INVESTIGATING THE RELATIONSHIP BETWEEN THREADS AND PROGRAM SLICES

By

ABDUNNASSAR USMAN

Bachelor of Technology

Government Engineering College, Trichur

University of Calicut

Kerala, India

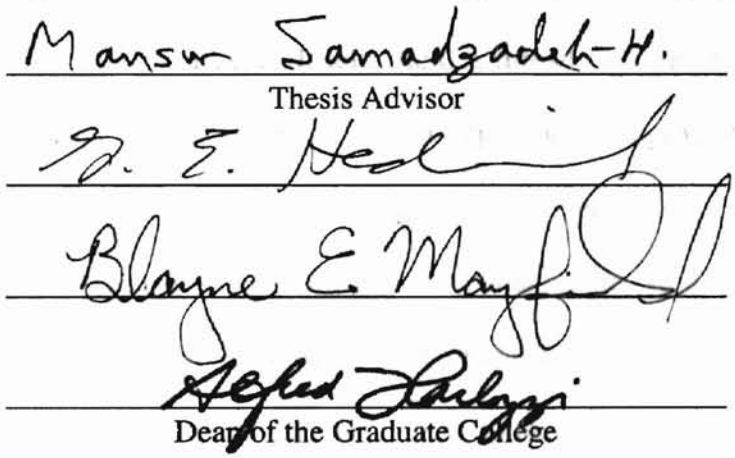
1996

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December 2000

INVESTIGATING THE RELATIONSHIP BETWEEN

THREADS AND PROGRAM SLICES

Thesis Approved:



Thesis Advisor

Dean of the Graduate College

PREFACE

A program slice is a sequence of instructions through a program that is capable of reproducing part of the program's behavior. A static slice can be a reuse component. Slicing exposes the control and data dependencies that affect portions of interest in a given program.

A thread is a sequence of program execution. Data flow and control flow characterize a thread; and data and control dependencies hold a thread together as a cohesive unit. Threads can be reuse candidates (as in concurrent object-oriented programming).

A relationship between threads and program slices seems to emerge when their concepts are juxtaposed. This study was an investigation to verify and validate the relationship. It examined research fundamentally relevant to the concepts of threads and program slices, and tabulated similarities on several points of comparison.

The concept of a dynamic slice is comparable to that of a thread of computation. Slices can be parallelly executable entities. Discernible commonalities were observed when the concepts of threads and program slices were juxtaposed from a reuse perspective. Traditional parallelization techniques and slicing were observed to be homologous.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my graduate advisor, Dr. Mansur H. Samadzadeh for his supervision and guidance through the duration of this research, and for his keen interest in my progress. I thank him for suggesting the topic for research and for his invaluable input along the way.

I would also like to thank Drs. G. E. Hedrick and Blayne E. Mayfield for serving on my thesis committee. I am greatly indebted to Dr. J. D. Carlson of the Biosystems and Agricultural Engineering Department for employing me for over two years, during which time I was extended generous financial support and uncommon good will.

My parents have stood by me through a rather long stay here at Oklahoma State University. I immensely appreciate their patience and understanding.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table Page

CHAPTER I

INTRODUCTION

A program slice is a sequence of instructions through a program that is capable of reproducing part of the program's behavior. Slicing deletes from consideration those statements in the program that are irrelevant to a computation or a set of computations of interest. A thread is a sequence of program execution.

Since the inception of the idea of program slicing, the plausibility of its application to parallelizing program execution has been suggested by some prominent researchers in the area, e.g., Mark Weiser [Weiser 84] stated the following.

> Because slices execute independently, they are suitable for parallel execution on multiprocessors without synchronization or shared memory. ... Parallel execution of slices might be particularly appropriate for distributed systems, where shared memory is impossible and synchronization requires excessive handshaking.

Agrawal, in his 1991 Ph.D. dissertation [Agrawal 91], hints at the conceptual similarity of program slices to threads, as follows:

> Conceptually a program may be considered as a collection of threads, each computing a value of a program variable. Several threads may compute values of the same variable. Portions of these threads may overlap one another. The more complex the control structure of the program, the more complex the intermingling of these threads. Static program slicing isolates all possible threads computing a particular variable. Dynamic slicing, on the other hand, isolates the unique thread computing the variable for the given inputs.

Both threads and program slices can be viewed in terms of computations through a program. The thesis takes a look at the relationship between threads and program slices as reusable software units [Samadzadeh and Zand 99] [Zand and Samadzadeh 95]. It examines research fundamentally relevant to the concepts of threads and program slices, and tabulates similarities on several points of comparison.

The rest of the thesis is organized as follows. Chapter II presents a brief overview of program slices. Chapter III discusses the concept of threads. Information on threads and program slices are juxtaposed, in Chapter IV, to bring out their similarities. Chapter V makes concluding remarks and identifies directions for future inquiry.

CHAPTER II

PROGRAM SLICES

Program slicing [Binkley and Gallagher 96] [Tip 94] involves operations on source code that isolate part of the behavior of a program when viewed from a point with in the program. The process yields a relevant subset of program statements, referred to as a *program slice*. Slices are computed with respect to a slicing criterion. Program slicing has been broadly classified as *static* or *dynamic* based on the slicing criterion. *Conditioned slicing* attempts at providing for a more general slicing model.

Dependence analysis forms the core of program slicing. Slicing uses terms, concepts and techniques from program dependence theory [Korel and Rilling 98]. Directed graphs that capture data dependence and control dependence relationships are used in most slicing algorithms as intermediate program representations.

2.1 Static Slices

Static slicing refers to slicing methods that preserve the behavior of the program for all possible executions. A static slice consists of the subset of program statements that affect a set of variables at a particular location in the program for all input combinations. In other words, input values are not considered in computing the slice. The static slicing

criterion consists of a pair <*V*, *p*>, where *V* is the set of variables of interest and *p* is the point of interest in the program. A sample program and its static slice is shown in Figure 1 (example taken from [Tip 94]).

Much of program slicing literature, to date, focuses on slicing sequential programs. Cheng [Cheng 93] [Cheng 97] and Krinke [Krinke 98] detail static slicing of threaded programs and suggest modified intermediate program representations. Zhao et al. [Zhao et al. 96] describe static slicing of concurrent object-oriented programs. Dwyer et al. [Dwyer and Hatcliff 99] [Dwyer et al. 99] and Hatcliff et al. [Hatcliff et al. 99] present static slicing of multi-threaded Java programs, and formalize some notions of program dependence in this context.

```
1     input (n);
2     i := 1;
3     sum := 0;
4     prod := 1;
5     while i <= n do
      begin
6        sum := sum + i;
7        prod := prod * i;
8        i := i + 1;
      end;
9     output (sum);
10    output (prod);
```

a. A sample program

```
1     input (n);
2     i := 1;
4     prod := 1;
5     while i <= n do
      begin
7        prod := prod * i;
8        i := i + 1;
      end;
10    output (prod);
```

b. Static slice

Figure 1. A sample program, and its static slice corresponding to slicing criterion <*V*, *p*>, where $V = \{\text{prod}\}$ and $p = 10$

2.2 Dynamic Slices

Dynamic slicing refers to a family of program slicing methods that use run-time information in the computation of a slice [Korel and Rilling 98]. A dynamic slice consists of the subset of statements in a program that affect a set of variables at a point of interest

in the program when the program is executed with a specific set of input values. A dynamic slicing criterion specifies a set of variables of interest V, a point of interest p in the program, and a set of input values I. The concept of an *execution trace* is central to dynamic slicing. Figure 3 gives an example of executable and non-executable dynamic slices of the program in Figure 2 (examples from [Korel and Rilling 98]). The idea of dynamic slicing has also been applied to distributed programs [Cheng 93] [Duesterwald et al. 92] [Korel and Ferguson 92].

```
1     input (n, a);
2     max := a[1];
3     min := a[1];
4     i := 2;
5     s := 0;
6     while i <= n do
      begin
7        if max < a[i] then
         begin
8           max := a[i];
9           s := max;
         end;
10       if min > a[i] then
         begin
11          min := a[i];
12          s := min;
         end;
13       output (s);
14       i := i + 2;
      end;
15    output (max, min)
```

a. A sample program

```
1(1)   input (n, a)
2(2)   max := a[1];
3(3)   min := a[1];
4(4)   i := 2;
5(5)   s := 0;
6(6)   i <= n
7(7)   max < a[i]
8(8)   max := a[i];
9(9)   s := max;
10(10) min > a[i]
13(11) output (s);
14(12) i := i + 2;
6(13)  i <= n
15(14) output (max, min)
```

b. Execution trace

Figure 2. A sample program, and its execution trace for input n=3, a=(1,2,3)

2.3 Conditioned Slices

A dynamic slicing criterion specifies a specific computation corresponding to a single initial state (i.e. a single set of input values). A static slicing criterion, on the other hand, specifies no initial state, and hence implies the set of all possible initial states.

Conditioned slicing [Canfora et al. 98] encompasses both notions by providing for a generalized slicing criterion. A conditioned slice is extracted by specifying a static slicing criterion together with a set of initial states that meet a particular condition. The condition is specified as a first order logic formula on a subset of input variables. The notion of a *quasi-static slice* has been defined to accommodate partial initial state specification in the slicing criterion. *Simultaneous dynamic slicing* is performed with respect to a slicing criterion that specifies a finite set of initial states. Conditioned slicing subsumes both these notions as well. The increased flexibility in specification of the slicing criterion translates to increased control of refinement in decomposition.

```
1       input (n, a);
2       max := a[1];
4       i := 2;
6       while i <= n do
        begin
7          if max < a[i] then
           begin
8             max := a[i];
           end;
14         i := i + 2;
        end;
15      output (max, min)
```

a. Executable

```
1       input (n, a);
2       max := a[1];
4       i := 2;
6       while i <= n do
        begin
7          if max < a[i] then
           begin
8             max := a[i];
           end;
        end;
15      output (max, min)
```

b. Non-executable

Figure 3. Executable and non-executable dynamic slices of program in Figure 2 for the slicing criterion given by <*I*, *V*, *p*>, where I = {n=3, a=(1,2,3)}, V = {max} and p=15

2.4 Applications of Program Slicing

Slicing has several applications. It was introduced as a debugging technique [Weiser 82]. It has been used in software maintenance and regression testing. Slicing has been used as a component in applications that involve restructuring, re-engineering,

comprehension, [Binkley and Gallagher 96] [Tip 94] and reuse [Cimitile 95a] [Cimitile 95b]. Lakhotia and Deprez describe a method of restructuring programs into small cohesive functions, using slicing [Lakhotia and Deprez 98]. Slicing has been used in deriving ADTs from FORTRAN programs [Liu and Ellis 93]. It has found use as an objective basis for cohesion measurement and in software validation. Weiser suggests the use of static slicing for parallel execution in distributed systems [Weiser 83] [Weiser 84]. Static slicing has been used as a component in constructing finite-state models of sequential and threaded software systems [Dwyer and Hatcliff 99] [Dwyer et al. 99] [Hatcliff et al. 99]. Dynamic slicing has been used for software debugging, software maintenance, program comprehension and software testing [Korel and Rilling 98].

2.5 Tools Based on Slicing

Tools based on program slicing have largely been built as research prototypes in academia. Commercial availability of such tools is in its incipient stages. CodeSurfer is a commercially available program analysis and understanding tool, from GrammaTech, based on static slicing [GrammaTech 00]. Currently, it supports programs written in C. Unravel is a prototype static slicing tool from National Institute of Standards and Technology [Unravel 00]. Spyder is a prototype dynamic slicing based debugging tool developed in the early 90s at Purdue University [Spyder 00]. Samadzadeh and Wichaipanitch [Samadzadeh and Wichaipanitch 93] detail the implementation of an interactive debugging tool for C based on dynamic slicing. Samadzadeh and Hsiao [Samadzadeh and Hsiao 93] describe the implementation of a parallel program slicer.

CHAPTER III

THREADS

A thread is an independent sequence of program execution. When threads are executed concurrently and share resources (memory, I/O stream, etc.), control of access becomes an issue; synchronization mechanisms are employed to try and prevent known concurrency problems. Section 3.1 lays out the variety in notions of threads relevant to this study and sets the stage for looking at threads by their common denominator, i.e., as parallelly executable entities, in Section 3.2. Parallelization of a sequential program involves operations on the program that identifies and extracts potential parallelism; Section 3.3 briefly highlights some observations in this context. Section 3.4 samples parallelization tools and programming aids.

3.1 Notions of Threads

A thread is a unit of program execution. Notions of a thread differ in shades contextually. References to the term *thread* in the following conceptual settings are relevant to this discussion: Thread of a Computation, Thread of a Process, and Thread in Multi-threaded Programming.

3.1.1 Thread of a Computation

A dynamic execution sequence with respect to source text is referred to as a thread.

Ravi Sethi [Sethi 89] explains:

> A dynamic computation can be visualized as a thread laid down by the flow of control through the static program text. Imagine program points appearing before the first instruction, between any two adjacent instructions, and after the last instruction. The thread of a computation consists of the sequence of program points that are reached as control flows through the program text.

Sebesta refers to it as a *thread of control*. He notes [Sebesta 99]:

> One useful technique for visualizing the flow of execution through a program is to imagine a thread laid on the statements of the source text of the program. Every statement reached on a particular execution is covered by the thread representing that execution. Visually following the thread through the source program traces the execution flow through the executable version of the program. A **thread of control** in a program is the sequence of program points reached as control flows through the program.

3.1.2 Thread of a Process

A thread is a sequence of execution in the context of a process. A process is a program in execution. In the UNIX environment a C program in execution is composed of the following parts [Stevens 93]:

- Text segment: contains machine instructions executed by the CPU.
- Initialized data segment: contains global variables specifically initialized in the program.
- Uninitialized data segment: contains uninitialized global variables in the program.

- Stack: each function call involves pushing on to the stack, some information required to continue the normal execution sequence of the program after the control is returned to the caller. The called function also allocates space on the stack for its automatic and temporary variables.
- Heap: space for dynamic memory allocation is provisioned from the heap.

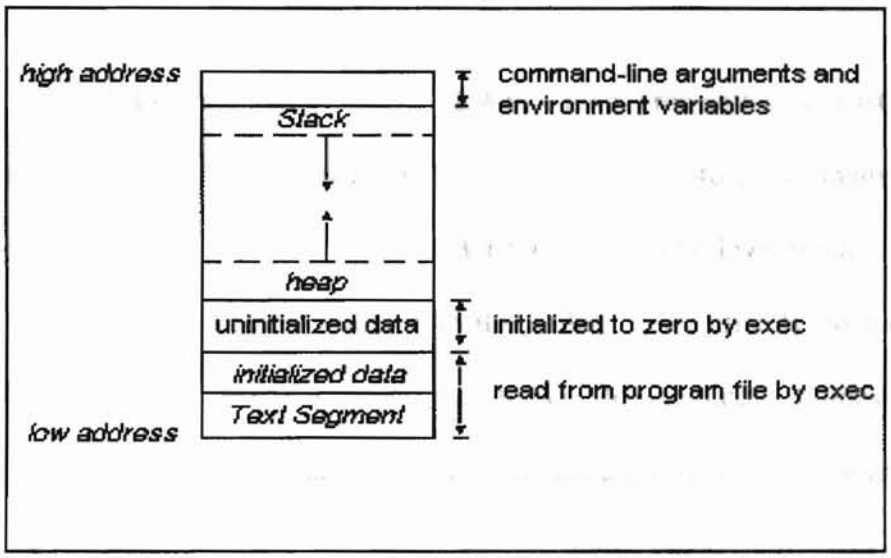


Figure 4. Typical memory arrangement of a C/UNIX process

The logical arrangement of memory for the process could be represented as shown in Figure 4 [Stevens 93].

Program Counter, register values, and I/O streams constitute part of the other information relevant to the state of a process. A traditional UNIX process has a single thread of execution. Stevens [Stevens 98] observes that in a multi-threaded process, all threads share process instructions in the text segment, most data, open files, signal handlers and signal dispositions, current working directory, and user and group Ids. But they have their own thread ID, set of registers (program counter, stack pointer, etc.), stack

(for local variables and return addresses), `errno`, signal mask, and priority. Lewis and Berg [Lewis and Berg 96] comment about the concept of a thread within a process, as follows:

> A thread is an abstract concept that comprises everything a computer does in executing a traditional program. It is the program state that gets scheduled on a CPU, it is the thing that does the work. If a process comprises data, code, kernel state and a set of CPU registers, then a thread is embodied in the contents of those registers – the program counter, the general registers, the stack pointer, etc., and the stack. A thread, viewed at an instant of time, is the state of the computation.

The kernel schedules *lightweight processes* (LWPs). Each process has at least one LWP. Threads are scheduled by binding them to LWPs in either a one-to-one, a many-to-one, or a many-to-many fashion. Solaris [Lewis and Berg 96] uses a two-level model, which uses a variation of the many-to-many model with the ability to specifically do a one-to-one binding request. Windows NT [Custer 93] uses the one-to-one model. It does not have a concept of LWPs distinct from that of threads. Threads are kernel-schedulable entities in Windows NT [Custer 93].

3.1.3 Thread in Multi-Threaded Programming

A thread is a unit of abstraction of a program in multi-threaded programming paradigm [Lewis and Berg 96] [Pham and Garg 96] [Kleiman et al. 96] [Northrup 96]. Multi-threaded programming is a concurrent programming methodology that gives users the ability to write programs which exploit subprogram-level concurrency [Sebesta 99] within the shared-memory context of a process. Its power is equivalent to programming with multiple processes using shared memory; but thread creation and manipulation

within a process have significantly lesser overhead. Concurrency in accessing shared resources cause issues of synchronization. Sebesta observes that "two of the primary facilities that concurrent languages must provide are mutually exclusive access to shared data structures (competition synchronization) and cooperation among tasks" [Sebesta 99]. To aid synchronization, language constructs like semaphores, mutexes, monitors, and condition variables are provided, either as a built-in feature of a language (e.g., multi-threaded programming support in Java and Ada 95) or as extensions through language-specific bindings to thread libraries (e.g., multi-threaded programming support for C/C++ and PASCAL). Thread creation and manipulation facilities are also provided in similar fashion.

3.2 Threads as Parallelly Executable Entities

Typically, the term "multi-threaded program" refers to programs that correspond to a single process with concurrent threads of execution using shared memory. Distributed programs (programs implemented as communicating concurrent processes) have multiple threads of execution, which are synchronized by message passing. In general, threads can be viewed as parallelly executable entities. Krinke uses the terms, *threaded* and *concurrent*, interchangeably when he discusses slicing of threaded programs [Krinke 98]. Anderson et al. [Anderson et al. 97] describe this general concept of threads, and list the different models for organization of threads and address spaces.

3.3 Benefits of Threads

Execution efficiency in a multi-processor environment is perhaps the most prominent advantage in using concurrent threads of execution. Multiple threads performing independent tasks with localized interactions can increase the responsiveness of programs. Concurrent programming uses threads as an abstraction paradigm. Reducing the complexity of writing large programs by breaking it into smaller interacting independent units is a major motivation behind the development of concurrent programming [Pham and Garg 96]. The encapsulation provided by a process, the facilities for inter-process communication, and the mechanisms for specifying hierarchical relationship make processes naturally suited for implementation of object-oriented design. Objects can be implemented using concurrent processes [Sommerville 00].

3.4 Parallelization of Sequential Programs

Dependence analysis is used in parallelizing compilers to detect and extract potential parallelism, and usually serves as the first step in the parallelization process [Wolfe 96b]. In many algorithms, there exists intrinsic parallelism that can be detected and used quite easily. The form of parallelism in such cases is referred to as *trivial parallelism* [Lewis and El-Rewini 92]. A supervisor/worker model [Lewis and El-Rewini 92] [Pham and Garg 96] or a divide-and-conquer approach is generally sufficient to exploit trivial parallelism. But, finding and exploiting parallelism in general is much more difficult. Much of the research effort in parallelizing sequential programs is focussed on the problem of parallelization of loops. In sequential programs there could be potential

parallelism in branches and functions/procedures, but extraction of parallelism from loops holds the greatest promise for high performance [Lewis and El-Rewini 92] [Wolfe 96a]. There are limitations to a wholly static analysis approach to parallelization, e.g., when dealing with loops that possess inter-iteration dependencies [Rauchwerger and Padua 95]. *Speculative multithreading* [Akkary and Driscoll 98] [Kazi and Lilja 98] [Marcuello et al. 98] [Oplinger et al. 99] [Rauchwerger et al. 95] [Rauchwerger and Padua 95] aims at relieving such limitations through run-time parallelization, using control and data dependence speculations in executing threads.

3.5 Parallelization Tools and Programming Aids

Concurrent Pascal and Concurrent C are examples of traditional concurrent programming languages. Concurrent programming support (multiple processes and/or multiple threads) is also available as a built-in feature of languages such as Ada 95 and Java, or as extensions to existing languages such as C and Pascal.

Much progress has been made in the area of Automatic Parallelization. Polaris is a compiler development tool, developed at the University of Illinois at Urbana-Champaign, that automatically parallelizes Fortran 77 programs, for execution on shared-memory multiprocessors [Padua et al. 00]. The SUIF compiler is an infrastructure for collaborative research in optimizing and parallelizing compilers developed by the Stanford Compiler Group as part of the National Compiler Infrastructure project [Lam and Hennessy 00].

CHAPTER IV

JUXTAPOSITION

Selected observations about threads and program slices have been placed side-by-side in Table I, in an effort to bring out their similarities. Our focus here is on reuse. It becomes apparent that there exists a good intersection in the concepts of threads and program slices.

Table I. Juxtaposition of selected information on Threads and Program Slices from a Reuse perspective

JUXTAPOSITION POINT	PROGRAM SLICES	THREADS
Direct Reuse Scenario	Slicing has been used for identifying reusable functions. It has been used in strategies for extracting components for reuse.	Reuse of concurrent code is a major aim of concurrent object-oriented programming.
Cohesion	Slices have been used as a measure of cohesion. Slicing has been used to restructure programs into cohesive functions.	Independence in execution characterizes a thread as a cohesive entity. Good use of threads must make sure that threads have highly localized interaction with other threads.
Comprehension	Slicing is especially suited for isolating focus on a program behavior of interest.	Abstraction for better comprehension is motivation also for concurrent programming.
Decomposition	Slices are composed of a subset of statements in a program. It reduces problems of program size by allowing focus of attention on a subcomputation corresponding to a slicing criterion.	Concurrent programming is a paradigm for decomposing programming problems. Reducing the complexity of writing large programs by breaking it into smaller interacting independent units is a major motivation behind the development of concurrent programming.
Restructuring	Slicing has been used for restructuring of programs into cohesive functions. It has been applied to the problem of salvaging useful components from software systems that are deemed to be beyond repair.	Restructuring sequential programs into threaded equivalents is a major research area.

Subprogram abstraction	A slice is a subset of lines in a program. Slicing enables focus on a subcomputation that is relevant to a slicing criterion. Slicing has been used to create cohesive subprograms (functions).	Threads in multi-threaded programming are subprograms executable in parallel. Run-time parallelization schemes use successive iterations of loops as concurrent threads.
Object-orientation	Slicing of object oriented programs has been studied. Object-orientation using slices relevant to a data structure looks plausible. We could not find any direct reference to this in slicing literature, however slicing has been used in deriving ADTs from FORTRAN programs.	Objects can be implemented using concurrent processes. Concurrent object-oriented programming aims at combining the benefits of both concurrent code and object-oriented programming.
Uncertainty	Uncertainty is not directly obvious in a slice. But, the notion of a static slice corresponds to all computation that *may* affect a given behavior of interest. This involves a certain notion of uncertainty, which could be exploited.	Speculative multithreading involves parallel execution of successive loop iterations using control and data dependence speculations.
Formalism	General and application-specific formalism in the notion of a slice exists. Formalism exists in program representations, dependence notions, and techniques for slicing. But the notion of a program slice is quite flexible owing to its psychological basis. It could be applied to programs at any level of abstraction – e.g., high-level, assembly, or machine code. There is also room for new application-specific notions of slices.	A thread is an abstract concept that stretches across the software/hardware interface. Formalism exists in program representations, dependence analysis, transformations, etc., and in the form of language constructs for concurrent programming and software/hardware implementation support for multiple threads.
Dependence Analysis	At the core of slicing.	Major part in detecting potential parallelism in programs.
Timing	Timing is not inherent to slices. But a notion of timing emerges when computations corresponding to slices (especially in executable slices) are considered.	Execution has some notion of timing inherent to it. A thread is a sequence of execution. Timing relative to other threads is also highly relevant.

Run-time information	Dynamic slicing uses run-time information for computing slices.	Speculative multithreading uses run-time information for parallelizing loops.
Slicing threads	Some work has been done in this area.	N/A
Threading slices	N/A	Weiser suggested threading of slices in distributed applications where shared memory is unavailable. Not much has been done, since then, in using program slices for automatic parallelization of programs.

CHAPTER V

SUMMARY, CONCLUSION, AND FUTURE WORK

5.1 Summary and Conclusion

The concept of a dynamic slice is comparable to that of a thread of computation [Agrawal 91]. Weiser demonstrated that static slices can be parallelly executable entities [Weiser 83] [Weiser 84]. In this thesis we presented brief overviews of program slices and threads, and juxtaposed their concepts from a reuse perspective. It was observed that there are discernible commonalities in their concepts.

Program slices were introduced with verification, in 1979, as a natural psychological process in abstracting programs [Weiser 79]. Programmers use this while debugging [Weiser 82]. It fundamentally asks the questions: "Given a behavior of interest at a point of observation what set of predecessors affect it?" (as in a backward slice) or "What set of successors are affected by the behavior at a point of observation?" (as in a forward slice). The process of parallelizing programs fundamentally asks the same questions to detect and extract potential parallelism. In this sense threads and program slices are homologous.

In the foreword to the *Information and Software Technology* (Special Issue on Program Slicing) [Weiser 98], Weiser reiterates the psychological underpinnings of

slices, re-advocates research into the use of the notion of program slices for parallelizing programs, and suggests slices as an abstraction paradigm in programming languages. The results from the juxtaposition of the concepts of threads and program slices, presented here, lead us to look with increased confidence in these directions.

5.2 Future Work

The increased confidence in the relationship between threads and program slices is not without qualification. Does program slicing really show promise in parallelizing sequential programs? Weiser seems to be the only one who has pursued this direction. In evaluating Weiser's claim one is led to ask:

1. What kinds of programs has Weiser (or others) parallelized using program slicing?
2. A closely related question is, *What kinds of sequential programs have multithreaded equivalents?* Or, looking at the same question in the reverse direction, *What kinds of multithreaded programs can have completely sequential equivalents?*

Program slicing seems to be of no use for parallelization of loops, which is the major focus of parallelization efforts on sequential programs. But program slicing has been successfully used in the process of decomposing programs into cohesive functions. Divide-and-conquer solutions are inherently parallelizable and it is likely that program slicing is effective in isolating parallelizable segments from such programs. If static slices of a given program are computed with respect to each of its output variables at the last line of the program, each slice can be delegated to a "worker" process and their results can be managed by a "supervisor" process.

An empirical study of a wide range of programs (both sequential and multithreaded) and their slices is necessary to decide conclusively on the relationship between threads and program slices. The non-availability of adequate and dependable program slicing tools is a major obstacle at this time. Experimental inquiries into the relationship could be made at several levels. Two of the basic exploratory queries that could be asked are listed below:

1. Can slices of a program be an indicator of its threadability? In other words, can slices of a program indicate whether it can or cannot be implemented using multiple threads?
2. Do certain types of programs produce slices that display a common property when seen in relation to threads? Could this property of the slices not be a means of classifying programs?

Techniques used in program slicing and those used in program parallelization seem to have considerable region of intersection. A detailed comparative study of the techniques can give greater insight into their relationship.

Program slicing is a maturing research area. On the other hand, threads, as concurrent sequences of execution, have been studied for over three decades. As program slicing research progresses it is possible that more similarities would emerge between program slices and threads. There is good flexibility in the definition of a program slice and some interesting new notions of program slices are being introduced.

REFERENCES

[Agrawal 91] H. Agrawal, "Towards Automatic Debugging of Computer Programs," Ph.D. Dissertation, Department of Computer Sciences, Purdue University, West Lafayette, IN, 1991.

[Akkary and Driscoll 98] H. Akkary and M. A. Driscoll, "A Dynamic Multithreading Processor," *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 31)*, pp. 226-236, Dallas, TX, December 1998.

[Anderson et al. 97] T. Anderson, B. Bershad, E. Lazowska, and H. Levy, "Thread Management for Shared-Memory Multiprocessors," In *The Computer Science and Engineering Handbook*, Edited by: Allen B. Tucker, pp. 1665-1676, CRC Press, Boca Raton, FL, 1997.

[Binkley and Gallager 96] D. W. Binkley and K. B. Gallager, "Program Slicing," In *Advances in Computers*, Vol. 43, Edited by: Marvin Zelkowitz, pp. 1-50, Academic Press, San Diego, CA, 1996.

[Confora et al. 98] G. Canfora, A. Cimitile, A. De Lucia, "Conditioned Program Slicing," In *Information and Software Technology*, Vol. 40, No. 11-12, pp. 595-607, November 1998.

[Cheng 93] J. Cheng, "Slicing Concurrent Programs," *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, Lecture Notes in Computer Science, Vol. 749, pp. 232-245, Springer, Berlin, Germany, 1993.

[Cheng 97] J. Cheng, "Dependence Analysis of Parallel and Distributed Programs and Its Applications," *Proceedings of the International Conference on Advances in Parallel and Distributed Computing*, pp. 370-377, Shanghai, China, March 1997.

[Cimitile et al. 95a] A. Cimitile, A. De Lucia, and M. Munro, "Identifying Reusable Functions Using Specification Driven Program Slicing: A Case Study," *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'95)*, pp. 124-133, Opio (Nice), France, October 1995.

[Cimitile et al. 95b] A. Cimitile, A. De Lucia, and M. Munro, "Qualifying Reusable Functions Using Symbolic Execution," *Proceedings of the 2nd Working Conference on Reverse Engineering*, pp. 178-187, Toronto, Canada, July 1995.

[Custer 93] H. Custer, *Inside Windows NT*, Microsoft Press, Redmond, WA, 1993.

[Duesterwald et al. 92] E. Duesterwald, R. Gupta, and M. Soffa, "Distributed Slicing and Partial Re-execution for Distributed Programs," *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, pp. 329-337, New Haven, CT, August 1992.

[Dwyer and Hatcliff 99] M. B. Dwyer and J. Hatcliff, "Slicing Software for Model Construction," *Proceedings of the ACM Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pp. 105-118, San Antonio, TX, January 1999.

[Dwyer et al. 99] M. B. Dwyer, J. C. Corbett, J. Hatcliff, S. Sokolowski, and H. Zheng, "Slicing Multi-threaded Java Programs: A Case Study," Technical Report 99-7, Department of Computing and Information Sciences, Kansas State University, Manhattan, KS, 1999.

[GrammaTech 00] GrammaTech CodeSurfer Software Analysis and Understanding Tool, http://www.grammatech.com/products/codesurfer/codesurfer_index.html, access date: April 13, 2000.

[Hatcliff et al. 99] J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, and H. Zheng, "A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives," *Proceedings of the International Symposium on Static Analysis (SAS'99)*, pp. 1-18, Venice, Italy, September 1999.

[Kazi and Lilja 98] I. H. Kazi and D. J. Lilja, "Coarse-Grained Speculative Execution in Shared-Memory Multiprocessors," *Proceedings of the 1998 International Conference on Supercomputing (ICS'98)*, pp. 93-100, Melbourne, Australia, July 1998.

[Kleiman et al. 96] S. Kleiman, D. Shah, and B. Smaalders, *Programming with Threads*, SunSoft Press/ Prentice Hall PTR, Mountain View, CA, 1996.

[Korel and Ferguson 92] B. Korel and R. Ferguson, "Dynamic Slicing of Distributed Programs," *Applied Mathematics and Computer Science Journal*, Vol. 2, No. 2, pp. 199-215, December 1992.

[Korel and Rilling 98] B. Korel and J. Rilling, "Dynamic Program Slicing Methods," *Information and Software Technology*, Vol. 40, No. 11-12, pp. 647-659, November 1998.

[Krinke 98] J. Krinke, "Static Slicing of Threaded Programs," *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pp. 35-42, Montreal, Canada, June 1998.

[Lakhotia and Deprez 98] A. Lakhotia and J. Deprez, "Restructuring Programs by Tucking Statements into Functions," *Information and Software Technology*, Vol. 40, No. 11-12, pp. 677-689, November 1998.

[Lam and Hennessy 00] M. S. Lam and J. L. Hennessy, The Stanford SUIF Compiler Group Page, http://suif.stanford.edu, access date: April 13, 2000.

[Lewis and El-Rewini 92] T. G. Lewis and H. El-Rewini, *Introduction to Parallel Computing*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Lewis and Berg 96] B. Lewis and D. J. Berg, *Threads Primer*, Prentice-Hall, Upper Saddle River, NJ, 1996.

[Liu and Ellis 93] L. Liu and R. Ellis, "An Approach to Eliminating COMMON Blocks and Deriving ADTs from FORTRAN Programs," Technical Report, University of Westminster, UK, February 1993.

[Marcuello et al. 98] P. Marcuello, A. Gonzalez, and J. Tubella, "Speculative Multithreaded Processors," *Proceedings of the 1998 International Conference on Supercomputing (ICS'98)*, pp. 77-84, Melbourne, Australia, July 1998.

[Northrup 96] C. J. Northrup, *Programming with Unix Threads*, John Wiley & Sons, Inc., New York, NY, 1996.

[Oplinger et al. 99] J. T. Oplinger, D. L. Heine, and M. S. Lam, "In Search of Speculative Thread-Level Parallelism," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, Newport Beach, CA, October 1999.

[Padua et al. 00] D. A. Padua, J. Torrellas, and R. Eigenmann, Polaris Project Home Page, http://polaris.cs.uiuc.edu/polaris/polaris.html, access date: April 13, 2000.

[Pham and Garg 96] T. Q. Pham and P. K. Garg, *Multithreaded Programming with Windows NT*, Prentice-Hall PTR, Upper Saddle River, NJ, 1996.

[Rauchwerger et al. 95] L. Rauchwerger, N. M. Amato, and D. A. Padua, "Run-Time Methods for Parallelizing Partially Parallel Loops," *Proceedings of the 9th ACM International Conference on Supercomputing (ICS'95)*, pp. 137-146, Barcelona, Spain, July 1995.

[Rauchwerger and Padua 95] L. Rauchwerger and D. A. Padua, "The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization," *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pp. 218-232, La Jolla, CA, June 1995.

[Samadzadeh and Hsiao 93] M. H. Samadzadeh and Ting-Huan Hsiao, "An Interactive Parallel Program Slicer for the Hypercube," *Proceedings of the Twelfth Annual IEEE International Phoenix Conference on Computers and Communications (IPCCC'93)*, pp. 66-72, Tempe, Arizona, March 1993.

[Samadzadeh and Wichaipanitch 93] M. H. Samadzadeh and W. Wichaipanitch, "An Interactive Debugging Tool for C Based on Dynamic Slicing and Dicing," *Proceedings of the Twenty-First Annual ACM Computer Science Conference (CSC'93)*, pp. 30-37, Edited by: Stan C. Kwasny and John F. Buck, Indianapolis, Indiana, February 1993.

[Samadzadeh and Zand 99] M. H. Samadzadeh and M. K. Zand, "Software Houses," In *Encyclopedia of Electrical and Electronics Engineering*, Edited by: John G. Webster, Vol. 19, pp. 473-483, John Wiley & Sons, Inc., New York, NY, 1999.

[Sebesta 99] R. W. Sebesta, *Concepts of Programming Languages*, Fourth Edition, Addison Wesley Longman, Inc., Reading, MA, 1999.

[Sethi 89] R. Sethi, *Programming Languages*, Addison-Wesley Publishing Company, Reading, MA, 1989.

[Sommerville 00] I. Sommerville, *Software Engineering*, Sixth Edition, Addison-Wesley Publishing Company, Reading, MA, 2000.

[Spyder 00] Spyder Debugger Project Page, http://www.cerias.purdue.edu/homes/spaf/spyder.html, access date: April 13, 2000.

[Stevens 93] R. W. Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley Publishing Company, Reading, MA, 1993.

[Stevens 98] R. W. Stevens, *UNIX Network Programming*, Vol. 1, Second Edition, Prentice Hall, Upper Saddle River, NJ, 1998.

[Tip 94] F. Tip, "A Survey of Program Slicing Techniques," Report CS-R9438, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, Netherlands, 1994.

[Unravel 00] The Unravel Program Slicing Tool, http://www.nist.gov/itl/div897/sqg/unravel/unravel.html, access date: April 13, 2000.

[Weiser 79] M. Weiser, "Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Abstraction Method," Ph.D. Thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, MI, 1979.

[Weiser 82] M. Weiser, "Programmers Use Slices When Debugging," *Communications of the ACM*, Vol. 25, No. 7, pp. 446-452, July 1982.

[Weiser 83] M. Weiser, "Reconstructing Sequential Behavior from Parallel Behavior Projections," *Information Processing Letters*, Vol. 17, No. 3, pp. 129-135, October 1983.

[Weiser 84] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, pp. 352-357, July 1984.

[Weiser 98] M. Weiser, "Foreword," *Information and Software Technology*, Vol. 40, No. 11-12, p. 575, November 1998.

[Wichaipanitch 92] Winai Wichaipanitch, "An Interactive Debugging Tool for C Based on Dynamic Slicing and Dicing," Master of Science Thesis, Computer Science Department, Oklahoma State University, Stillwater, OK, 1992.

[Wolfe 96a] M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, Redwood City, CA, 1996.

[Wolfe 96b] M. Wolfe, "Parallelizing Compilers," *ACM Computing Surveys*, Vol. 28, No. 1, pp. 261-262, March 1996.

[Zand and Samadzadeh 95] M. K. Zand and M. H. Samadzadeh, "Software Reuse: Current Status and Trends," *Journal of Systems and Software*, Editorial for the Special Issue on Software Reuse, Vol. 30, No. 3, pp. 167-170, September 1995.

[Zhao et al. 96] J. Zhao, J. Cheng, and K. Ushijima, "Static Slicing of Concurrent Object-Oriented Programs," *Proceedings of the 20th IEEE Annual International Computer Software and Application Conference*, pp.312-320, Seoul, Korea, August 1996.

APPENDICES

APPENDIX A

GLOSSARY

Action	An instance of an instruction in the *trajectory* for a specific input in the context of dynamic slicing, e.g., action X(p) represents an instance of instruction X at position p in the trajectory.
ADT	Abstract Data Type
Conditioned Slicing	Slicing performed by specifying a static slicing criterion together with a set of initial states that meet a particular condition. The condition is specified as a first order logic formula on a subset of input variables. A conditioned slicing criterion is a triple $<F(V_{in}), V, p>$, where: V is a set of variables of interest, p is a point of interest in the program, $F(V_{in})$ is a first order logic formula on a subset of the set of input variables, V_{in}, specifying a desired condition.
Divide-and-Conquer Approach	A solution strategy where the algorithm partitions the problem into independent subproblems, solves the subproblems recursively, and combines their solutions to solve the original problem.
Dynamic Slicing	Slicing performed by specifying values for each element in the set of input variables. A dynamic slicing criterion specifies an initial state along with a static slicing criterion. See also *slicing criterion.*
Execution Trace	A sequence of *actions* performed on a particular input in the context of dynamic slicing. Also referred to as *Trajectory.*
LWP	Light Weight Process. Kernel-schedulable entity in multi-level architectures for Multithreaded Programming.

Multithreading	Implementation of solutions incorporating multiple threads of control. The scenario may involve either multiple threads and a single address space, or multiple threads and multiple address spaces.
Multithreaded	Concurrent; unless the Multithreaded Programming paradigm is obvious from context.
Multithreaded Programming	A programming paradigm characterized by the facilities for implementing multiple concurrent threads of control in a single process.
Potential Parallelism	Potential for concurrent execution.
Process	A program in execution.
Program Slice	A subset of program statements obtained by *program slicing* on a particular *slicing criterion*.
Program Slicing	A family of techniques involving operations on source code that isolate part of the behavior of a program when viewed from a point of interest with in the program.
Quasi-Static Slicing	A slicing technique that allows partial initial state specification. A quasi-static slicing criterion is a triple $<I, V, p>$, where: V is a set of variables of interest, p is a point of interest in the program, I is a subset of the elements in a complete initial state specification.
Simultaneous Dynamic Slice	Slicing performed with respect to a slicing criterion that specifies a finite set of initial states. A slicing criterion for simultaneous dynamic slicing is a triple $<I, V, p>$, where: V is a set of variables of interest, p is a point of interest in the program, I is a set of initial states.
Slice	See *Program Slice*.
Slicing Criterion	Specification for a particular behavior of interest while slicing. It is expressed as a pair $<V, p>$ for static slicing and a triple $<I, V, p>$ for dynamic slicing, where: V is a set of variables of interest, p is a point of interest in the program,

	I is a set specifying an initial state.
Speculative Multithreading	A run-time parallelization technique that uses control and data dependence speculations in executing threads.
Slicing	See *Program Slicing*
Static Slicing	Slicing performed without considering the input. See also *slicing criterion.*
Supervisor/Worker Approach	A multithreaded solution strategy where the algorithm divides a problem into independent subproblems and delegates each subproblem to a "worker" under its "supervision".
Thread	In this study we look at a thread in its broadest sense, i.e., as parallelly executable sequence of instructions in a program. Depending on context it may be a *thread of computation*, a *thread of a process*, or a *thread in Multithreaded Programming.*
Threadability	Ability of programs to be written using multiple threads.
Thread of Computation	A sequence of statements in the source code of a program that are stepped through while performing a particular computation.
Thread of a Process	A sequence of execution in the context of a process.
Thread in Multithreaded Programming	A unit of abstraction in Multithreaded Programming paradigm.
Thread Library	Library of routines that constitute the operating system support for Multithreaded Programming.
Trajectory	See *Execution Trace.*
Trivial Parallelism	Potential parallelism that can be detected and exploited quite easily.

APPENDIX B

TRADEMARK INFORMATION

Solaris	A registered trademark of Sun Microsystems, Inc.
UNIX	A registered trademark of The Open Group in the United States and other countries.
CodeSurfer	Trademark of GrammaTech, Inc.

APPENDIX C

EXAMPLES OF PROGRAM SLICES

C.1 Example Set I

```
var
   MaxData, Count        : integer;
   Sum, Avg        : real;
   Data, CountNumber     : array[1..10] of integer;
                          /* MaxData = 5 */
                       /* Data = (3,5,5,2,2) */
begin
1  read(MaxData, Data);
2  Count := 1;
3  Sum := 0;
4  while Count <= MaxData do
   begin
      /* count occurrences of number */
5     if Data[Count] = 1 then
6        CountNumber[1] = CountNumber[1] + 1;
7     if Data[Count] = 2 then
8        CountNumber[2] = CountNumber[2] + 1;
9     if Data[Count] = 3 then
10       CountNumber[3] = CountNumber[3] + 1;
11    if Data[Count] = 4 then
12       CountNumber[4] = CountNumber[4] + 1;
13    if Data[Count] = 5 then
14       CountNumber[5] = CountNumber[5] + 1;
      /* computing the sum */
16    Sum := Sum + Data[Count];
17    Count := Count + 1;
   end
   /* Compute average */
18 Avg := Sum / MaxData;
19 write(CountNumber, Sum, Avg);
end.
```

Figure 5. Program to compute number of occurrences and to calculate the sum and average of a set of numbers, the size of the set and its elements are provided as input [Wichaipanitch 92]

```
begin
1   read(MaxData, Data);
2   Count := 1;
4   while Count <= MaxData do
    begin
5       if Data[Count] = 1 then
6           CountNumber[1] = CountNumber[1] + 1;
7       if Data[Count] = 2 then
8           CountNumber[2] = CountNumber[2] + 1;
9       if Data[Count] = 3 then
10          CountNumber[3] = CountNumber[3] + 1;
11      if Data[Count] = 4 then
12          CountNumber[4] = CountNumber[4] + 1;
13      if Data[Count] = 5 then
14          CountNumber[5] = CountNumber[5] + 1;
17      Count := Count + 1;
    end
19  write(CountNumber, Sum, Avg);
end.
```

Figure 6. Static Slice computed based on variable `CountNumber` in line 19 of the program in Figure 5 [Wichaipanitch 92]

```
1(1)        read(MaxData, Data);
2(2)        Count := 1;
3(3)        Sum := 0;
4(4)        Count <= MaxData
9(5)        Data[Count] = 3
10(6)       CountNumber[3] = CountNumber[3] + 1;
16(7)       Sum := Sum + Data[Count];
17(8)       Count := Count + 1;
4(9)        Count <= MaxData
13(10)      Data[Count] = 5
14(11)      CountNumber[5] = CountNumber[5] + 1;
16(12)      Sum := Sum + Data[Count];
17(13)      Count := Count + 1;
4(14)       Count <= MaxData
18(15)      Avg := Sum / MaxData;
19(16)      write(CountNumber, Sum, Avg);
```

Figure 7. Execution trace of the program in Figure 5 on input data `MaxData` = 2, `Data` = (3,5) [Wichaipanitch 92]

```
begin
1   read(MaxData, Data);
2   Count := 1;
4   while Count <= MaxData do
    begin
9       if Data[Count] = 3 then
10          CountNumber[3] = CountNumber[3] + 1;
13      if Data[Count] = 5 then
14          CountNumber[5] = CountNumber[5] + 1;
17      Count := Count + 1;
    end
19  write(CountNumber, Sum, Avg);
end.
```

Figure 8. Dynamic Slice computed for input set {MaxData = 2, Data = (3,5)} based on variable CountNumber in line 19 of the program in Figure 5 [Wichaipanitch 92]

C.2 Example Set II

The following examples have been taken from [Canfora et al. 98]. Figure 9 lists a program that takes inputs `n`, `test0`, and a sequence of `n` integers, computes integers `possum`, `posprod`, negsum, and `negprod`, and outputs the greatest sum and the greatest product.

- `possum` accumulates the sum of positive numbers.
- `negsum` accumulates the sum of absolute values of negative numbers.
- `posprod` accumulates the product of positive numbers.
- `negprod` accumulates the product of absolute values of the negative numbers.

Whenever an input `a` is zero, the greatest sum and the greatest product are reset if the value of `test0` is not zero.

```
1  main() {
2    int a, test0, n, i, posprod, negprod, possum, negsum, sum, prod;
3    scanf("%d", &test0); scanf("%d", &n);
4    i = posprod = negprod = 1;
5    possum = negsum = 0;
6    while (i <= n) {
7        scanf("%d", &a);
8        if (a > 0) {
9           possum += a;
10          posprod *= a; }
11       else if (a < 0) {
12                negsum -= a;
13                negprod *= (-a); }
14            else if (test0) {
15                     if (possum >= negsum)
16                          possum = 0;
17                     else negsum = 0;
18                     if (posprod >= negprod)
19                          posprod = 1;
20                     else negprod = 1; }
21       i++; }
22   if (possum >= negsum)
23         sum = possum;
24   else sum = negsum;
25   if (posprod >= negprod)
26        prod = posprod;
27   else prod = negprod; }
28   printf("%d \n", sum);
29   printf("%d \n", prod); }
```

Figure 9. A program that computes the integers `possum`, `posprod`, `negsum`, and `negprod`, and outputs the greatest sum and the greatest product

```
1  main() {
2    int a, test0, n, i, possum, negsum, sum;
3    scanf("%d", &test0); scanf("%d", &n);
4    i = posprod = negprod = 1;
5    possum = negsum = 0;
6    while (i <= n) {
7        scanf("%d", &a);
8        if (a > 0) {
9           possum += a;
11       else if (a < 0) {
12              negsum -= a;
14           else if (test0) {
15                   if (possum >= negsum)
16                        possum = 0;
17                   else negsum = 0;
21       i++; }
22   if (possum >= negsum)
23        sum = possum;
24   else sum = negsum;
28   printf("%d \n", sum);
```

Figure 10. Static slice of the program in Figure 9 for slicing criterion $<V,p>$, where $V =$ {sum} and $p = 28$

```
1  main() {
2    int a, test0, n, i, possum, negsum, sum;
3    scanf("%d", &test0); scanf("%d", &n);
4    i = posprod = negprod = 1;
5    possum = negsum = 0;
6    while (i <= n) {
7        scanf("%d", &a);
8        if (a > 0) {
9           possum += a;
21       i++; }
22   if (possum >= negsum)
23        sum = possum;
28   printf("%d \n", sum);
```

Figure 11. Conditioned slice of the program in Figure 9 for slicing criterion $<F(V_{in}), V, p>$, where $V = \{\text{sum}\}$, $p = 28$, and $F(V_{in}) = (\forall i, 1 \leq i \leq n, a_i > 0)$

```
1  main() {
2    int a, test0, n, i, possum, negsum, sum;
3    scanf("%d", &test0); scanf("%d", &n);
4    i = posprod = negprod = 1;
5    possum = negsum = 0;
6    while (i <= n) {
7        scanf("%d", &a);
8        if (a > 0) {
9           possum += a;
21       i++; }
22   if (possum >= negsum)
23        sum = possum;
28   printf("%d \n", sum);
```

Figure 12. Dynamic slice of the program in Figure 9 for slicing criterion <*I*, *V*, *p*>, where $I = \{(\text{test0},0), (\text{n},2), (a_1,0)\ (a_2,2)\}$, $V = \{\text{sum}\}$, and $p = 28$

```
1  main() {
2    int a, test0, n, i, possum, negsum, sum;
3    scanf("%d", &test0); scanf("%d", &n);
4    i = posprod = negprod = 1;
5    possum = negsum = 0;
6    while (i <= n) {
7        scanf("%d", &a);
8        if (a > 0) {
9           possum += a;
11       else if (a < 0) {
12                negsum -= a;
21       i++; }
22   if (possum >= negsum)
23        sum = possum;
24   else sum = negsum;
28   printf("%d \n", sum);
```

Figure 13. Quasi-static slice of the program in Figure 9 for slicing criterion <*I*, *V*, *p*>, where $I = \{(\text{test0},0)\}$, $V = \{\text{sum}\}$, and $p = 28$

```
1  main() {
2    int a, test0, n, i, possum, negsum, sum;
3    scanf("%d", &test0); scanf("%d", &n);
4    i = posprod = negprod = 1;
5    possum = negsum = 0;
6    while (i <= n) {
7        scanf("%d", &a);
8        if (a > 0) {
9           possum += a;
11       else if (a < 0) {
14            else if (test0) {
15                    if (possum >= negsum)
16                       possum = 0;
21       i++; }
22   if (possum >= negsum)
23        sum = possum;
28   printf("%d \n", sum);
```

Figure 14. A Simultaneous dynamic slice of the program in Figure 9 for slicing criterion $<I, V, p>$, where $V = \{\text{sum}\}$, $p = 28$, $I = \{I_1, I_2\}$, $I_1 = \{(\text{test0},0), (\text{n},2), (a_1,0), (a_2,2)\}$, $I_2 = \{(\text{test0},1), (\text{n},2), (a_1,0), (a_2,2)\}$

APPENDIX D

AN EXAMPLE ON THREADING SLICES

Figure 15 shows a C language implementation of the program in Figure 5. A multithreaded equivalent of the program is shown in Figures 16a and 16b. The function Read simulates the reading of input to the program. The two programs have identical output for the same input, as shown in Figure 17 for the input data set MaxData = 5, Data = (3,5,5,2,2).

Figure 18 shows the dynamic slice of the program in Figure 5 for the slicing criterion (x, {CountNumber[1]}, 19), where x = (MaxData, Data) = (2, (3, 5)) and is identical to the dynamic slice obtained based on {CountNumber[2]} and {CountNumber[4]} for this particular value of x. The dynamic slice corresponding to slicing criterion (x, {CountNumber[3]}, 19) shown in Figure 19, has structural similarity with that for criterion (x, {CountNumber[5]}, 19) shown in Figure 20. Figures 21 and 22 show the slices for (x, {Sum}, 19) and (x, {Avg}, 19), respectively.

It is obvious that the program selected here is amenable to multithreading. Can it be determined by looking at the slices of the program in Figure 5, as shown in Figures 6, 8, 18, 19, 20, 21, and 22, that it is multithread-able? If yes, can a multi-threaded equivalent of the program be constructed algorithmically from the slices? Would the techniques developed for this program be generic enough to be applied to similar programs? Do slices of other classes of programs show useful patterns that could be exploited for multithreading? Would alternative means of representation of slices in relation to the program - for instance, a modified PDG - help in better understanding and/or in extracting their relationship with threads? These are some of the basic steps in the sequence of fundamental questions that need to be addressed.

```c
#include <stdio.h>

/* variable declartions - global */
int MaxData, Count;
float Sum, Avg;
int Data[10], CountNumber[10];

/* read function definition */
Read( int *maxData, int *data )
{
   data[1] = 3;
   data[2] = 5;
   data[3] = 5;
   data[4] = 2;
   data[5] = 2;
   *maxData = 5;
}

/* write function definition */
Write( int *countNumber, float sum, float avg )
{
   int i;

   for ( i = 1; i <= MaxData; i++ )
      printf( "CountNumber[%d]: %d\n", i, countNumber[i] );

   printf( "Sum: %f\n", sum );
   printf( "Avg: %f\n", avg );
}

/* program body */
main()
{
   Read( &MaxData, Data );
   Count = 1;
   Sum = 0;
   while ( Count <= MaxData )
   {
      if ( Data[Count] == 1 )
         CountNumber[1] = CountNumber[1] + 1;
      if ( Data[Count] == 2 )
         CountNumber[2] = CountNumber[2] + 1;
      if ( Data[Count] == 3 )
         CountNumber[3] = CountNumber[3] + 1;
      if ( Data[Count] == 4 )
         CountNumber[4] = CountNumber[4] + 1;
      if ( Data[Count] == 5 )
         CountNumber[5] = CountNumber[5] + 1;

      Sum = Sum + Data[Count];
      Count = Count + 1;
   }

   Avg = Sum / MaxData;
   Write( CountNumber, Sum, Avg );
}
```

Figure 15. C language equivalent of the program in Figure 5 for input `MaxData` = 5, `Data` = (3,5,5,2,2).

```
#include <thread.h>
#include <stdio.h>
#include <unistd.h>


/* variable declartions - global */
int MaxData, Count;
float Sum, Avg;
int Data[10], CountNumber[10];


/* Count Number of index */
void *CntNumber( void *arg )
{
   int ind;
   static i = 0;

   ind = (int) arg;

   if ( Data[Count] == ind)
      CountNumber[ind] = CountNumber[ind] + 1;

   thr_exit((void *) 0);
}

/* read function definition */
Read( int *maxData, int *data )
{
   data[1] = 3;
   data[2] = 5;
   data[3] = 5;
   data[4] = 2;
   data[5] = 2;
   *maxData = 5;
}

/* write function definition */
Write( int *countNumber, float sum, float avg )
{
   int i;

   for ( i = 1; i <= MaxData; i++ )
     printf( "CountNumber[%d]: %d\n", i, countNumber[i] );

   printf( "Sum: %f\n", sum );
   printf( "Avg: %f\n", avg );
}

/* program body */
main()
{
   thread_t *tidArray;
```

Figure 16a. Multithreaded equivalent of the program in Figure 15 (continued in Figure 16b).

```
    int error = 0;
    int numpaths;
    int index;
    void *status = NULL;

    Read( &MaxData, Data );
    Count = 1;
    Sum = 0;
    while ( Count <= MaxData )
    {
        /* create thread for counting */
        for ( index = 1; index <= 5; index ++ )
        {
            thr_create( NULL, 0, CntNumber, (void *)index, 0,
                        &tidArray[index]);
        }

        /* join all threads */
        for ( index = 1; index <= 5; index ++ )
        {
            status = NULL;
            error = thr_join( tidArray[index], NULL, &status );
            if (! error && status != NULL )
            {
            }
        }

        Sum = Sum + Data[Count];
        Count = Count + 1;
    }

    Avg = Sum / MaxData;
    Write( CountNumber, Sum, Avg );
    thr_exit((void *)0);
}
```

Figure 16b. Multithreaded equivalent of the program in Figure 15 (continued from Figure 16a).

```
CountNumber[1]: 0
CountNumber[2]: 2
CountNumber[3]: 1
CountNumber[4]: 0
CountNumber[5]: 2
Sum: 17.000000
Avg: 3.400000
```

Figure 17. Output of programs in Figure 15 and Figure 16

```
begin
19 Write(CountNumber, Sum, Avg);
end.
```

Figure 18. A dynamic slice based on variable `CountNumber[1]` in line 19 of the program in Figure 5

```
begin
1   read(MaxData, Data);
2   Count := 1;
4   while Count <= MaxData do
    begin
9       if Data[Count] = 3 then
10          CountNumber[3] = CountNumber[3] + 1;
17      Count := Count + 1;
    end
19  write(CountNumber, Sum, Avg);
end
```

Figure 19. A dynamic slice based on variable `CountNumber[3]` in line 19 of the program in Figure 5

```
begin
1   read(MaxData, Data);
2   Count := 1;
4   while Count <= MaxData do
    begin
13      if Data[Count] = 5 then
14          CountNumber[5] = CountNumber[5] + 1;
17      Count := Count + 1;
    end
19  write(CountNumber, Sum, Avg);
end.
```

Figure 20. A dynamic slice based on variable `CountNumber[5]` in line 19 of the program in Figure 5

```
begin
1   read(MaxData, Data);
2   Count := 1;
3   Sum := 0;
4   while Count <= MaxData do
    begin
16      Sum := Sum + Data[Count];
17      Count := count + 1;
    end
19  write(CountNumber, Sum, Avg);
end.
```

Figure 21. A dynamic slice based on variable Sum in line 19 of the program in Figure 5.

```
begin
1   read(MaxData, Data);
2   Count := 1;
3   Sum := 0;
4   while Count <= MaxData do
    begin
16      Sum := Sum + Data[Count];
17      Count := count + 1;
    end
18  Avg := Sum / MaxData;
19  write(CountNumber, Sum, Avg);
end.
```

Figure 22. A dynamic slice based on variable Avg in line 19 of the program in Figure 5.

VITA

Abdunnassar Usman

Candidate for the Degree of

Master of Science

Thesis: INVESTIGATING THE RELATIONSHIP BETWEEN THREADS AND PROGRAM SLICES

Major Field: Computer Science

Biographical:

Personal Data: Born in Chavakkad, Kerala, India, June 21, 1974, son of Theruvath Veliyancode Usman and Zuhara Usman.

Education: Graduated from the Indian High School, Dubai, United Arab Emirates, in May 1991; received Bachelor of Technology in Electronics and Communication Engineering from Government Engineering College, Trichur, under the University of Calicut, Kerala, India, in September 1996; completed the requirements for Master of Science in Computer Science at the Computer Science Department of Oklahoma State University in December 2000.

Experience: Employed by Biosystems and Agricultural Engineering Department, Oklahoma State University, as a Research Assistant / Programming Specialist from May 1997 to December 1999; employed by Computer Science Department, Oklahoma State University, as a Graduate Teaching Assistant from June 2000 to July 2000.