Espresso: A Slicer Generator

Sebastian Danicic
School of Informatics
University of North London
London, N7 8DB
England
Tel: +44 (0)171 973 4833
Fax: +44 (0)171 753 7009

s.danicic@unl.ac.uk

Mark Harman
Dept. of Mathematical & Computing Sciences
Goldsmiths College, University of London
London, SE14 6NW
England
Tel: +44 (0)171 919 7860
Fax: +44 (0)171 919 7853

m.harman@gold.ac.uk

Keywords

Slicing, Software Surgery, Java Threads

ABSTRACT

This paper introduces Espresso, a slicer generator. Espresso compiles the program, p, to be sliced and outputs a slicer. This slicer is a multi–threaded Java program tailored to produce static slices for the program p, (and no other) but with respect to arbitrary slicing criteria. The concurrent nature of the slicers produced by Espresso renders them amenable to parallel execution: Using Java's *Remote Invocation Package*, the programs output by Espresso can be distributed amongst many computing agents thereby speeding up the slicing process. This slicer generator approach has a number of advantages. It facilitates portability and provides efficiency improvement opportunities (via code optimisation and specialisation and via automatic parallelization). The slicers generated by Espresso also produce simultaneous slices and software surgery support information at no additional cost.

1. INTRODUCTION

Program slicing [46] is an automatic program extraction technique, which identifies statements and predicates in a subject program which potentially affect the values of certain variables at some point in the subject program. Informally, a program p is sliced with respect to a slicing criterion which is a pair (V, i), where V is a set of variables and i is a 'point' in the program. As is standard practice, the statements of the program's CFG (Control Flow Graph) will be labelled by 'line numbers' to allow this point to be identified. The slice s of p is obtained from p by deleting statements and has the property that p and s behave identically with respect to the slicing criterion. Tip [44] and Binkley and Gallagher [10] provide detailed surveys of the paradigms, applications and algorithms for program slicing.

To illustrate, consider program p_1 in Figure 1. Slicing[1] p_1 with respect to the set of variables {x} at the end of the program would yield the program p_1'.

```
x=y;
if (x==3)
 {
 c=y;
 x=25;
 }
 i=i+1;
```

```
x=y;
if (x==3)
 {

   x=25;
 }
```

Figure 1: Program p_1 and its slice, p_1'

Slicing has many applications to software engineering problems. The program simplification arising from slicing has been used to assist program comprehension activity [18; 25; 27; 24]. The identification of system elements affected by a system modification has been used to support software surgery [23; 22], and to reduce regression testing effort after such surgery [8; 41]. The ability to extract sub-components of a system according to arbitrarily defined criteria has been used in re-engineering [12; 43; 6]. Slicing has also been applied to

- cohesion measurement [7; 39; 37],
- algorithmic debugging [42; 34],
- component re-use [6; 14],

[1]Throughout this paper, the slices produced are 'static backward slices' [45]. There are, however, many other forms of slices, for example, *dynamic* [35; 3], *conditioned* or *constrained* [11; 21] and *forward* [32].

SAC'00 March 19-21 Como, Italy
(c) 2000 ACM 1-58113-239-5/00/003>...>$5.00

- program integration [31],
- dynamic memory analysis [27],
- array access safety analysis [9]
  and
- software certification [36].

The central problem in slicing is to efficiently compute the dependence relation D, which is the transitive closure of the union of data and control dependence [46; 40; 20]. Data and control dependence are both binary relations defined between the nodes of the CFG of the program being sliced. The slice of p with respect to a slicing criterion (V, i) is simply the set of all nodes of the CFG of p which are related to node i with respect to D.

The most commonly used program slicing algorithms are based on the PDG (Program Dependence Graph) [40], which is merely a representation of the dependence relation D (and its interprocedural counterpart the System Dependence Graph) [33]). This approach consists of a *compilation* phase where the PDG is constructed. PDG construction from a program's CFG is comparatively expensive (approximately cubic in program size) [44]. Before the PDG is constructed it is not known which *actual* slicing criteria the user will wish to apply and therefore using the PDG approach some dependencies between nodes may be computed unnecessarily.

In Weiser's original slicing algorithm [47], the inputs are:

1. A representation of the CFG of the program p being sliced and
2. The slicing criterion (V, i).

Since the slicing criterion is known, only the dependencies relevant to this particular criterion need to be computed. This is approximately quadratic in the size of the program to be sliced [44]. The disadvantage of the approach is that the dependencies are recalculated for *each* new slicing criterion.

Logically, there is no significant difference between, the PDG approach and Weiser's algorithm. Clearly, the more slices of the same program need to be performed, the more attractive becomes the PDG approach.

In this paper a slicer-generator, Espresso, is introduced. Again, 'logically' there is no difference between the approach introduced in this paper and the other approaches described above. The main difference is that given a program p to be sliced, Espresso produces a multi–threaded Java program [28; 38], $S(p)$, which when compiled and run, will produce slices for program p (and p only). Espresso can thus be thought of as a slicer–generator. Since the output of Espresso is a concurrent program[2] it is amenable to parallel execution and to distribution among many hardware agents. In all but pathological cases this parallelisation will lead to significant improvements in efficiency. Also, since the output of Espresso is a program, optimizations can be applied in the production of each slicer.

[2]Espresso is based on the parallel algorithm introduced in [17] which has been proved equivalent to Weiser's algorithm in [15].

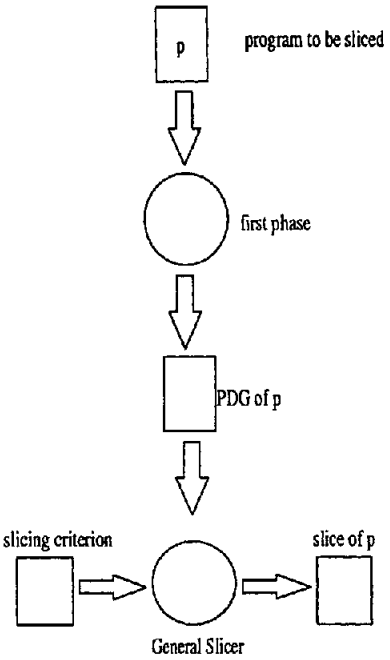

Figure 2: The Current Approach

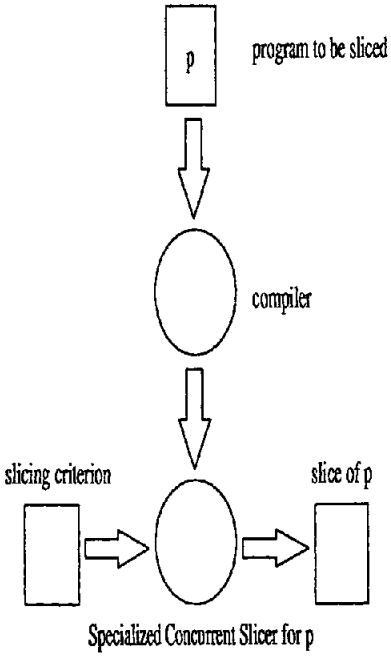

Figure 3: Espresso

```
1   a=0;
2   while (s<t)
3   {if (t==4)
4         c=t;
5     s=2;
6     c=t+7;
7     t=a+4;
    }
```

Figure 4: The program p to be Sliced

$S(p)$, as a by–product, yields additional useful information about p, not available through the PDG approach. In addition to the slice, it yields a set of 'relevant variables' at each node in p's CFG. These relevant variables can be used to assess the impact of software modifications.

The rest of the paper is organized as follows:

Section 2 describes the compilation process and Section 3 presents a worked example, showing the slicer, $S(p)$, that is produced as a result of the compilation process applied to a particular program p and Section 4 illustrates the execution of this slicer with respect to a particular slicing criterion. Section 5 briefly enumerates some of the advantages of the approach. Sections 6 and 7 present conclusions and future work.

2. THE COMPILATION PROCESS

Espresso compiles a subject program p into a Java program $S(p)$. $S(p)$ is a *specialized* slicer for p which waits for inputs of slicing criteria for p and outputs the resulting slice of p.

$S(p)$ consists of a number of communicating threads, each corresponding to a node of p's CFG together with a special *manager* thread, which initiates and terminates the computation of a slice.

Communication between the threads of $S(p)$ is constructed so that the resulting network topology mimics exactly the topology of the CFG, $G(p)$, of p. Communication between the threads in $S(p)$ is, however, in the reverse direction of the edges in $G(p)$. The thread corresponding to node n in $G(p)$ *outputs* messages to all threads that correspond to nodes m with an out–edge from m to n.

Consider, for example, the program given in Figure 4. The topology of the resulting network produced by the compiler is given in Figure 5.

All messages sent and received in this system are of sets containing variable names(strings) and node identifiers(integers). The network structure is realized by the compiler by creating an instance of the generic class `Node` (described in section 2.1), for each node i in $G(p)$. A representation of $G(p)$ is produced. For each node of $G(p)$, the set of defined and referenced variables and the set of nodes it controls are calculated. The current implementation of Espresso assumes that all expressions are side–effect free, so $def(i) = \emptyset$ for all predicate nodes. Espresso also assumes that programs are `goto`–free, so calculating controlled nodes is relatively straight–forward. For a predicate B the set of controlled nodes is simply the set of all nodes corresponding to

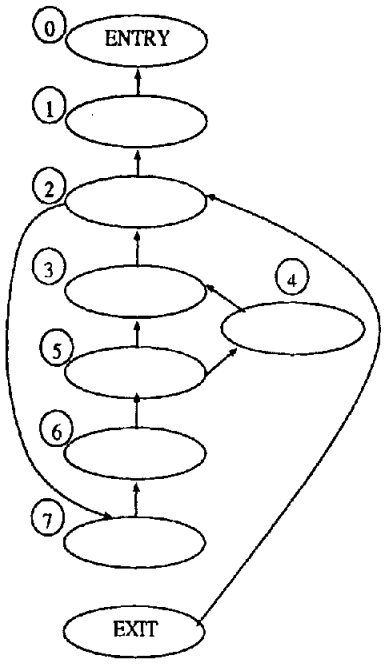

Figure 5: Network Topology of $S(p)$

i	Unique Node ID
ref	The set of variables referenced this node
def	The set of variables defined by this node
C	The set of nodes controlled by this node
$Outs$	The set of all IDs of nodes with out–edges in the CFG to this node

Figure 6: Node Information

atomic statements in the body of B together with the predicate nodes which guard of all non–atomic statements in the body of B. For unstructured languages, the calculation of controlled nodes can be achieved using the algorithm of Ferrante, Ottenstein and Warren [20]. However, slicing unstructured programs[26; 5; 13; 2] is not so straightforward.

For each node i of $G(p)$, an instance of the generic class `Node`, called Node(i) is created by Espresso. Given different programs p and q to be sliced, the only differences in the Java code produced by compiling p and q using Espresso are:

- the number of instances of `Node` (a Java Constant) and
- the values of Node(i) (created by an appropriate call to `Node`'s constructor)

The generic class `Node` is a subclass of the Java `Thread` class, which defines the behaviour of a general node in terms of parameters given in Figure 6. A single instance of a generic subclass `Manager`, also extending `Thread`, is included in the slicer generated by Espresso. It is the `Manager` that is responsible for checking that the system has *stabilized* [17] as well as acting as a simple user interface for the slicer.

F_i :	set(name) $\rightarrow$ set(name);
$F_i(S) =$	if $S \cap (def \cup C) \neq \emptyset$ then $(S - def) \cup ref \cup \{i\}$ else S

Figure 7: The Node Function

2.1 The Generic Node Thread

Apart from the components given in Figure 6, the class `Node` has the following components:

- A local variable, `input`, of type `Set` used to store inputs from neighbouring nodes.
- A method, `void input(Set S)`, which is invoked by threads which output to this node. The result of calling `input(S)` is to update the current value of `input` with the union of the sets: `input` and `S`.
- A method, `output input(Set S, Set outids)` which calls `n.input(S)` for all nodes in the set `outids`. The has the effect of outputting the value `S` to all nodes, n, in the network whose ID is in `outids`.

The `run` method of `Node` repeatedly calls `output(`F`(input),Outs)` to output the value F(`input`) on all its *out* neighbours (`Outs`), where F is the *Node* Function given in Figure 7. The `run` method then reports its output to the *Manager*.
The value of a node's `input` is continually updated *asynchronously* as a result of its *in* neighbours invoking its `input` method.
The *Node* function F of a node n is at the heart of the computation of a slice. Its behaviour is defined as a function on sets of names in Figure 7. more informally, if the current input, S, to the node n, intersects n's its defined variables or its controlled nodes then F yields the set consisting of :-

1. All its input variables (elements of S) that it n not define,
2. All variables that n references,
3. The node ID, i of the node n.

Otherwise (if S has elements in common neither with n's defined variables nor its controlled nodes) then F merely yields the input set S unchanged.
The result of executing the concurrent network produces the functional composition of instances of F_i, for each node i, which result in slices identical to those produced by Weiser's algorithm [46]. This result is formally proved correct in [15].
The F_i are implemented in Java with a method, `Set nodeFunction(Set s)` (Figure 8) in class `Node`. Using functional design patterns[19] to implement polymorphic sets allows `nodeFunction` to be written in a way that closely resembles the mathematical definition given in Figure 7 .

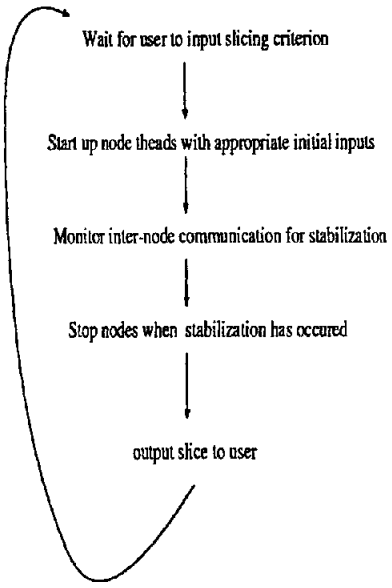

Figure 9: The Manager Cycle

2.2 The Manager Thread

Initially, the manager waits for the user to input a simultaneous slicing criterion, $\{(V_i, n_i)\}$. Having received user input, it then causes each node n_i to output the set V_i by invoking n_i's `output` method as described in Section 2.1. It then starts each node thread. The Manager is informed each time a node outputs a message. If there has been no change on the values of each node's `input` after every node has output a new message, then further activity cannot produce new values [17]. That is, the network has *stabilized*. The manager stops all the threads, outputs a representation of the slice and awaits another slicing criterion. If, on the other hand, there has been some change to the network state, then the manager records the new state and allows communication to continue.

3. EXAMPLE COMPILATION

Let p, the program to be sliced be the one shown in Figure 4.
Espresso outputs, $S(p)$, a Java program consisting of a network of threads whose initial state is shown in Figure 10. in this figure each node depicts three items of state information. The controlled nodes (at the top) and the defined and references variables sets (on the bottom left and right respectively).
In this case, Espresso generated eight instances of the class `Node`, one for each node of $G(p)$ (apart from the exit node).
Each node is generated by a call to the `Node` constructor function. For example, to generate the instance corresponding to Node 2 in Figure 10, the code containing a call to the `Node` constructor shown in Figure 11 was generated.

```
Set nodeFunction(Set s){
if (!s.intersect(def.union(C)).isEmpty())
return s.minus(def).union(ref).union(new ConsSet(new Integer(id),new Empty()));
else return s;}
```

Figure 8: The Node Function in Java

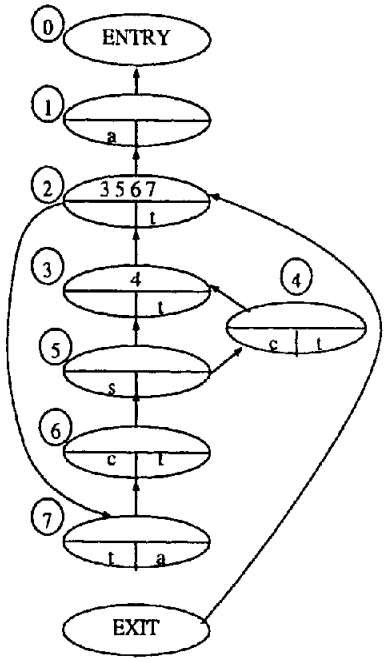

Figure 10: Initial State of $S(p)$

```
processes[2] =
new Node(2, /*id*/

new ConsSet(new Integer(7),
 new ConsSet(new Integer(1),
  new Empty())), /*outs={1,7}*/

new ConsSet(new Integer(3),
 new ConsSet(new Integer(4),
  new ConsSet(new Integer(5),
   new ConsSet(new Integer(6),
    new ConsSet(new Integer(7),
     new Empty())))))), /*C={3,4,5,6,7}*/

new ConsSet("s",
 new ConsSet("t",
  new Empty())), /*ref={"s","t"}*/

new Empty(), /*def=emptyset*/

new Empty() /*input=emptyset*/
);
```

Figure 11: The Code Generated for Node 2

4. EXAMPLE EXECUTION

Suppose a slice of p is to be constructed with respect to the criterion ({c},7) using the Espresso-generated slicer. The manager causes node 7 to output the message {c} to node 6 and then starts the network.

A trace of the execution sequence that occurred is shown below. Espresso-generated slicers are non–deterministic in the sense that different executions may produce different interleavings of events [30]. However, the final output is deterministic, and so Espresso-slicers will always generate the same slices, but the computations involved may differ depending upon the scheduling of execution of node functions. For brevity, only communication which contributes to changes in the value on an edge are shown (whenever the manager thread executes a line is also included in the trace to show this.)

```
Manager.....................
Input to  process 6: {c}
Input to  process 5: {t,6}
Input to  process 4: {t,6}
Input to  process 3: {t,6}
Input to  process 2: {t,6}
Input to  process 7: {6,s,t,2}
Input to  process 1: {6,s,t,2}
Input to  process 6: {c,6,s,2,a,7}
Manager.....................
Input to  process 0: {6,s,t,2}
Input to  process 5: {s,2,a,7,t,6}
Manager.....................
Input to  process 4: {2,a,7,t,6,5}
Input to  process 3: {2,a,7,t,6,5}
Manager.....................
Input to  process 2: {2,a,7,t,6,5}
Manager.....................
Input to  process 7: {a,7,6,5,s,t,2}
Input to  process 1: {a,7,6,5,s,t,2}
Input to  process 6: {c,6,5,s,2,a,7}
Manager.....................
Input to  process 5: {5,s,2,a,7,t,6}
Input to  process 0: {7,6,5,s,t,2,1}
Manager.....................
Manager.....................
```

The manager has noticed that system has stabilized i.e. it has reached a *fixed-point*[16]. Further communication cannot add any new information to the edges. The final state of the network is shown in Figure 12.

From the final state of the network, the slice of the original program is constructed by including those statements and predicates whose node identifiers, $\{1, 2, 5, 6, 7\}$, have reached the ENTRY node (node 0).

5. ADVANTAGES OF ESPRESSO

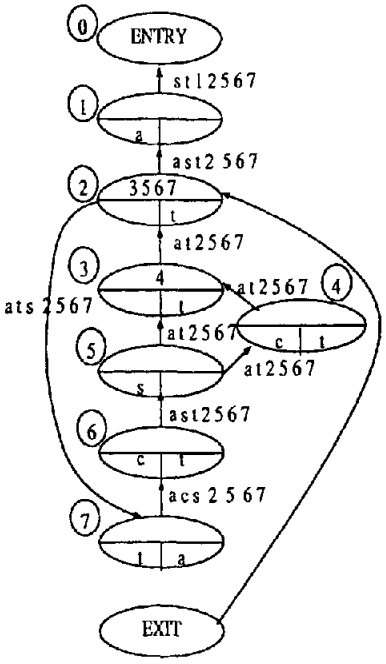

Figure 12: Final State

1	a=0;
2	while (s<t)
	{
5	s=2;
6	c=t+7;
7	t=a+4
	}

Figure 13: The Slice

The Espresso slicer-generator approach has several advantages, some of which derive from the parallel algorithm [17] which its slicers implement, and some of which result from the slicer-generator approach itself. This section briefly describes some of these advantages to motivate the use of Espresso and a slicer generator approach.

5.1 Software Surgery Assistance

A useful by–product of slicing using Espresso is the set of final labellings of each arc. This by–product does not exist in any other approach to program slicing.

During software development and particularly maintenance, it is important to affect a source code change without introducing undesired ripple effects. This has been called software surgery [22; 23]. Typically, regression testing is required after software surgery to test for the presence of ripple effects. Slicing can be used to reduce the regression testing effort required by identifying the subset of tests which could be affected by surgery [8; 41]. However, it would be better to remove the requirement for regression testing altogether. This is the goal of the Surgeon's Assistant [22] project, in which changes are made to the decomposition slice [23], and changes which could affect the complement of this decomposition slice are disallowed.

Using the final edge labelling produced by Espresso, these illegal changes can be identified in a straightforward manner. Suppose S is the slice of p with respect to the slicing criterion C. On each edge e there will be a slicing criterion e_C, which is equivalent to C in the sense that the slice of P with respect to e_C will be a subset of S [17]. This means that changes to e_C will impact C. Using this information, it is possible to assess the ripple effects of changes during software surgery [22]. A modification of any node upon which edge e is incident, must preserve e_C if it is to preserve C.

To illustrate, consider the example in Figure 14. Suppose a change is to be made to the computation of the product, p, which is computed incorrectly. Such a change is software surgery. The surgery should affect the final value of p, but leave the final value of s unaffected. To affect the final value of s, would be to introduce a harmful ripple effect.

The left hand section of Figure 14 shows a node from the program's CFG, while the right hand section shows the slicing criterion which is computed to apply to that node, when the Espresso-generated slicer computes a slice for the final value of s. The sets of variables in these criteria must remain unaffected in order to preserve the final value of s. For example, suppose the maintainer wants to change to node 2, replacing it with p=1; to correct the fault in the computation of the product. This is allowable, because p is not in the set of variables to preserve at node 2. However, replacing this node with one which affects s is not allowed.

5.2 Simultaneous Slicing

Although the algorithm underlying Espresso was originally designed as a parallel version of Weiser's, it has since been noticed that it produces *simultaneous slices*. A simultaneous slice is a generalised version of a slice in the sense that rather than giving a single slicing criterion, a whole set $C = \{(n_i, V_i) | i = 1 \cdots k\}$ of k slicing

Original Program		Variables to Preserve
1	s=0;	s
2	p=0;	s
3	scanf("%d",&n);	s,n
4	while (n>=0)	s,n
5	{ s=s+n;	s,n
6	p=p*n;}	s,n
7	printf("%d",s);	s
8	printf("%d",p) ;	s

Figure 14: Software Surgery: To Preserve s

criteria is given. A simultaneous slice p' of p with respect to C must be such that for all i p' is a slice of p' with respect to (n_i, V_i).
Simultaneous slices can also be computed using the PDG-based approach [33] and using the iterative solution of data flow equations [47], because the slice with respect to criteria $\{C_1, \ldots, C_k\}$ is simply the distributed union of the slices constructed for the individual slicing criteria $C_1, \ldots, C_k$. However, using Espresso-produced slicers, the simultaneous slice can be produced at no extra computational cost.

5.3 Portability

The Java virtual machine is supported on many platforms, with the result that Espresso produces highly portable slicers. For example, a Espresso-produced slicer can be executed in any Java-enabled web browser. In addition, changing the 'back end' code generator is far more easy that changing the 'front end' compiler for the source language. The authors therefore believe that the slicer-generator approach could be used in a manner somewhat similar to a cross-compiler, in which the Espresso's code generator is replaced with different code generators to produce slicers for different platforms.

5.4 Efficiency through Optimization

Since Espresso produces a slicing program rather than a slice of a program, compiler optimisation may be applied to improve the efficiency of a particular slicer constructed. For example, the code for computing the output of a node which does not define a variable can be optimized, because the rule involves calculating the difference of the defined variables with the slicing criterion at that node (see Figure 7). In the case where such a node defines no variables the computation of the output slicing criterion $F_i(S)$ will be

$$S - \emptyset \cup ref \cup \{i\}$$

which the compiler optimisation can reduce using constant folding [4] to give

$$S \cup ref \cup \{i\}$$

This optimisation can be applied to all side-effect free predicate nodes (which will, by definition, have empty defined variable sets). Similarly, for constant assignments, which have no referenced variables, constant folding will optimise the code which computes the output slicing criterion to:

$$S - def \cup \{i\}$$

5.5 Efficiency through Parallelism

Java's *Remote Method Invocation* (RMI) package provides the ability to make remote procedure calls[28; 38]. This means that a thread on one physical machine can invoke a method running on a different physical machine.
The Espresso-produced slicer is highly concurrent since it has a different logical thread for each node in the program's CFG. Using Java's RMI there is the possibility, therefore, to distribute [1] the work performed by these individual nodes among different pieces of hardware.
To improve speed, our multi-threaded approach allows the system to be designed in such a way that the number of processors used to perform slicing can vary with the size and topology of the program being sliced. Large programs will require many processors and small programs few.
Since the granularity is so fine, this distribution can be achieved in a variety of ways, from one extreme where all threads are running on the same machine, to the opposite extreme where each node has its own processor. In most situations the latter approach would be wasteful in communication overhead; it will be more effective to have clusters of nodes that communicate often, running on the same piece of hardware.
The nodes of sub CFGs which form hammock graphs [29] (single entry single exit graphs) tend to communicate more with other nodes of the sub CFG than with those in the surrounding graph. Such sub-CFG may, therefore, form the basis for a suitable decomposition of the overall task of slicing for distributed computation. However, the current implementation of Espresso does not exploit RMI. The authors are currently working on the implementation of this feature, which remains the subject of future work.

6. CONCLUSION

Espresso is a slicer generator, whose input is a program to be sliced and whose output is a specialized multi-threaded slicer for that particular program.
The approach is essentially a slicer-generator approach, in which slices are constructed by a tailor-made slicer. Thus, instead of slicing using a single generic slicer, which takes a program p and a criterion c to produce a slice s, Espresso takes a program p and produces a slicer $S(p)$. $S(p)$ takes a criterion c and produces a slice s.
The slicer generated implements the parallel algorithm for static program slicing, which allows it to exploit the inherent parallelism in the subject program's CFG. This is achieved by Espresso using Java threads.

7. FUTURE WORK

Among the advantages of the approach listed in Section 5, was the possibility of exploiting Java's RMI to

distribute the task of slicing a large program across several machines. This facility is not present in the current implementation of Espresso. More work is required in order to implement analysis of the CFG of the program being sliced in conjunction with the physical configuration of hardware. This analysis will be used by the next version of Espresso to enable it to find a suitable distribution of nodes and subgraphs across processors and machines.

Espresso is essentially a compiler which given a program p to be sliced, produces a Java program of that implements a parallelisation of Weiser's Algorithm. We are also currently working on a similar parallelisation which uses the PDG approach to program slicing.

Espresso has currently been implemented to perform intra–procedural slicing. Some theoretical work on handling program's with procedures has been undertaken [16]. It is currently envisaged that each procedure body will be compiled separately and that and new instances of each procedure thread will be dynamically created as the result of procedure calls. more work is required to implement this and to experiment with the results.

Espresso is a system where a parallel solution to a graph–analytical problem (program slicing) has been produced by a compilation process which creates a network of concurrent processes whose topology is identical to that of the graph being analysed. The authors believe that it is unlikely that program slicing is the only problem where this method will be useful. Further work is required to investigate whether such an approach is suitable for other problems in program dependence in particular and to graph analysis in general.

8. BIOGRAPHIES

Sebastian Danicic has a BSc in Pure Mathematic from Queen Mary College, London University, an MSc in Computation from Oxford University and a PhD in Computer Science from the University of North London. He is currently a senior secturer in computing at the University of North London.

Mark Harman has an MEng. in Computing from Imperial College, London University and a PhD in Computer Science from the University of North London. He is currently a lecturer in computing at Goldsmiths' College, London University.

9. ACKNOWLEDGEMENTS

The work reported here is supported, in part, by Engineering and Physical Science Research Council grants GR/M58719 and GR/M78083

10. REFERENCES

[1] ABRAMSKY, S. Reasoning about concurrent systems. In *Distributed Computing* (London, 1984), pp. 307–319.

[2] AGRAWAL, H. On slicing programs with jump statements. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Orlando, Florida, June 20–24 1994), pp. 302–312. Proceedings in SIGPLAN Notices, 29(6), June 1994.

[3] AGRAWAL, H., AND HORGAN, J. R. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, June 1990), pp. 246–256.

[4] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, techniques and tools*. Addison Wesley, 1986.

[5] BALL, T., AND HORWITZ, S. Slicing programs with arbitrary control-flow. In 1^{st} *Conference on Automated Algorithmic Debugging* (Linköping, Sweden, 1993), P. Fritzson, Ed., Springer, pp. 206–222. Also available as Uniersity of Wisconsin–Madison, technical report (in extended form), TR-1128, December, 1992.

[6] BECK, J., AND EICHMANN, D. Program and interface slicing for reverse engineering. In *IEEE/ACM* 15^{th} *Conference on Software Engineering (ICSE'93)* (1993), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 509–518.

[7] BIEMAN, J. M., AND OTT, L. M. Measuring functional cohesion. *IEEE Transactions on Software Engineering 20*, 8 (Aug. 1994), 644–657.

[8] BINKLEY, D. W. The application of program slicing to regression testing. In *Journal of Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier, 1998, pp. 583–594.

[9] BINKLEY, D. W. Computing amorphous program slices using dependence graphs and a data-flow model. In *ACM Symposium on Applied Computing* (The Menger, San Antonio, Texas, U.S.A., 1999), ACM Press, New York, NY, USA, pp. 519–525.

[10] BINKLEY, D. W., AND GALLAGHER, K. B. Program slicing. In *Advances of Computing, Volume 43*, M. Zelkowitz, Ed. Academic Press, 1996, pp. 1–50.

[11] CANFORA, G., CIMITILE, A., AND DE LUCIA, A. Conditioned program slicing. In *Journal of Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier Science B. V., 1998. to appear.

[12] CANFORA, G., CIMITILE, A., DE LUCIA, A., AND LUCCA, G. A. D. Software salvaging based on conditions. In *International Conference on Software Maintenance (ICSM'96)* (Victoria, Canada, Sept. 1994), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 424–433.

[13] CHOI, J., AND FERRANTE, J. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems 16*, 4 (July 1994), 1097–1113.

[14] CIMITILE, A., DE LUCIA, A., AND MUNRO, M. A specification driven slicing process for identifying reusable functions. *Software maintenance: Research and Practice 8* (1996), 145–178.

[15] DANICIC, S. *Dataflow Minimal Slicing*. PhD thesis, University of North London, School of Informatics, Apr. 1999.

[16] DANICIC, S., AND HARMAN, M. A simultaneous slicing theory and derived program slicer. In 4^{th} *RIMS Workshop in Computing* (Kyoto University, Kyoto, Japan, July 1996).

[17] DANICIC, S., HARMAN, M., AND SIVAGURUNATHAN, Y. A parallel algorithm for static program slicing. *Information Processing Letters 56*, 6 (Dec. 1995), 307–313.

[18] DE LUCIA, A., FASOLINO, A. R., AND MUNRO, M. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension* (Berlin, Germany, Mar. 1996), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 9–18.

[19] FELLEISEN, M., AND FRIEDMAN, D. P. *A Little Java, A Few Patterns*. The MIT Press, 1998.

[20] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems 9*, 3 (July 1987), 319–349.

[21] FIELD, J., RAMALINGAM, G., AND TIP, F. Parametric program slicing. In 22^{nd} *ACM Symposium on Principles of Programming Languages* (San Francisco, CA, 1995), pp. 379–392.

[22] GALLAGHER, K. B. Evaluating the surgeon's assistant: Results of a pilot study. In *Proceedings of the International Conference on Software Maintenance 1992* (Nov. 1992), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 236–244.

[23] GALLAGHER, K. B., AND LYLE, J. R. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering 17*, 8 (Aug. 1991), 751–761.

[24] HARMAN, M., AND DANICIC, S. Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability 5*, 3 (Sept. 1995), 143–162.

[25] HARMAN, M., AND DANICIC, S. Amorphous program slicing. In 5^{th} *IEEE Internation Workshop on Program Comprehesion (IWPC'97)* (Dearborn, Michigan, USA, May 1997), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 70–79.

[26] HARMAN, M., AND DANICIC, S. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance 10*, 6 (1998), 415–441.

[27] HARMAN, M., SIVAGURUNATHAN, Y., AND DANICIC, S. Analysis of dynamic memory access using amorphous slicing. In *IEEE International Conference on Software Maintenance (ICSM'98)* (Bethesda, Maryland, USA, Nov. 1998), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 336–345.

[28] HARTLEY, S. J. *Concurrent Programming – The Java Programming Language*. Oxford University Press, 1998.

[29] HECHT, M. S. *Flow Analysis of Computer Programs*. Elsevier, 1977.

[30] HOARE, C. A. R. *Communicating Sequential Processes*. Prentice–Hall, 1985.

[31] HORWITZ, S., PRINS, J., AND REPS, T. Integrating non–interfering versions of programs. *ACM Transactions on Programming Languages and Systems 11*, 3 (July 1989), 345–387.

[32] HORWITZ, S., REPS, T., AND BINKLEY, D. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988), pp. 25–46. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[33] HORWITZ, S., REPS, T., AND BINKLEY, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12*, 1 (1990), 26–61.

[34] KAMKAR, M. *Interprocedural dynamic slicing with applications to debugging and testing*. PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1993. Available as Linköping Studies in Science and Technology, Dissertations, Number 297.

[35] KOREL, B., AND LASKI, J. Dynamic program slicing. *Information Processing Letters 29*, 3 (Oct. 1988), 155–163.

[36] KRINKE, J., AND SNELTING, G. Validation of measurement software as an application of slicing and constraint solving. In *Journal of Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier, 1998. to appear.

[37] LAKHOTIA, A. Rule–based approach to computing module cohesion. In *Proceedings of the* 15^{th} *Conference on Software Engineering (ICSE-15)* (1993), pp. 34–44.

[38] LEA, D. *Concurrent Programming in Java: Design Principles and Patterns*. Addison Wesley Longman Inc., 1997.

[39] OTT, L. M., AND THUSS, J. J. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium* (Baltimore, Maryland, USA, May 1993), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 71–81.

[40] OTTENSTEIN, K. J., AND OTTENSTEIN, L. M. The program dependence graph in software development environments. *SIGPLAN Notices 19*, 5 (1984), 177–184.

[41] ROTHERMEL, G., AND HARROLD, M. J. Selecting tests and identifying test coverage requirements for modified software. In *ACM International Symposium on Software Testing and Analysis* (Aug. 1994), pp. 169–184.

[42] SHAHMEHRI, N. *Generalized algorithmic debugging*. PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1991. Available as Linköping Studies in Science and Technology, Dissertations, Number 260.

[43] SIMPSON, D., VALENTINE, S. H., MITCHELL, R., LIU, L., AND ELLIS, R. Recoup – Maintaining Fortran. *ACM Fortran forum 12*, 3 (Sept. 1993), 26–32.

[44] TIP, F. A survey of program slicing techniques. *Journal of Programming Languages 3*, 3 (Sept. 1995), 121–189.

[45] VENKATESH, G. A. The semantic approach to program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Canada, June 1991), pp. 26–28. Proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

[46] WEISER, M. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[47] WEISER, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.