# CPU Trace Recompilation

Sam Epstein*

January 14, 2025

**Abstract**

We introduce a new type of algorithm with $O(n \log n)$ time complexity that recompiles long sequences $C$ of assembly instruction traces of length $n$ into a novel code called SECode (short for side effect code). When $C$ is encountered again, SECode is run, producing the same side effects as $C$. SECode also matches and reproduces the side effects of instruction traces that have registers pointing to different memory locations than $C$. The only two data structures in SECode is a graph for matching and a table for the changes. During operation, a match produces a single array which is applied to the table for the side effects. If $C$ uses $r$ and $s$ non-transient heap read and write operations and $t$ non-transient stack operations, then the size of SECode is $O(r+s+t)$, and it runs in time $O(r \log r + s + t)$, with small constants. No code annotations are needed; it works on all compiled code.

## 1 Introduction

CPU designers have focused away from increasing the clock speed because the processors cannot be cooled fast enough. More heat is created from faster clock speed and a point has been reached where the amount of energy it takes to cool increasingly fast processors is prohibitively expensive. CPU manufacturers have turned to parallelism for clock speedups. For example, Intel has released the *Intel Parallel Studio*, which is a software development project to facilitate parallel programming to take advantage of multi-core processors. However, this complicates the coding process, eliminating coding abstractions that isolate the programmer from the underlying details of the computer running the code.

This obstacle invites new approaches to increasing the speed of CPU's. With the Programmable Smart Machine Lab (PSML)[AWE+14, WAS12, App14], a new model is proposed. It is noted that traces of CPUs are very non-random. This implies there's redundancy in the trace streams, which implies there exists so-called hot paths, which are long pieces of code with minimal side-effects that are repeatedly executed by the computer. PSML proposes a form of *Memoization* of the hot paths, originally proposed by [Mic68]:

*samepst@jptheorygroup.com

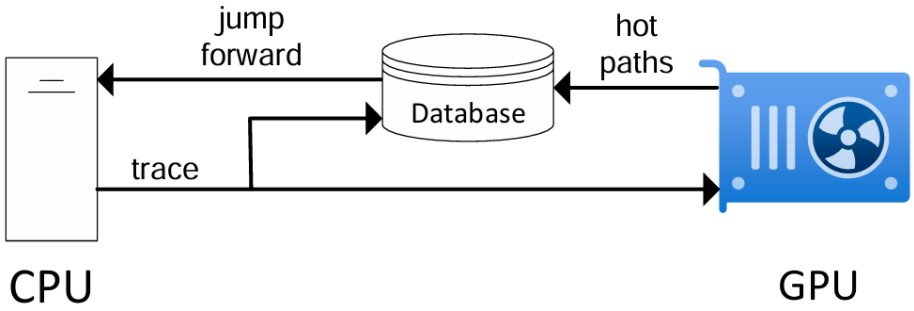



Figure 1: The proposed setup for exploiting hot paths.

> "It would be useful if computers could learn from experience and thus automatically improve the efficiency of their own programs during execution... When I write a clumsy program for a contemporary computer a thousand runs on the machine do not re-educate my handiwork. On every execution, each time-wasting blemish and crudity, each needless test and redundant evaluation, is meticulously reproduced."

Mitchie further noted that functions can be *memoized* and either calculated directly (rote) or through a lookup table (rule).

> "On each given occasion proceed either by rule, or by rote, or by a blend of the two, solely as dictated by the expediency of the moment ... rule versus rote decisions shall be handled by the machine behind the scenes."

With memoization in mind, the proposed setup (with some minor changes) can be seen in Figure 1. The CPU sends its operating trace to a GPU, which uses machine learning to identify hot paths. The hot paths are sent to a database. This database, upon receiving the same trace as the GPU, will periodically initiate a jump forward in the state of the computer. To the author's knowledge, PSML was the first team to propose this configuration.

This paper introduces a solution to the database component in Figure 1. This database is called the SEDatabase, short for side effect database. The machine learning portion is out of the scope of this paper.

A proposed description of the database was discussed in [AWE+14]. The state of the computer was represented as a $1 \times n$ binary picture which evolves over time, as seen in Figure 2 (seen as an $n \times n$ picture). They described the evolution of the computer as a very long video. Partial states of the image are queried into the database and if a matching rule is found, the state of the computer is updated with a new partial picture.

However, take the following example, where the assembler instruction trace starts with a register pointing to a linked list in memory. Every node has a number. The instruction trace sorts the list and returns the head of the list. The

Figure 2: The evolution of the state of the computer, represented as a binary image.

above proposed method in Figure 2 cannot be applied to this example because different runs will have registers pointing to linked lists in different memory locations. This paper proposes a method to overcome this difficulty. If the linked list is of the form $(3, 7, 4, 5, 1)$ then whenever the SEDatabase encounters the starting point of compiled assembler trace, it will return $(1, 3, 4, 5, 7)$ regardless of the memory locations. The SEDatabase also handles memory creation. If the source trace constructs a balanced binary tree in memory on input $(15, 9, 10, 3, 6, 7)$ then the next time those values are encountered, the SEDatabase will directly construct the resultant tree (without any balancing operations), populate it with the correct values and then terminate.

There are two phases in the overall setup, the SECode construction phase and the execution phase. The construction process starts when the machine learning component selects a region of trace assembler code to be compiled. Once the trace code is identified, it is recompiled into SECode. To do this, the original code is compiled into a annotated directed graph. This graph is represented visually with SEDiagrams. We will use the term graph and SEDiagrams interchangeably. SEDiagrams can be reduced to a minimal set of conditions and side effects of the original code. Then the SEDiagrams is recompiled into SECode, in particular the SEConditions and the SEChanges. The SEConditions part is a single graph used for testing whether the system is match. The SEChanges is a single table whose members have two entries. The first entry is the location to write (memory, register, or stack) and the second entry is the value to write (number or memory location). The entire process of creating the SECode can be completed in $O(n \log n)$ time. This configuration can be found in Figure 3.

The execution of the SEDatabase is as follows. Say the computation uses $r$ and $s$ non-transient read and write heap operations and $t$ non-transient stack operations. For each compiled SECode the SEConditions checks whether there is a match in $O(r \log r + s + t)$ time. If, during the course of operation, a SECondition is satisfied, then the output is an array of pointers of size $\leq r$. This array is applied the the table that is SEChanges. The side effects of the compiled code is then computed in $O(r+s+t)$ time. After that, normal operation resumes. The tight bounds are achieved because the following components of the original instruction trace are removed.

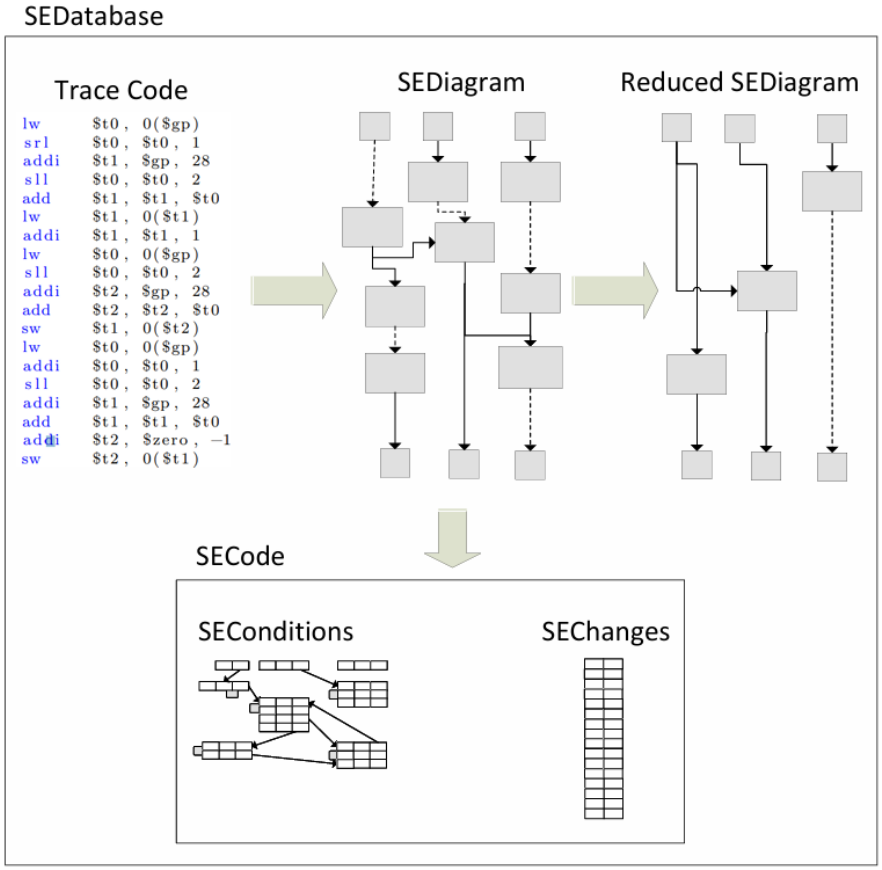



Figure 3: An overview of the construction of SECode in an SEDatabase. The original code is compiled into an SEDiagram which is reduced and recompiled into SECode. SECode consists of SEConditions for the matching, which is a graph, and SEChanges for the side effects, which is a table.

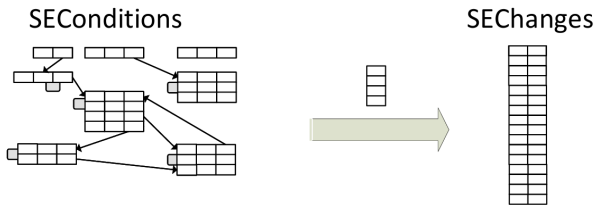



Figure 4: The execution of the SECode. If a match occurs then an array of memory locations is produced. This is sent to SEChanges which is comprised of a single table where each entry has two members.

- Redundant memory operations.
- Math operations.
- Transient register operations.
- Transient memory operations.
- Transient stack operations.

The SECode is able to be memory agnostic because it uses the following key property of pointers[1] to identify which registers contain raw data and which contain memory locations:

> *Pointers are only dereferenced, added to, subtracted from, and compared with other pointers or zero.*

## 2 Related Work

In [GTM99], sequences of assembler instructions are jumped over using a lookup table. Memoization has been implemented on functional languages, [ABH11]. A program to dynamically identify areas of code that represent good candidates for memoization was introduced in [TPG15]. In [ACV05], lossy memoization for multimedia floating-point applications was proposed.

There is a lot of work on combining pattern recognition and computing when the results only need to be approximately correct [Mit16]. Machine learning has been incorporated at the operating system level with regard to learning configurations, learning policies, learning mechanism, and process scheduling [KK20]. Machine learning has also been suggested as a tool for managing the configuration of operating systems [ZH19]. A machine learning library for kernel space has been developed [UAZ20]. However, to the author's knowledge, there is nothing in the literature (aside from PSML) which suggests applying machine learning directly to CPU traces with the intention of leveraging hot paths.

## 3 MIPs Instruction Set

In this paper, a subset of MIPs instruction set is used for the traces. With some redundancy, the entire instruction set can be used. The majority of this paper assumes the operations are over integers. We assume the reader is familiar with properties of the heap, registers, and the stack. All registers are denoted by \$n, for some number $n$. The \$0 register is always set to 0. The stack pointer register is represented by \$sp. We assume a small, but unspecified, number of registers. The following MIPs operations will be used in this paper.

[1] This paper doesn't deal with the case of testing the difference between two pointers, but this can be handled readily.

### Load Immediate

```
li  $1, 300
```

The load immediate operation puts the constant second argument into the register specified in the first argument.

### Move

```
move  $1, $2
```

The move operations puts the contents of register $2 into register $1.

### Add Immediate

```
addi $1, $2, 200
```

The contents of register $1 is set to the contents of register $2 plus the absolute number (in this case 200).

### Add

```
add $1, $2, $3
```

The contents of register $1 is set to the contents of register $2 plus register $3.

### Subtract

```
sub $1, $2, $3
```

The contents of register $1 is set to the contents of register $2 minus register $3.

### Multiply

```
mul $1, $2, $3
```

The contents of register $1 is set to the contents of register $2 times register $3. The way SEDatabase handles MUL is the same way it handles all mathematical operations other than addition and subtraction. This includes OR, AND, DIV, an SHIFT. Therefore, we exclude these operations from the scope of the paper.

### Branch on (Not)Equal

```
beq $1, $2, 100
```

If register $1 is equal to register $2, then goto the program counter + 4 + offset (in this case, 100). In addition, we will also use branch on not equal, BNE, which is of the same format as BEQ.

```
bne $1, $2, 100
```

### Branch on Less Than

```
blt $1, $2, 100
```

If register $1 is less than register $2, then goto the program counter + 4 + offset (in this case, 100). SEDatabase handles branch on less than as it handles branch on less than or equal, greater than, and greater than or equal, so these operations are not included in this paper.

### Load Word

```
lw $1, 100($2)
```

Register $1 is set to the contents of the memory at the location specified by register $2 plus the offset (in this case 100).

### Store Word

```
sw $1, 100($2)
```

The contents of register $1 is saved into memory at location in register $2 plus the offset (in this case 100).

### New

To allocate memory, MIPs specifies placing the memory size a register and performing a syscall and the address location is specified in a destination register. However, for simplification purposes, we will use the following notion to allocate memory.

```
new $1, $2
```

This operation creates memory of size equal to the contents of register $2 and places its location in register $1.

### Free

The MIPs instruction set does not include a method for freeing memory. We create a new operation of the following form.

```
free  $1
```

This operation frees the memory location at the address in register $1.

### Jumps

MIPs has operations that allow the control flow jump to a new location. The j jumps to the absolute location specified in the argument. The jr operation jumps to the address specified in the register argument. The jal operation jumps to a set address and puts the next address in the $ra register.

```
j    1000
jr   $1
jal  1000
```

### Trace Code

When the assembler code is run, the registers will have values associated with them. In order to represent them, numbers are point in the comments section.

```
add  $1 $2 $3 # 5 4 1
```

This indicates register $1 results in a value of 5 and registers $2 and $3 are 4 and 1, respectively. Depending on the operation, the first argument equals either the current register value or a new assigned value. An example is the LW and SW operations.

```
lw $1, 100($2) # 200 14352
sw $1, 400($2) # 300 45902
```

The LW operation indicates register $1 results in a value of 200 and $2 contains the address 14352+200 that was referenced. The SW indicates 300 is in register $1, which is stored in address 45902+400.

## 4 SEDiagram Construction

SEDiagrams are linearly constructed from large swaths of assembler trace code. SEDiagrams consists of two types of constructs: nodes and lines. Nodes are represented by gray rectangles and the lines have arrows at the end. The diagram is a directed acyclic graph that flows downward. An abstract SEDiagram can be seen in Figure 5. For each operation in the trace code, zero, one or two nodes are created at the bottom of the SEDiagram. New lines going into the node are constructed, connecting to previous created nodes higher up in the SEDiagram. After the SEDiagram has been constructed, it can be reduced and then used to create the SEConditions and SEChanges. The SEConditions construct is an annotated directed graph representing pointers and constant values. SEChanges is a table where each entry has two parts: the source and the destination. The construction of the SEConditions and SEChanges from the SEDiagram occurs with algorithms starting at the top of the diagram and then working its way down.

If, during the course of the operation of the computer, there is a match with a SEConditions construct, then the current state of the computer matches the state of the computer when the original trace code was compiled. Thus, there is a match between values accessed during the running of the trace and a bijection between pointers of interest. This means that the control flow of the two states are the same. Due to this match, math and branching operations do not need to be recomputed.

The SECode distinguishes between two different types of data: pointers and numbers. The key difference between these two datatypes is that pointers can only perform one of the following operations:

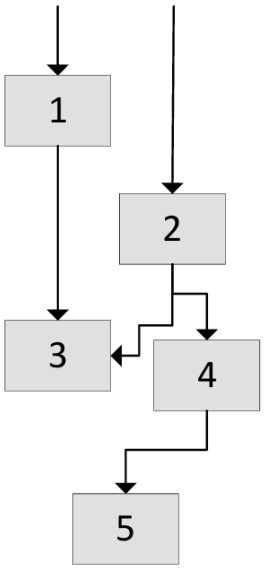



Figure 5: An abstract SEDiagram. The nodes of the SEDiagram are compiled into SECode from top to bottom, in the order specified.

- Dereferenced.
- Added by a number.
- Subtracted by a number.
- Compared to NULL.
- Compared to another pointer.

Using this criteria, the construction of the SEDiagram determines which information is a regular number or a pointer. The set of node types of SEDiagrams can be seen in Figure 6.

## 4.1 Lines

Lines contain three pieces of information: a val, a register number, and a type. There are three different types of lines: *Const*, *Var*, and *SP*. The *Const* lines represent values in the computation which are not pointer values. The val of a *Const* line is the value of the constant. For example, if the trace code is the sorting of a linked list, then the *Const* lines represent the integer values accessed and modified in the linked list nodes. The *Var* lines represent the pointer values used in the original trace code. The val of a *Var* line is the raw pointer value. The *Var* lines have the restriction that the only operations that can be applied

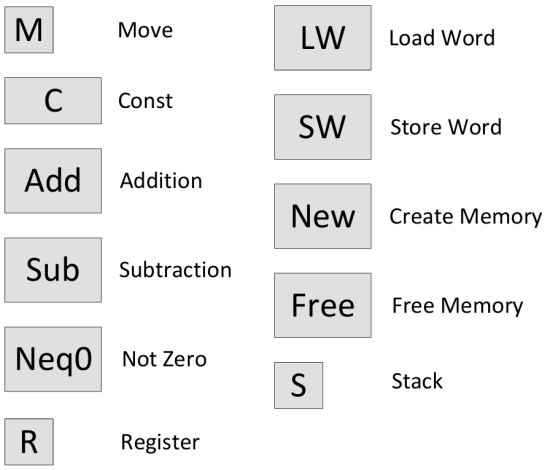



Figure 6: The eleven different node types of the SEDiagram.

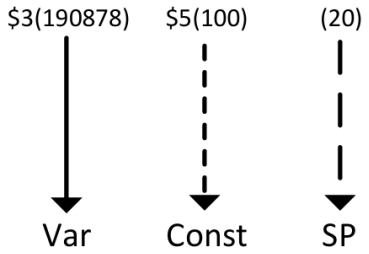



Figure 7: The three different type of lines. The *Var* line is on register $3 with val 198078. The *Const* line is on register $5 with val 100. The *SP* line has offset equal to 20.

to them is deferencing, addition or subtraction by a constant, or testing equality with other *Var* lines or NULL. The *SP* lines, short for Stack Pointer, contains the value of the $sp register. Thus the register of the *SP* is always $sp. The *SP* line always starts with a value of 0, representing an offset from the original $sp value. Thus if the *SP* line contains val of 100, then this means the stack pointer is 100 more than its original value. Other than the *SP* line, all lines are initiated with the *Var* type.

A line is active if there is no line further down with the same register. When a operation occurs, it may create a node. Some of the arguments of this operation are registers. If there are active lines with these registers, then they are connected to the input of these nodes. Otherwise, a new line is created with an indication that its starting point is the register at the starting point of the trace code. The end state of registers can be simply calculated from the active set of lines at the bottom of the SEDiagram.

Figure 7 shows how the lines are displayed graphically, with their information shown on top. The register number and/or the value can be omitted to not overly clutter the diagram. An example use of the R and S nodes can be seen in Figure 8.

## 4.2 Registers and Stack

The SEDiagram contains information about the starting and ending register values and the starting and ending state of the stack. Register and stack information is represented in the SEDiagram by the R and S nodes, respectively. At the top of the diagram, *Const* and *Var* lines come out of the R nodes to represent the starting register values. The R nodes contain the register value. Lines also come out of the S nodes, which contain an integer representing the values location in terms of the offset from the stack pointer register's starting value.

At the bottom of the SEDiagram, R and S nodes represent the end state of the registers and stack. Lines go into these bottom nodes. If they are of type *Var*, then there is an additional integer used in the construction of the SEChanges code specifying how the pointer variable changed. This is represented on the left of the R node.

In addition the SEDiagram contains an integer SPChange representing the amount that the stack pointer has changed. An example use of R and S nodes can be seen in Figure 8.

## 4.3 Jumps

Since the instruction trace always handle the same logical path, the j, jr, and jal jump operations can be ignored when they are encountered in source intrsuction trace during the creation of the SEDiagram.

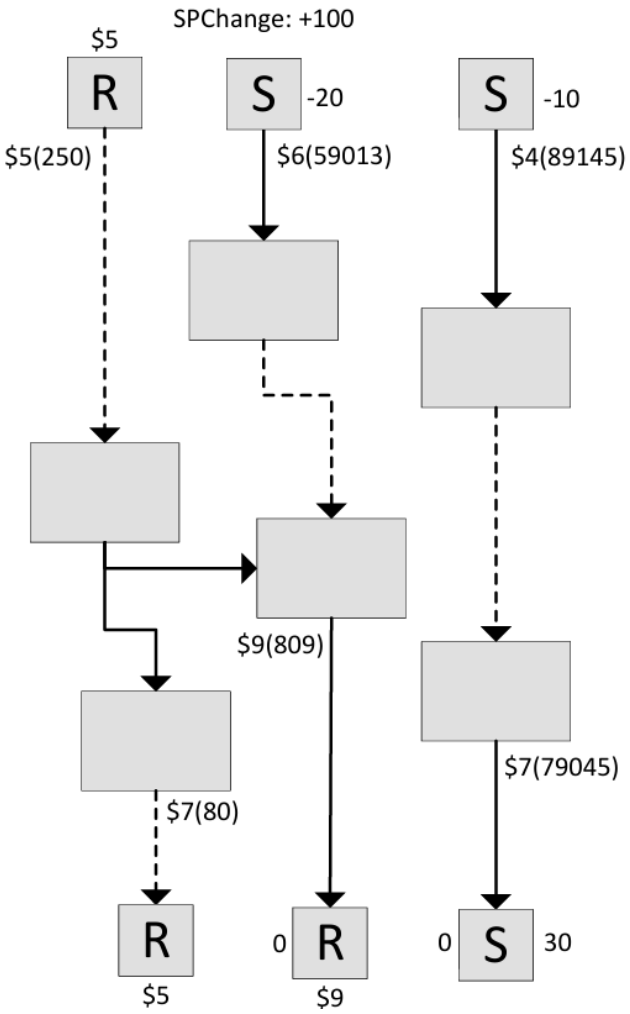
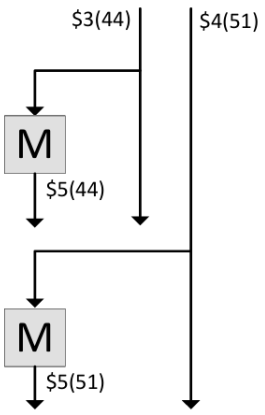



Figure 8: The R and S nodes at the top and bottom represent the starting and ending states of a subset of the registers and stack, respectively. The S nodes have offset information which is on their right. The nodes at the bottom with incoming *Var* links have an additional integer offset information on the left, which always starts at zero. The SPChange value indicates that the stack pointer has increased by 100.




```
1 move   $5 , $3 # 44 44
2 move   $5 , $4 # 51 51
```

Figure 9: The move operation is represented by the M node. The value 44 is passed from register $3 to register $5. Then this line becomes inactive when 51 is passed from register $4 into register $5.

## 4.4 Move

The move operation transports the contents of one register into another register. It corresponds to the M node in the SEDIAGRAM, which is shown in Figure 9.

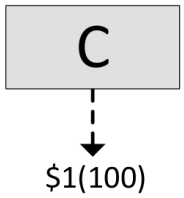
## 4.5 Const

The input to a Const node is zero, one, or two lines and the output is zero or one line. When a Const node is created, if an input line is of type *Var* then it will be changed to type *Const*. An example of a Const node with zero input lines can be seen with the LI command, which loads a constant into a register. Its corresponding SEDIAGRAM can be found in Figure 10.

The multiply operator MUL (as well as the OR, AND, DIV, an SHIFT operations) will create a Const node. This is because we assume that the computation does not perform these operations on pointers. An example of multiple Const nodes in tandem can be found in Figure 11.

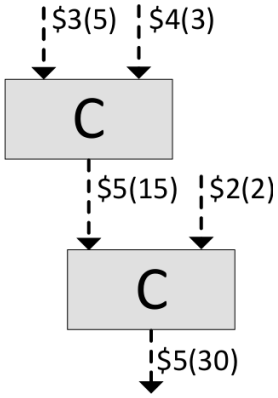
## 4.6 Add and Subtract

The ADD and SUB instructions will create Add and Sub nodes, respectively. Both nodes take two inputs and one output. They can be seen in Figure 12.



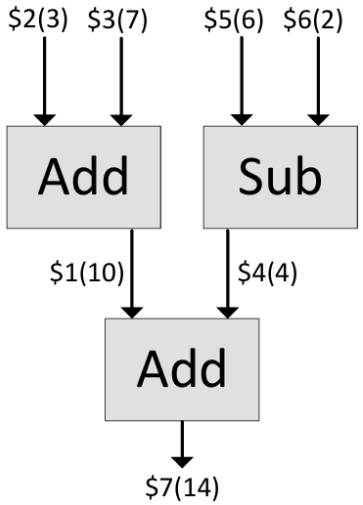

```
li $1, 100 # 100
```

Figure 10: The SEDiagram constructed from one LI operation.



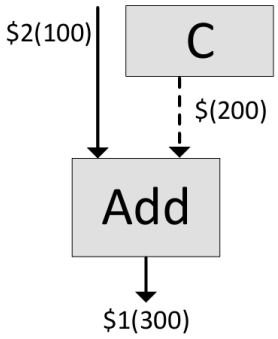

```
mul $5, $3, $4 # 15 5  3
mul $5, $5, $2 # 30 15 2
```

Figure 11: The SEDiagram constructed from two MUL operations. Note that the diagram would be the same f the DIV, AND, or OR operations were used instead. Note that that, to remove clutter, the active lines are not extended to the bottom of the diagram, though technically they should be.




```
add $1, $2, $3 # 10 3  7
sub $4, $5, $6 # 4  6  2
add $7, $1, $4 # 14 10 4
```

Figure 12: The SEDiagram constructed from two ADD operations and one SUB operation.




```
addi $1, $2, 200 # 300 100
```

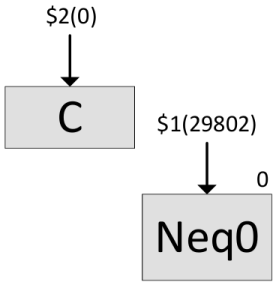
Figure 13: The SEDiagram constructed from an ADDI operation. There are two nodes created. The first node is of type Const, and outputs the constant parameter of the ADDI command. The second node is of type Add, and it sums the constant with the first register parameter. The const line that is outputted from the Const node has no register associated with it.

## 4.7 Add Immediate

The ADDI operation adds a constant number to a register. The corresponding SEDiagram produces two nodes, a Const node and an Add node. The Const node has no input and produces one output, which is the constant to be added to. The output of the Const node is sent to the second input of the Add node. This line does not have a register. An example can be seen in Figure 13.

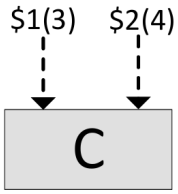
## 4.8 Branch on (Not)Equal

The BEQ command branches if the two register arguments are equal. Similarly, the BNE command branches if the two register arguments are not equal. If the corresponding lines of the two arguments are of type *Const*, then the branch will always occur or not occur, because all computer states that match the SEConditions will have the same const values. Thus the current computation will take the same branch choice as the original compiled trace code. Similarly, if the arguments are two pointers, as we will see later in the paper, all computer states that match the SECondion will result in an identical branch outcome. However, pointers can be tested to see if their values are 0. For a BEQ or BNE command with a $0 argument, if the other register argument has value 0, then a Const node is created. Otherwise a Neq0 node is created. This node specifies that the incoming *Var* line is not equal to 0. However, the val of the line unspecified. The Neq0 node also contains an input offset, to be used in the construction of the SEChanges code. The integer offset is on the upper right. This can be seen in Figure 14.




```
1 bne $0, $2, 200 # 0      0
2 beq $1, $0, 200 # 29802  0
```

Figure 14: An example SEDIAGRAM produced from a BNE and a BEQ operation. Since $2 is equal to 0, a Const node is created. Furthermore, since $1 is a variable not equal to 0, a Neq0 node is created. The integer offset, to be used in the diagram reduction, is on the upper right. It is always initialized to 0.




```
1 blt $1, $2, 100 # 3 4
```

Figure 15: The BLT operation produces a Const node with two input *Const* lines. There is no output lines.

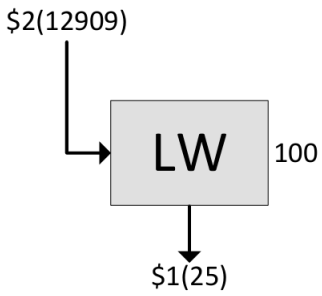
## 4.9 Branch on Less Than

We assume that two pointers are not compared to each other by the less than operation (or the other inequalities). Thus only numbers are compared to. The BLT operation creates a Const node with two *Const* input lines and no output lines. If the input line is initially of type *Var*, then it is changed to type *Const*. An example of the BLT command can be found in Figure 15

## 4.10 Load Word

The LW operation loads the contents of the memory with the address of the second register parameter into the first parameter. The address is offset by a constant number. The address line cannot be of type *Const* and if this happens,

the trace code is not compatible with the SEDatabase. This operation creates a single LW node as shown in Figure 16.




```
1 lw $1, 100($2) # 25 12909
```

Figure 16: The LW operation produces a LW node with on input *Const* line and one output line. The address is on the left side of the node and the data that was loaded is outputted at the bottom. The address offset is on the right.

## 4.11 Store Word

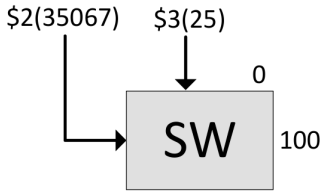
The SW operation stores data in the memory at the address specified by the second register argument. The address is offseted by a constant. The SW operation results in a single SW node. The node takes in an address on its left side and information from the top. In addition, the store word has an additional parameter, the input offset, which is to be used when creating the SEChanges code. It is initially set to 0, and can change when the SEDiagram is reduced. Note that LW and SW nodes with the same address are aligned onto the same vertical column.

## 4.12 New and Free

The NEW operation takes in an argument specifying the amount of memory to allocate and returns a pointer to the memory allocated. The FREE operation takes in a single argument specifying a pointer to memory to be freed. The New and Free nodes of the SEDiagram are created. An example can be seen in Figure 18.

## 4.13 Stack Pointer

Operations with the stack pointer register, $sp, produce nodes in SEDiagram in the same way as if a regular register is used. For simplicity, we will not explicitly deal with the frame pointer. An example of the stack pointer register in use can be seen in Figure 19




```
1 sw $3, 200($2) # 25 35067
```

Figure 17: The SW operation produces a SW node with two inputs and no outputs. The address is fed into the node on the left. The data to be stored reaches the node from the top. The address offset is on the left. The input offset, which is always initialized to 0, is on the upper right.

### 4.14 Implementation

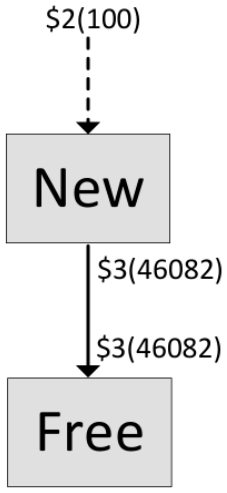
The construction of the unreduced SEDiagram can be completed in $O(n)$ time. The key datastructure is a lookup table that maps registers to the active line. When a new node is created, its inputs are connected to active lines specified by the table. If a line does not exist, then a new one is created and an R node is created. The end register states are simply the set of active lines when the SEDiagram is completed.

Note that technically, in the implementation, the C nodes don't need to be explicitly created. This is because in Section 5.5, all superfluous C nodes are removed.

## 5 SEDiagram Reduction

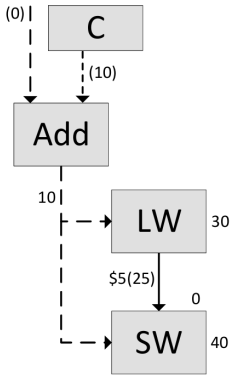
Once the SEDiagram has been constructed, it is then reduced. This reduced SEDiagram differs from the original one in a couple of ways. Firstly, the *Var* lines don't have register values associated with them. Secondly, all *Const* lines are connected to a C node with one outgoing line. Thirdly, the integer offsets in the SW, R, S, and Neq0 nodes are utilized. SEDiagram reduction consists of the followig steps.

1. Register Removal
2. Transient Stack Memory Removal
3. Const Reconciliation
4. Move Node Removal
5. Const Line Stubbing
6. Add and Sub Node Removal




```
1 new  $3, $2 # 46082 100
2 free $3     # 46082
```

Figure 18: Following code creates and frees memory. The NEW operation creates a New node, which takes in a *Const* line argument specifying the amount of memory to allocate. The output of the New node is sent to the input of the Free node, which is created from the FREE operation.




```
1 addi $sp $sp 10  # 0
2 lw   $5, 30($sp) # 25 10
3 sw   $5, 40($sp) # 25 10
```

Figure 19: The stack pointer starts with an initial value of 0 (regardless of its actual value). The stack pointer is moved up by 10, reads a value into register $5 at offset 40 and then stores it in the stack at offset 50.

7. Var Datum Removal
8. Transient Memory Removal
9. Redundant Store Word Removal

First, using inference, a maximal set of *Var* lines are converted to *Const* lines. Once this occurs, all Add and Sub nodes are removed. Transient memory is all memory that is created by the NEW operation and then destroyed by the FREE operation. All remaining redundant memory operations are removed. All transient memory operations are removed. Finally, all transient stack operations are removed.

## 5.1 Register Removal

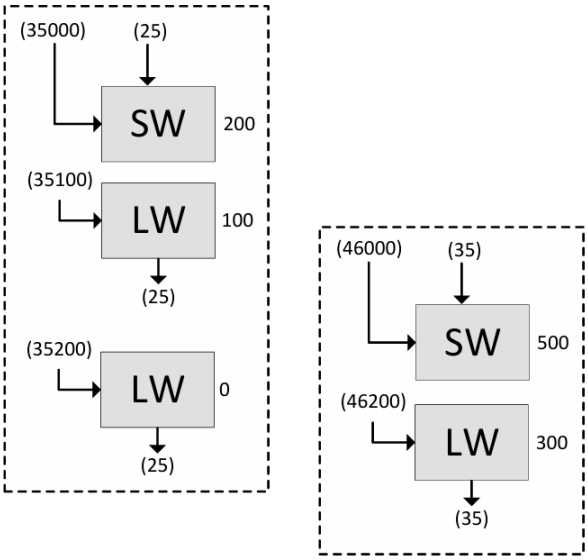
The first step involves removing the register value from all lines. This can be done because we are only interested in the starting and ending register values which are handled by the R nodes.

## 5.2 Transient Stack Memory

All LW and SW nodes whose address lines are of type *SP* are removed. To acomplish this, a temporary data structure CURRENTSTACK is maintained. The structure is lookup table of lines, each indexed by an offset value. If a LW occurs and the address is not in the CURRENTSTACK, a new S node is placed at the top of the SEDIAGRAM with the same output line as the LW node, the LW node is removed, and the entry is added to the CURRENTSTACK. If addess is in CURRENTSTACK, then the LW node is removed, and its output line is replaced by the line in the table. A SW Node whose address line is of type *SP* will insert a new entry into CURRENTSTACK and then the SW node is removed. If the value of *SP* line is less than any entry in CURRENTSTACK, then the data is out of the stack frame and the data is removed from the table. At the end of the diagram, all remaining entries to the CURRENTSTACK whose lines have changed will connect these lines to S nodes, with offsets equal to the key value. All *SP* lines and operations on them are removed and *SPChange* is set to the overall change in *SP* line values.

## 5.3 Const Reconciliation

The second step is to determine which lines are of type *Const*. Const Reconciliation is the process of converting some of the *Var* lines into *Const* lines. Reconciliation will use Add nodes, Sub nodes, and datums to process the lines. A datum is a SW node followed by a maximal set of LW nodes, where all the nodes share the same memory address. An example can be seen in Figure 20. Const reconciliation occurs by continuously applying the following set of rules and changes until there is nothing left to match.




```
1 sw $3, 200($2) # 25 35000
2 lw $1, 100($4) # 25 35100
3 sw $6  500($5) # 35 46000
4 lw $7, 0($8)   # 25 35200
5 lw $8  300($9) # 35 46200
```

Figure 20: A datum is a SW operation followed by the maximal collection of LW operations, working on the same memory location. The above illustration describes two datums, one at memory location 35200, and the other at location 46500.

| Condition | Change |
|---|---|
| If the output line of an M node is of type *Const*. | Set the input line to type *Const*. |
| If the output of a Add or Sub node is a *Const* line. | The two input lines are set to type *Const*. |
| If both inputs to an Add or Sub node are of type *Const*. | The output line is set to type *Const*. |
| For a datum, if the input line to the SW operation or an output line of a LW operation is of type *Const*. | Then all input and output lines of the datum are set to type *Const*. |
| If no other rules can be applied, and there is an Add or Sub node with all inputs and output lines are of type *Var*. | Then all inputs and output lines are set to type *Const*. |

The last rule assumes the code from which the trace was compiled does not add an unknown value to a pointer. The time complexity of this reconciliation process is $O(n \log n)$, for trace code of size $n$. Thus, after Const Reconciliation has occurred, all datums have all *Var* lines or *Const* lines.

## 5.4 Move Nodes

All M nodes are removed, and each input line is connected to the destination of the output line.

## 5.5 Const Line Stubbing

For the next step, if a datum has all output output lines of its LW nodes and the input line of the SW node of type *Const* then all the LW nodes of this datum are removed. Furthermore, every connected subgraph $G$ of the SEDiagram consisting solely of *Const* lines and C nodes are removed. This includes a single *Const* line but not a graph consisting of a single *Const* line going into or out of a C node. In its place all the *Const* lines going into $G$ and all the lines going out of $G$ are connected to a single C node. An example can be seen in Figure 21. Thus all *Const* lines will connect to a single isolated C node.

## 5.6 Add and Sub Nodes

After const reconciliation and const line stubbing, all Add and Sub nodes have one *Var* input line, one *Const* line that is coming from a Const node, and a *Var* output line. Each sequence of Add/Sub nodes and its connecting lines are transformed into a single *Var* line. The values of the *Const* line inputs are summed together to produce a singal val. If the *Var* line ends in an address input for a LW node, then the val is added to the offset. If the *Var* line ends in address input for a SW node, then the val is add to the offset. If the *Var* line

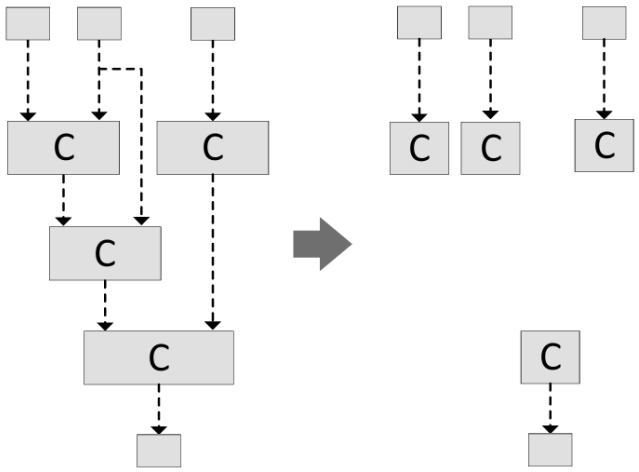



Figure 21: All connected graphs consisting of C nodes and *Const* lines are removed and replaced with C nodes connecting to the input and output lines of the graph.

ends in the input to a SW node, then the val is added to the integer offset, and similarly for ending in a Neq0, S, and R Node. This can be seen in Figure 22.

## 5.7 Var Datums

Due to algorithm of Section 5.5, the Const datums are removed. Thus the only remaining datums are of type Var. The next step is to remove all the LW operations from Var datums. For each LW operation in a datum, remove the address line, the LW, and connect the output line to the input line of the SW of the datum. This can be done because the value of the memory location accessed by LW operations will be equal to the value stored by the SW operation. An example can be seen in Figure 23.

## 5.8 Transient Memory

A set of memory locations is transient if it is created by the New node and then later removed with the Free node. Due to the Const and Var datum removal, transient memory will only have SW nodes, there are no LW Nodes. Since only the side effects are of concern, all such SW nodes are deleted. In addition, the New and Free nodes are removed as well as all SW nodes which store the memory address returned by each New operation.

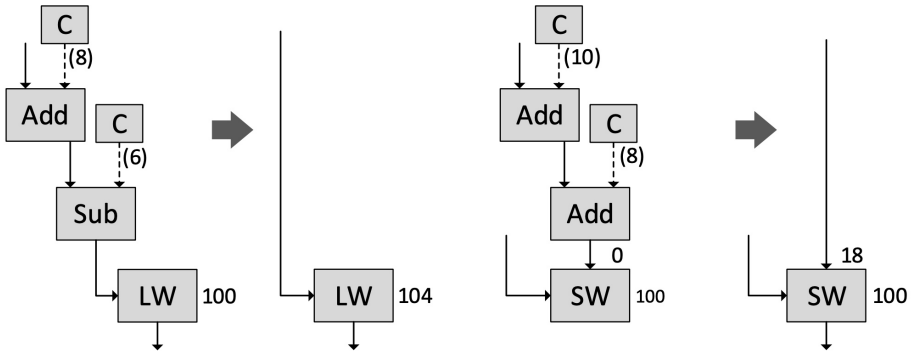



Figure 22: Sequences of Add and Sub nodes are converted into a single line. The values of the const parameters are summed together and either added to the destination address offset or integer offset.

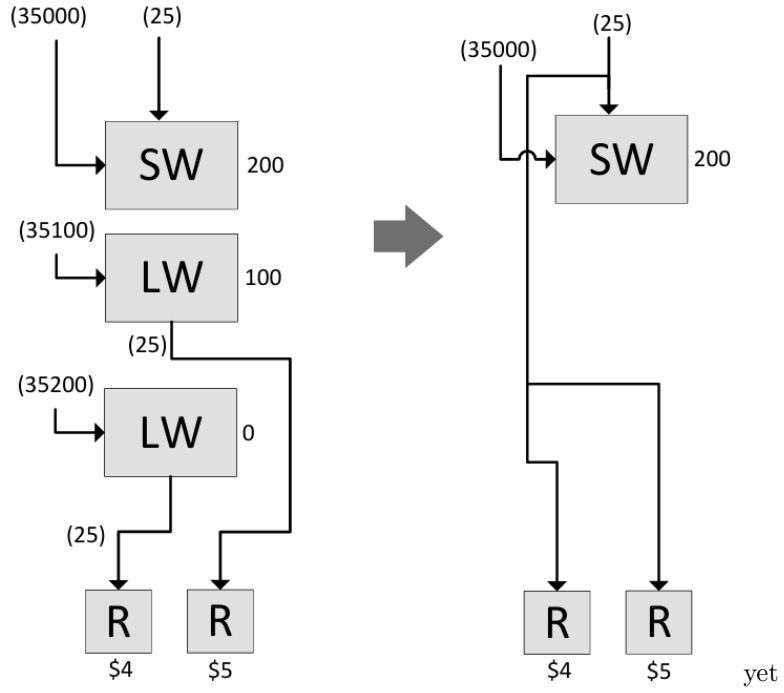



Figure 23: A Var datum is reduced by removing the LW nodes and connecting the input of the head SW node to the outputs of the LW nodes.

### 5.9 Redundant Store Words

Since the reduced SEDiagram only needs to capture the side effects of the original code, all redundant SW nodes are removed. More specifically, for each memory address, only the last SW node is saved, the rest are removed.

### 5.10 Deadend and Redundant Load Words

All LW Nodes whose output lines aren't connecting to anything are removed. If there is a LW node loading data from the same address as a LW node earlier in the diagram, then the node is removed, and output line is attached to the output line of the previous LW node. In its place, a "DLW" (short for Dead Load Word) node is created. This new node has the same address line and offset as the original node, but no output node. The reason for the construction of the "DLW" is that it is needed as a placeholder during construction of the SEConditions graph.

### 5.11 Size of Reduced Diagram

If the original trace code accessed $s$ unique heap memory locations, wrote to $r$ new memory locations and haf $t$ unique non-transient stack locations, then the reduced SEDiagram will have $O(s + r + t)$ nodes, with low constants. This is not counting the RLW redundant load words nodes which will be deleted after construction of the SEConditions graph. This is a simple consequence that each memory location is written to and read from at most once in the reduced diagram.

## 6 SEConditions

The SEConditions is a construct to see if there is a match with the SECode. If a match occurs, then the corresponding SEChanges can be applied to create the desired side effects. Intuitively, SEConditions captures all memory locations accessed and the pointer configurations of the memory of the original trace.

### 6.1 Construction

The SEConditions is an annotated graph, and is constructed from a construct called the SETrees which, in turn, is created from the reduced SEDiagram. The SCTree is a set of trees where each node of the trees is an array of ternary tuples.

The first part of the tuple is a memory location, the second part either blank or "!NULL". This indicates whether the memory location has been checked to see if it equals NULL. The third part is one of three options:

- A blank space.

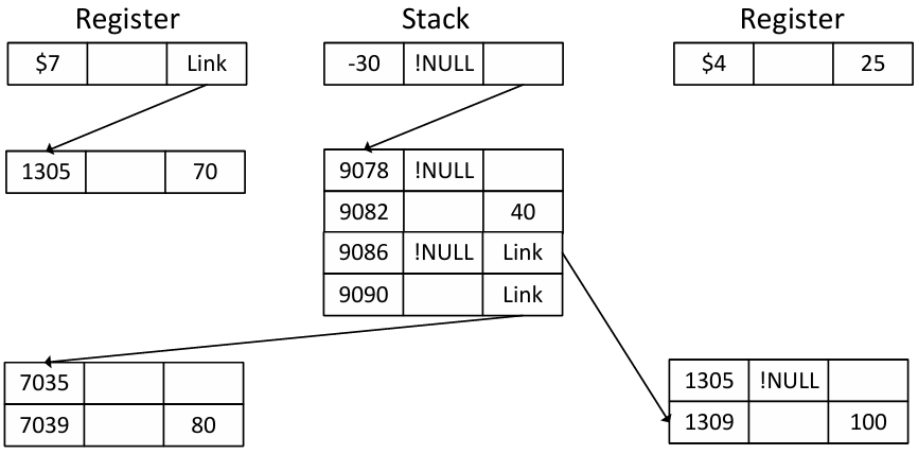



Figure 24: An example SETree. Notice that the address 1305 is indexed by two different cells. The blank cell with an address of 1305 must have been created by a Dead LW node.

- A number.
- A directed link to another node.

If the third part is a link, then it can point to any tuple in the child node; it does not have to point to the top tuple. Note that two nodes can have overlapping memory locations. In addition, there are a set of ternary tuples for registers, where the first part is a register number, the second part is either !NULL or blank, and the third part is either a number or a directed link to the head of a tree. There is also a set of ternary tuples representing stack information, where the first part is a stack offset, and the second and third part are the same as the register tuples. No two register or stack tuples can connect to the same tree.

SETrees are created from reduced SEDiagrams. This can be done with $O(n)$ time complexity. Algorithms 1 and 2 describe how this construction can be achieved. Basically, the algorithm, starting at the R and S nodes, traverses through the LW nodes of the SEDiagram and constructs nodes in the new constructed tree. An example SETree can be found in figure 24. One note is that the algorithm can be improved in the following way. A group of LW nodes are siblings if they originally share the same address line. The SETree then breaks apart its block nodes by siblings. If there is an overlap between sibling groups then the two sibling groups are combined together.

The SEConditions is similar to the SETree except the first part of the cells containing the memory address are replaced by the memory offset number from the top address and the links can form an arbitrary directed graph. Furthermore, there can be intra-block links. The blocks are numbered from 1 to $n$, where $n$ is the number of blocks A SEConditions is constructed from

**Algorithm 1** The pseudocode to create an SETree from a reduced SEDiagram. It calls the CreateBlock function in the next block of pseudocode.

```
function CreateSETree(SEDiagram diagram)
    Create an empty SETree tree
    for all R node r in diagram do
        Create a register box rb in tree
        Set register part in rb to register value in r
        if r has link to a Neq0 node then
            Add !NULL to second part of rb
        end if
        if r has a Const link to a C Node then
            Put the val of the link into the third part of rb
        else
            Create an empty cell block block
            Create an empty cell cell in block
            Create a link from the third part of rb to cell in block
            In the first part of cell, put the value of the Var line varline
            CreateBlock(var, block)
        end if
    end for
                                        ▷ The stack is handled similarly
    return tree
end function
```

**Algorithm 2** This pseudocode that will construct a node in the resultant SE-Tree. Each node consists of a block of ternary tuples.

```
function CreateBlock(Var line varline, cell block block)
    for all Neq0 node n connected to varline do
        cell ← GetOrCreateCell(block, varline.val + n.intOffset)
        The second part of cell is set to !NULL
    end for
    for all LW Node n connected to varline do
        cell ← GetOrCreateCell(block, varline.val + n.addrOffset)
        if n is connected to a Const line then
            Put the value of the line in the third part of cell
        else
            Var line childvarline ← outgoing link value of n
            Create an empty cell block childblock
            Create an empty cell childcell in childblock
            Put the value of childvarline into the first part of childcell
            Create a link from the third part of cell to childcell
            CreateBlock(childvarline, childblock)
        end if
    end for
    for all Dead LW Node dn connected to varline do
        GetOrCreateCell(varline.val + dn.addrOffset)
    end for
end function

function GetOrCreateCell(cell block block, int address)
    if block contains address then
        return cell whose first part matches address
    else
        Create new cells on either the top or the bottom of block
        so that a border cell has address equal to address
        return the newly created border cell of block
    end if
end function
```

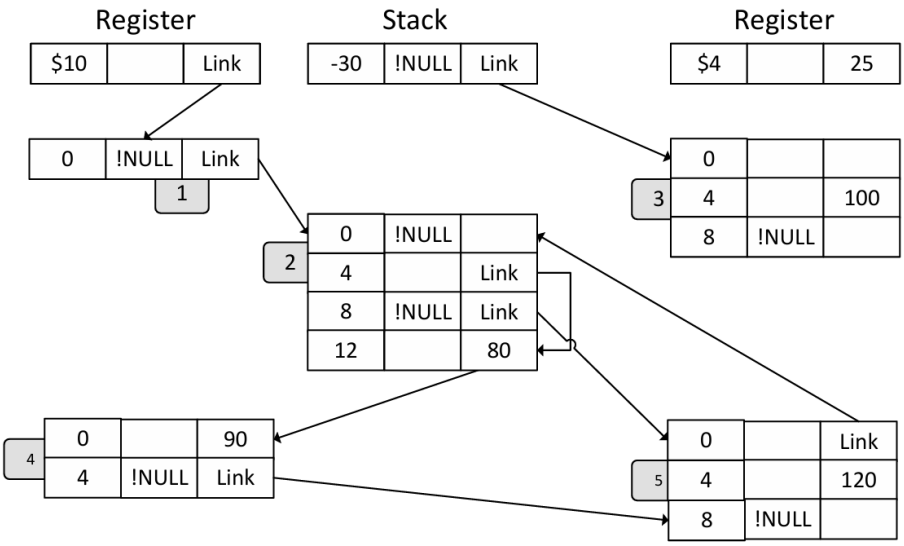



Figure 25: An example SEConditions. It is similar to an SETree except it is missing the memory addresses and the nodes can be any arbitrary directed graph. In addition, the blocks are numbered.

a SETree by merging together blocks with overlapping addresses. Due to the removal of redundant LW nodes operation in Section 5.10, if two cells in an SETree share the same memory address then on must have been blank and created from a dead LW node. A link in a cell can point to another cell in the same block. The merging of the blocks can be achieved in $O(n \log n)$ time. If the original trace code makes $s$ read operations from heap (including testing that a pointer is not null) and $t$ stack read operations, then the size of the graph of the SEConditions is not more than $s + t + O(1)$. Intuitively, the SEConditions captures the memory footprint that the original trace code will reference. It contains the constants that are required to produce a match. An example SEConditions graph can be found in Figure 25.

Note that the above construction can be improved such that links can point to ghost cells that are on the boundary of the block. This would make the matching algorithm more powerful. However it adds complications to the construction algorithm, so we leave its implementation to the reader.

# 7 Matching Algorithm

During operation of the computer, when the point in which the original trace code was found is reached, the SEDatabase uses a SEConditions to query the memory of the system to see if there is a match. If the graph of the SEConditions has $r$ nodes and the stack has $t$ non-transient read operations, then

the matching algorithm has time complexity $O(r \log r + t)$. Algorithms 3 and 4 performs the match. If there is a match, and the SECONDITIONS contains $n$ graph nodes, then the output is an array of $n$ memory locations. Otherwise the output is NoMatch.

**Algorithm 3** The pseudocode to see if the memory of the system matches with the SECONDITIONS.

```
function MatchConditions(SEDiagram diagram, System system)
    If diagram has n blocks, then create an array array of size n
    for all Register boxes r in diagram do
        Let rnum be the register number of r
        if r has !NULL and system.registers[rnum] =NULL then
            return NoMatch
        end if
        if r has number and system.registers[rnum] ≠ r.num then
            return NoMatch
        end if
        if r has link l then
            if MatchBlock(l, array, r.num, system)=NoMatch then
                return NoMatch
            end if
        end if
    end for
                                ▷ The stack tuples are handled in the same way
    Create n intervals intervals
    for i = 1 to n do
        intervals[i] ← [array[i], array[i] + diagram.blocks[i].size]
    end for
    if intervals contains overlapping intervals then
        return NoMatch
    end if
    return array
end function
```

# 8 SECHANGES

If SECONDITIONS matches the state of the system, then the output is an array of memory locations. The SECHANGES code uses the array to create new memory, write to memory, write to registers, move the stack pointer, and write to the stack. The only data constructs of SECHANGES are a table where each entry is a write operation, the an array of new memory location sizes, and the stack pointer change. Each entry of SECHANGES consists of two parts. The first part is the location of the write. It is one of the following five locations.

**Algorithm 4** The pseudocode to see if a particular block of the SEDiagram matches with memory.

```
function MatchBlock(Link link, Array array, int address, System system)
    Let link point to cell in block
    if array[block.num] is not empty then
        if address − cell.offset ≠ array[block.num] then
            return NoMatch
        else
            return Match
        end if
    end if
    addr ← array[block.num] ← address − cell.offset
    for all Cells c in block do
        if c contains !NULL and system.mem[addr + c.offset] =NULL then
            return NoMatch
        end if
        if c has num and system.mem[addr + c.offset] ≠ c.num then
            return NoMatch
        end if
        if c has link l then
            if MatchBlock(l, array, system.mem[addr + c.offset], system)=NoMatch then
                return NoMatch
            end if
        end if
    end for
    return Match
end function
```

- A register, consisting of a number. This is represented by Reg($\$n$).
- A stack offset, consisting of a number. This is represented by Stack($num$)
- A memory location, consisting of a block number and an offset. This is represented by Memory($num1$, $num2$).
- The memory location in a register plus an offset. This is represented by RegMem($num1$,$num2$).
- The memory location in a stack memory plus an offset. This is represented by StackMem($num1$, $num2$).

There are four possibilities for the second part, which is where to write from.

- A number.
- A register number plus an offset.
- A stack offset plus an offset.
- A memory location, consisting of a block number and an offset.

An example SEChanges table can be seen in Figure 26.

## 8.1 Construction

The construction of the table requires the reduced SEDiagram and the SEConditions constructs. Note that some temporary information needs to be added to SW Nodes. In particular, let union $Data$ be equal to

- Mem($blockNum$,$off$)
- Reg($\$n$)
- Stack($off$)
- RegMem($\$n$,$off$)
- StackMem($\$n$,$off$).

Each SW has $addrInfo$ and $inputInfo$ of type $Data$ which are both initially set to NULL. The first tuple contains the information in the SEConditions graph of the link pointing to the address input of the SW node. The second tuple contains the same data, except for the regular input of the SW node. The algorithm for the construction of SEChanges can be seen in Algorithms 5, 6, 7, and 8. The reason why the pseudocode is so long is because there are many ways to load and store the information. However, the general idea of the construction algorithm is relatively straightforward. The general algorithm traverses through the the reduced SEDiagram and the SEConditions and adds entry to the table of the SEChanges. For each SW node in the reduced SEChanges, two pieces of data needed to be filled: the $addrInfo$ and $inputInfo$. Once these two pieces of information are filled, there is an entry into the table corresponding to the SW Node.

| | |
|---|---|
| StackMem(100,20) | Reg($5)+100 |
| RegMem($5,30) | Stack(200)-50 |
| Reg($3) | Num(90) |
| Mem(5,100) | Mem(6,30) |
| Stack(-100) | Reg($5)+100 |
| Mem(2, 300) | Stack(200)-50 |

SPChange: +400
New Memory Sizes: (100, 500, 400)

Figure 26: An example SEChanges construct. Each row consists of a location to write and the data to perform the write. For memory, the first number is the block number from the SEConditions construct. The second number is the offset from the memory location of the top of the block. For stacks operations, the number indicates the offset. In addition there is a number specifying how much the stack pointer should change and an array of sizes for the new memory operation.

**Algorithm 5** The pseudocode to create the SEChanges table from the reduced SEDiagram and the SEConditions.

```
function CreateSEChanges(diagram, conditions)
    SEChanges changes ←NewSEChanges()
    sizes ← empty array
    for all New operations n in diagram do
        PushBack(sizes, n.memorysize)
    end for
    t ← NewTable()
    changes.sizes ← sizes
    changes.table ← t
    changes.SPChange ← diagram.SPChange
    ProcessNewMemory(t, diagram, conditions)
    ProcessRegistersAndStack(t, diagram, conditions)
    return changes
end function
```

**Algorithm 6** The pseudocode to process blocks.

```
function ProcessLink(t, varlink, cell, block)
    off ← cell.firstPart
    if varlink connects to input of SW Node s then
        if s.addrInfo ≠NULL then
            Insert(t, s.addrInfo, Mem(block.num,off + s.addrOff))
        else
            s.inputInfo ←Mem(block.num,off + s.addrOff)
        end if
    else if varlink connects to address line of SW Node s then
        if Input to s is a C Node c then
            Insert(t, Mem(block.num,off + s.addrOff), Num(c.num))
        else if s.inputInfo ≠NULL then
            Insert(t, Mem(block.num,off + s.addrOff), s.inputInfo)
        else
            s.addInfo ← Mem(block.num,off + s.addrOff)
        end if
    else if varlink connects to address line of LW node n then
        outgoinglink ← n.outgoingLink
        Let childcell and childblock be the cell and block
        that outgoinglink points to in diagram
        ProcessLink(t,outgoinglink, childcell, childblock)
    else if varlink connects to an R node r then
        Insert(t, Reg(r.reg), Mem(block.num,off + r.addrOff))
    else if varlink connects to an S node s then
        Insert(t, Stack(s.off), Mem(block.num,off + r.addrOff))
    end if
end function
```

**Algorithm 7** The pseudocode to process the starting registers.

```
function ProcessRegistersAndStack(t, diagram, conditions)
    for all Starting register r in diagram do
        for all Var link l from r do
            if l links to the input SW node s4 then
                if s.addrInfo ≠ NULL then
                    Insert(t, s.addrInfo, Reg(r.num, s.iOff)
                else
                    s.inputInfo ← Reg($r.num, s.iOff)
                end if
            else if l links to the address line of SW Node s then
                if s has C node c connected to input then
                    Insert(t,RegMem($r.num, s.addrOff), s.inputInfo)
                else
                    s.addrInfo ← RegMem(r.num, s.addrOff)
                end if
            else if l links to an R Node rdest then
                Insert(t, Reg(rdest.num), Reg(r.num, rdest.iOff))
            else if l links to an s Node s then
                Insert(s,Stack(s.off),Reg($r.num, s.iOff)
            else if l links to the addr line of a LW node n then
                childlink ← n.outgoingLink
                Let cell and block be the cell and block
                that childlink points to in diagram
                ProcessLink(t,childlink, cell, block)
            end if
        end for
    end for
    for all Ending register R Node r in diagram do
        if r is connected to C node c then
            Insert(t,Reg($r.num), Num(c.val))
        end if
    end for
                        ▷ The stack is handled in the same way as the registers.
end function
```

**Algorithm 8** The pseudocode to process the new memory created by the original trace.

```
function ProcessNewMemory(t, diagram, conditions)
    m ← NumBlocks(conditions)
    i ← 1
    for all New nodes n in diagram do
        for all Var lines l connected to n do
            if l connects to input of SW Node s then
                if s.addrInfo ≠ NULL then
                    Insert(t,s.addrInfo, Mem(m + i,s.iOff))
                else
                    s.inputInfo ← Mem(m + i, s.iOff).
                end if
            else if l connects to address line of SW Node s then
                if s has C node c connected to input then
                    Insert(table,Mem(m + i, s.addrOff),Num(c.value))
                else if s.inputInfo ≠ NULL then
                    Insert(t,Mem(m + i,s.intOff), s.inputInfo)
                else
                    s.addrInfo ← Mem(m + i, s.intOffset).
                end if
            else if l connects to an R node r then
                Insert(t, Reg(r.reg), Mem(m + i,r.intOffset))
            else if l connects to an S node s then
                Insert(t, Stack(s.off), Mem(m + i,s.intOffset))
            end if
            i ← i + 1
        end for
    end for
end function
```

| | |
|---|---|
| StackMem($a$,$b$) | Store into memory at $b$ plus the value stored in the stack at offset $a$. |
| RegMem(\$$a$,$b$) | Store into memory at $b$ plus the value stored in register \$$a$. |
| Reg(\$$a$) | Store into register \$$a$. |
| Stack($a$) | Store into stack offset $a$. |
| Mem($i$, $b$) | Store into memory at the $i$th pointer of the array plus $b$. |

Table 1: The set of locations where information can be stored.

| | |
|---|---|
| Num($a$) | Writes the number $a$ |
| Reg(\$$a$)+$b$ | Writes the contents of register \$$a$ plus $b$ |
| Stack($a$)+$b$ | Writes the contents of the state at offset $a$ plus $b$. |
| Mem($i$, $b$) | Writes the memory location at the value of the $i$th pointer plus $b$. |

Table 2: The four possible sources of information for the SEChanges table.

### 8.2 Execution

The execution of SEChanges is very straightforward. After a match by SEConditions, an array of memory locations is sent to SEChanges, along with the stack pointer shift and new memory allocation sizes. SEChanges initializes the new memory calls using the numbers in the size array. It then adds the newly created memory locations to the inputted array from SEConditions. It then performs actions specified in the table. In detail, it will first load all the values specified by the sources. Then it will store the values in the destinations. For each of the five possible location types in the first part of the entries of the table, the corresponding storage actions can be found in Table 1. For each of the four possible information source types, in the second part of the entries, the corresponding sources of actions can be seen in Table 2. With these two tables, the execution of SEChanges is completely specified.

## 9 Discussion

The goal of the machine learning component is to find very long traces with small SECode. If the SEConditions is a large graph, then the original trace code has a lot of dependences with the state of the computer. If SEChanges is a large table, then the original trace code has a lot of side effects.

Most of the time, code is in for loops, so one possible extension would be to

compile together the SECodes of the most common paths through a specified loop. Instead of storing each SECode separately, one can combine the SEConditions graphs together into a bigger graph such that when this construct is run, it could match to any of the original conditions. An avenue of research is how to minimiz the size of the combined graph. Similarly, each table in all the SEChanges can be combined together into a tree. Again, the research question is how to minimize the size of the tree. But I suspect that the tree minimization problem is very simple.

Another open question is how to incorporate floats into the SEDatabase. One option is to have the programmer manual specify in the code the conditions for a match. For example, if the code were to process an image represented by a 2d array, then the condition is that the average distance of current image to the one that is memoized be small. These conditions are then compiled down into a new type of SEDiagram. However, this adds some complications which would need to be addressed.

There are many different machine learning approaches one can take to find hot paths. One method is to ignore the arguments and find common subsequences of the overall execution trace. One can construct a suffix tree, but this operation might be prohibitively expensive if the execution trace is truly massive. However, there might be a way to construct an approximate suffix tree using only limited resources.

It might be possible to improve the $O(r \log r + s + t)$ time complexity of the matching algorithm to $O(r + s + t)$. This is because it might not be required to see if the intervals are disjoint. However this topic requires careful study.

**Acknowledgements**

I would like to thank Jonathan Appavoo, Steve Homer, Katherine Missimer, and Amos Waterland for insightful discussions.

# References

[ABH11] U. Acar, G. Blelloch, and R. Harper. Selective memoization. *CoRR*, abs/1106.0447, 2011.

[ACV05] C. Alvarez, J. Corbal, and M. Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Transactions on Computers*, 54(7):922–927, 2005.

[App14] 2013-2014. Discussions in Programmable Smart Machine Lab Team. Jonathan Appavoo, Steve Homer, Katherine Missimer, and Amos Waterland.

[AWE+14] J. Appavoo, A. Waterland, S. Eldridge, K. Missimer, A. Joshi, S. Homer, and Seltzer. Programmable Smart Machines: A Hybrid Neuromorphic Approach to General Purpose Computation. .

In *Neuromorphic Architectures (NeuroArch) Workshop at 41th International Symposium on Computer Architecture (ISCA-41)*, 2014.

[GTM99] A. Gonzalez, J. Tubella, and C. Molina. Trace-level reuse. In *Proceedings of the 1999 International Conference on Parallel Processing*, pages 30–37, 1999.

[KK20] M. Kulkarni and T. Kamble. Integration of Machine Learning into Operating Systems: A Survey. *Internation Journal of Creative Research Thoughts.*, page 1270, 04 2020.

[Mic68] D. Michie. "Memo" Functions and Machine Learning. *Nature*, 218(5136):19–22, 1968.

[Mit16] S. Mittal. A survey of techniques for approximate computing. *ACM Comput. Surv.*, 48(4), 2016.

[TPG15] L. Toffola, M. Pradel, and T. Gross. Performance problems you can fix: a dynamic analysis of memoization opportunities. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, page 607–622, 2015.

[UAZ20] I. Umit, A. Aydin, and E. Zadok. Kmlib : Towards machine learning for operating systems. In *Proceedings of the On-Device Intelligence Workshop*, 2020.

[WAS12] A. Waterland, J. Appavoo, and D. Schatzberg. Programmable smart machines. *Technical Report BUCS-TR-2012-007, Computer Science Department, Boston University*, 2012. http://hdl.handle.net/2144/11395.

[ZH19] Y. Zhang and Y. Huang. "Learned": Operating Systems. *SIGOPS Oper. Syst. Rev.*, 53:40–45, 2019.