# CPU Trace Recompilation

Sam Epstein[*]

January 7, 2025

**Abstract**

We propose an $O(n \log n)$ time complexity algorithm to recompile long sequences $C$ of assembly instructions 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 effects as $C$. SECode also matches and reproduces the effects of trace code with the same non-memory register values as $C$. Thus a key property of SECode is that it matches with trace codes that have registers pointing to different memory locations. If $C$ uses $s$ non-transient heap space and $t$ non-transient stack operations, then the size of SECode is $O(s+t)$, and it runs in time $O(s \log s + t)$, with small constants. This is without code annotations.

## 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 Programming 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]:

"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

[*]samepst@jptheorygroup.com

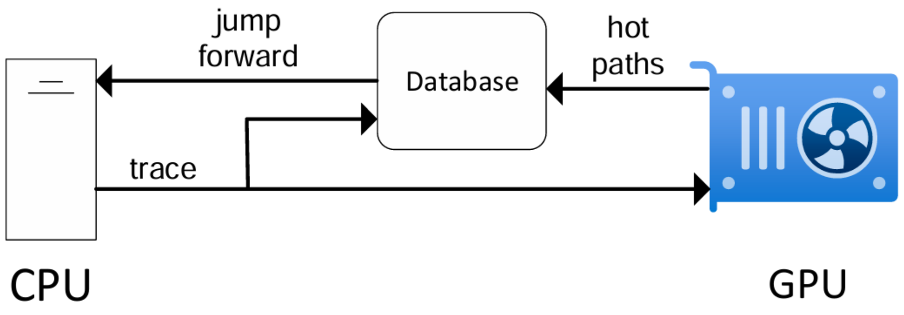


Figure 1: The proposed setup for exploiting hot paths.

> 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 indicate 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 proposes 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, but some discussion on this topic is included. We introduce a new method to directly compile, in $O(n \log n)$ time, large streams of assembler trace $C$ into new code that contains a minimal set of conditions and side effects of $C$. This new compiled code is called SECode, short for side effect code. SECode contains conditions, called SEConditions and changes, called SEChanges. If $C$ uses $s$ non-transient heap space and performs $t$ non-transient stack operations, then the size of SEConditions is $O(s + t)$ and it runs in time $O(s \log s + t)$. The running time of *SEChanges* is $O(s + t)$. e SEChanges is highly optimized. The following components of $C$ are removed:

- Redundant memory operations.

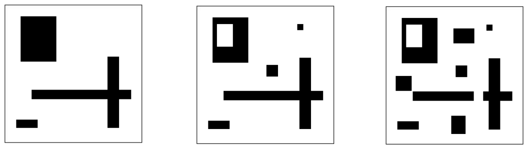


Figure 2: The evolution of the state of the computer, represented as a binary image.

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

Very large FOR loops are handled by compiling together the SEConditions of the most common paths taken. SECode is compiled without any programmer annotation. This method can also handle floating point registers with approximate programming. However this requires explicit markup in the code by the programmer.

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, this approach has some difficulties. Take, for example, the following code. It starts with an input of two integer variables $n$ and $m$ and a pointer *head*. It constructs a $m$ sized linked list starting at *head* and at node $i$ it stores $func(n, i)$, for some complicated math function $func$. It then terminates. Because the *head* pointer is different at every run, one cannot use the method described in [AWE+14]. One would like the recompiled code to be pointer agnostic. Indeed, in this paper, SECode treats pointers as variables, so it can create the linked list as well as memoize $func(n, i)$ for every node. The following criterion is used to distinguish pointers[1] from regular variables:

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

An example is trace assembler code $C$ that sorts a doubly linked list, with starting values $(5, 3, 2, 4, 1)$. If $C$ is compiled into SECode and sent to the SEDatabase, then the next time $C$ is reached and a match $(5, 3, 2, 4, 1)$ is made, then SEDatabase will automatically jump to the list being sorted, $(1, 2, 3, 4, 5)$.

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

Figure 3: The original code is compiled into SEDiagrams which are, in turn, reduced and recompiled into SECode.

This will happen each time $C$ is reached and matched to $(5, 3, 2, 4, 1)$, regardless of the particular pointer values of each run.

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. Discussion of this component can be found in Section 7. 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 entire process 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 $s$ non-transient heap space and $t$ non-transient stack operations. For each compiled SECode the SEConditions checks whether their is a match in $O(s \log s + t)$ time. Multiple SEConditions can be compiled together into a conditional tree structure. If, during the course of operation, a SECondition is satisfied, then the output is an array of pointers. Using this array, the side effects of the compiled code are computed with its corresponding SEChanges code in $O(s + t)$ time. After that, normal operation resumes.

## 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. Section 6 addresses how the SEDatabase handles floating point numbers. We assume the reader is familiar with operations of 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.

### 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 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 returns the address location in another. 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.

### 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. SEDiagramns 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 4. 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 used to create the SEConditions and SEChanges. The SEConditions construct is an annotated directed graph representing pointers and constant values. The SEChanges is assembler code to be run on the SEDatabase that will implement the side effects of the original code. 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

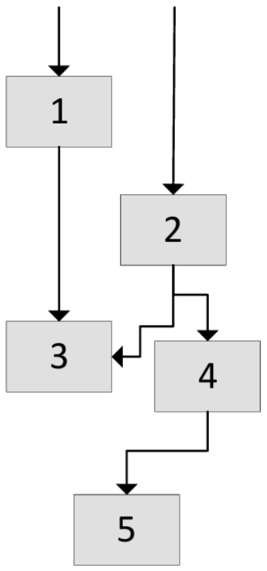


Figure 4: An abstract SEDiagram. The nodes of the SEDiagram are compiled into SECode from top to bottom, in the order specified.

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.

- Added by a number.
- Subtracted by a number.
- Compared to 0.
- 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 5.

## 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

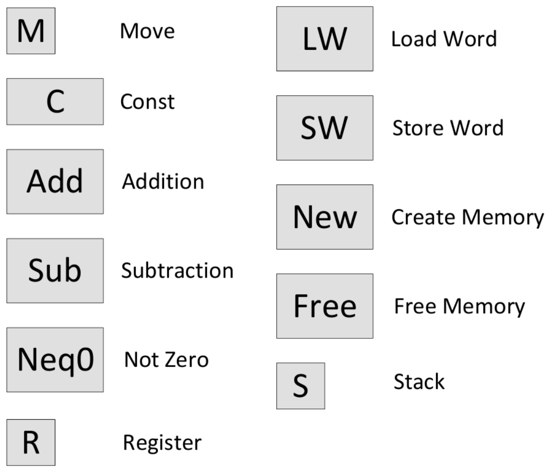


Figure 5: The eleven different node types of the SEDiagram.

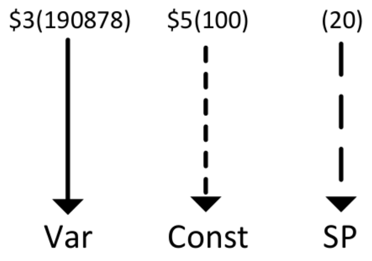


Figure 6: The three different type of lines. The *Var* line is on register \$3 with val 198078. The *Const* line is on register \$5 with val 100. The *SP* line has offset equal to 20.

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 to them is 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 ser of lines at the bottom of the SEDiagram.

Figure 6 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 7.

## 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. 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.

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 7.

## 4.3 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 8.

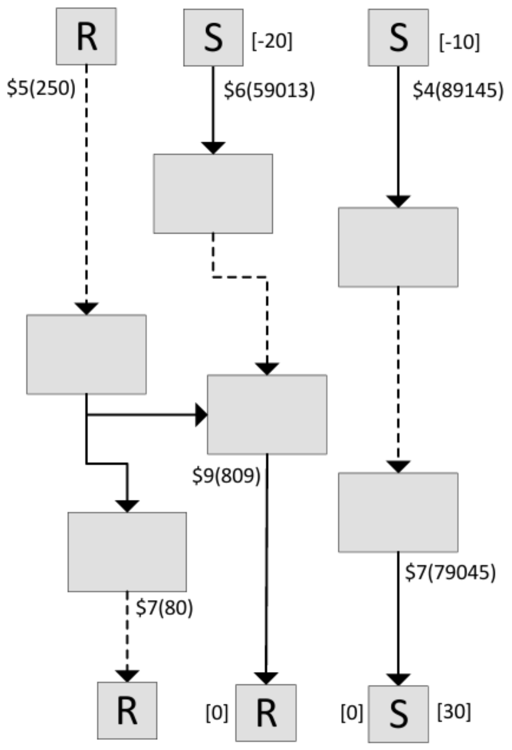
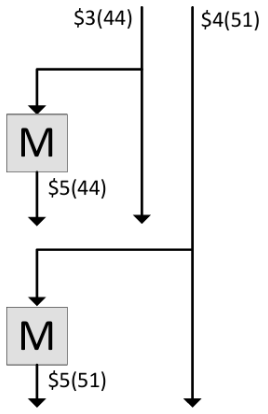


Figure 7: The R and S nodes at the top and bottomrepresent the starting and ending states of a subset of the registers and stack, respectively. The S nodes have offset information which is their right. The nodes at the bottom with incoming *Var* links have an additional integer offset information on the left, which always starts at zero.



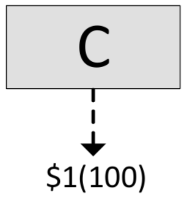
```
move   $5 ,  $3 # 44  44
move   $5 ,  $4 # 51  51
```

Figure 8: 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 Const

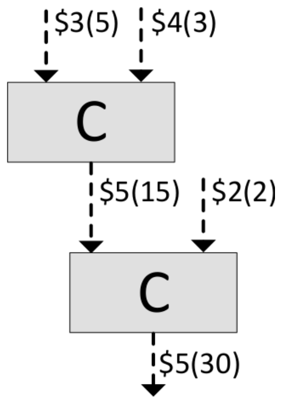
The input to a Const node is zero, one, or two lines and the output is zero or one line. When a Const node is created, if an input line is of type *Var* then it will be changed to type *Const*. An example of a Const node with zero input lines can be seen with the LI command, which loads a constant into a register. Its corresponding SEDIAGRAM can be found in Figure 9.

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



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

Figure 9: The SEDIAGRAM constructed from one LI operation.



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

Figure 10: The SEDIAGRAM constructed from two MUL operations. Note that the diagram would be the same for DIV, AND, or OR operations. Note that that, to remove clutter, the active lines are not extended to the bottom of the diagram, though technically they should be.

nodes in tandem can be found in Figure 10.

## 4.5 Add and Subtract

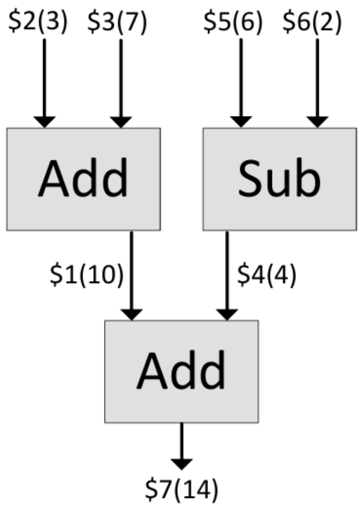
The ADD and SUB instructions will create Add and Sub nodes, respectively. Both nodes take two inputs and one output. They can be seen in Figure 12.

## 4.6 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 9.

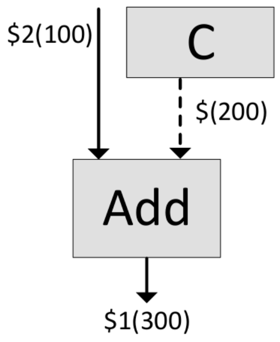
## 4.7 Branch on (Not)Equal

The BEQ command branches if the two register arguments are equal. Similarly, the BNE command branches if the two register arguments are 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,



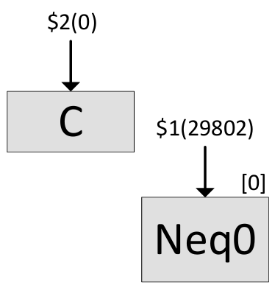
```
add $1, $2, $3 # 10 3  7
sub $4, $5, $6 # 4  6  2
add $7, $1, $4 # 14 10 4
```

Figure 11: The SEDIAGRAM constructed from two ADD operations and one SUB operation.



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

Figure 12: 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.



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

Figure 13: 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.

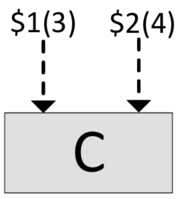
if the arguments are two pointers, as we will see later in the paper, all computer states that match the SECondion will result in an identical branch outcome. However, pointers can be tested to see if their values are 0. For a beq or bne command with a $0 argument, if the other register argument has 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 var is any value. The Neq0 node also contains an input offset, to be used in the construction of the SEChanges code. This can be seen in Figure 13.

## 4.8 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 14

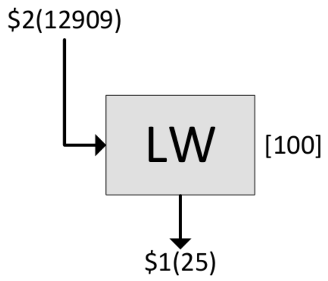
## 4.9 Load Word

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*. This operation creates a single LW node as shown in Figure 15.



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

Figure 14: The BLT operation produces a Const node with two input *Const* lines. There is no output lines.



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

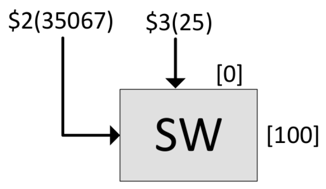
Figure 15: The LW operation produces a LW node with on input *Const* line and one output line. The address is on the left side of the node and the data that was loaded is outputted at the bottom. The address offset is on the left.

## 4.10 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.

## 4.11 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



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

Figure 16: 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.

in Figure 17

## 4.12 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

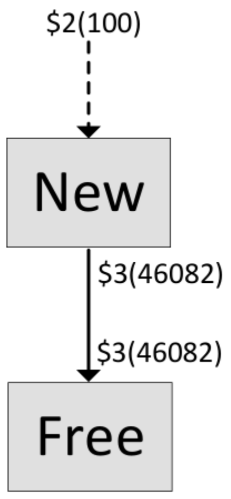
## 4.13 Implementation

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. A example SEDiagram from a block of code can be found in Figure **??**.

## 5 SEDiagram Reduction

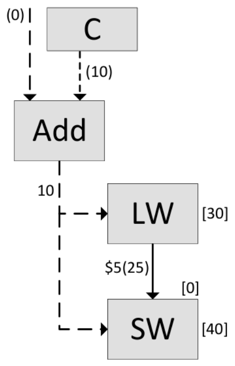
Once the SEDiagram has been constructed, it is then reduced. This reduced SEDiagram differs from the original one in a couple of ways. Firstly, the *Var* lines don't have register values associated with them. Secondly, all *Const* lines are connected to a c node. Thirdly, the integer offsets in the LW,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

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



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

Figure 17: 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.



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

Figure 18: 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.

by the FREE operation. All transient memory operations are removed. All remaining redundant memory operations are removed. Finally, all transient stack operations are removed.

## 5.1 Const Reconciliation

The first 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 **??**. Const reconciliation occurs by continuously applying the following set of rules and changes until there is nothing left to match.

| 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.

## 5.2 Const Line Stubbing

For the next step, all the LW operations of the Const datums are removed. Furthermore, every connected graph $G$ consisting 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 **??**. Thus all *Const* lines will connect to a C node.

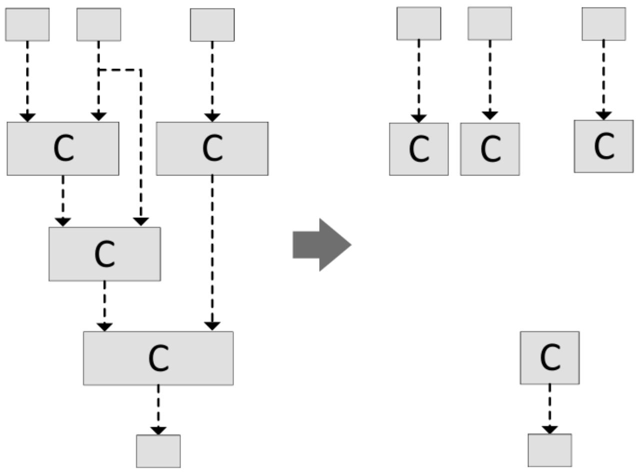


Figure 19: All connected graphs consisting of C nodes and *Const* lines are removed and replaced with C nodes connecting to the input and output lines of the graph.

### 5.3 Add and Sub Removal

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. This

## 6 Floating Point Numbers

## 7 Machine Learning

## 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. 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.