Blog on Assembly Theory

Sam Epstein

January 22, 2025

18: The Spectrum of Trace and Codes

On one hand you have code which can be run to execute algorithms. On the other hand you have traces which are records of information. But there is SO much information in traces that can be recovered. Assuming there is no error, for which there is no guarantee, then SEEnsembles represent the halfway point between code that can be executed and the records of computation, i.e. traces. By combining the top 1000 (ish?) traces you are partially reconstructing the code which generated them. In the limit, SEEnsembles become actual code. Then you have a construction: SEEnsembles that starts out a single trace and as traces are added, in the limit, SEEnsembles will become actual code!

17: Remarks on the Code

For the code, I wrote into the C to MIPs compiler options for memory creation and deletion. The MIPs code is sent to the SPIM simulator which outputs the trace which are the instructions that were run along with the values of the register arguments. The SPIM simulator was modified to have a memory model, so that you can create and delete memory from the heap. The trace is sent to the JPSegment C++ code, which produces a reduced SEDiagram. Then JPSegment removes all transient memory operations. However the code does not include the creation of SEGraphs, SEConditions, and SEChanges. This the current code state but I believe it supports proof of correctness for the SEDatabase.

16: Abstract of Second Paper

We introduce the notion of SEEnsembles. Given a loop of code with a massive amount of iterations, one can take the most popular traces, and then convert them into SECodes. The SECodes are compressed together into a construct I'd like to call SEEnsembles. SECodes can be added or removed depending on how many hits they get. A second CPU can try to compress the SEEnsembles to allow for more traces to compile into it. This is the most insane fucking thing.

15: Adding and Removing Traces From SEEnsembles

Traces can be dynamically added or removed from ensembles. When a set of traces is removed or added, the constraints are recompiled together. You can keep a tally for each trace, and if a new trace matches with it, then the tally is incremented. The traces that are underperforming will be culled. In fact, you can have a separate processor for the SEDatabase that can constantly trying to compress the SEEnsembles. When it is compressed, then it can add more traces.

14: Merge into One Paper

I've decided to merge the two papers together. It will encompass how to functionalize the consts, how to handle floats, and ensembles of traces. The key issue is how to compress the SECodes traces together. This can be done by merging together the constraints of each trace together to form a constraint tree. You start at the top of the tree, and if you reach a leaf node, then you have a match. Traces can be dynamically added or removed from SEEnsembles to compensate for changes in the CPU state.

13: Online Construction of SEEnsembles

In fact, you can handle constructing SEEnsembles in an online fashion. When a new trace t comes in, it compiled into SECode. It is put into an existing or new memory pointer profile of the SEEnsemble. The constraints need to be recomputed and this can be done in $O(nm)$ time where n is the number of traces, and m is number of nontransient heap operations.

12: Handling Constants

My first paper can also handle floats. Let me explain. Lets say the input is a linked list of size four. The trace takes in $(6, 9, 2, 10)$ and sorts the list. One way to memoize is to have the SEConditions for $(6, 9, 2, 10)$ and if there is a match, have the SEChanges output directly $(2, 6, 9, 10)$. Take another example, where there is a linked list of size 100 and the trace multiples every number in the linked list by two. The memoization can actually learn this and when a linked list of size 100 comes in, the SECode multiples every number by two. In this case, the numbers can be any value. In general there will be constraints on the values. These constraints are created from the branching operations BEQ, BNE, BLT, etc. Its hard to explain why this is. Furthermore because multiple traces will form a tree with respect to the branches, constraints of different traces can be compressed together. This general setup works on ints and floats. There's no clear rule on what numbers to make constant and what numbers to make variables. In general, highly mathematical code cannot easily be turned into functions. This is because the constraints will become enormous functions. But code that doesn't have such computations, such as database code, appears to be amenable to this method.

11: Implementation

I have a working version of the paper implemented in C and C++. Its from early 2024 and its based on an early draft version. I use a C to Mips compiler, though it cheated by saving everything on the stack. For the simulator, I used SPIM, but I added a memory model to it to allow for frees. My working version wasn't able to do much because I didn't have the memory-equivalence class idea then. But it did do SEDiagram creation and reduction. I was actually able to remove memory operations performed in transient memory. I even got it working on heirarchal transient memory. I can safetly say it works.

10: SEEnsembles

I have a clear idea for my second paper which is how to memoize in a loop with the 1000ish best traces. The traces will have functionalized consts which is from the first paper. Their SEConditions and SEChanges can be compressed together in an innovative way. For each trace, there is a tradeoff between tightness of its matching constraint versus the efficiency of the corresponding SEChanges construct.

9: Compressing Constraints

In the SEEnsemble, each trace will have a constraint that is the condition for the side effects to be used. Constraints are created from branching operations such as beq, bne, and blt. However if you take all the traces, the branches will form a tree. Thus you can actually compress constraints together by this tree. You're basically changing each trace into a constraint. You can then use its properties to make the SECode more efficient but the constraint tighter, or loosen the constraints at the cost of SECode speed.

8: The Information Content of Assembly Traces

There's a remarkable amount of information content in assembly traces. By encoding them into directed acyclic graphs, you can tease apart what sort of information is streaming through the traces. I'm planning to write two papers: the first one will be about turning constants to variables. The second one will be about ensembles of traces. For example, take a trace that recieves a linked list of size 5 and doubles the number of every node. For the first paper, an improvement can be made such that when the corresponding SECode is compilied and a new trace comes in with a linked list of size 5, then the values in the nodes will be automatically doubled. This is an example of how you can partially recover code from traces.

7: Writeup

My original excitement is well founded. Theres a complication but the overall setup is the same. Each trace results in a constraint and if the new trace matches

a constraint then the side effects can be used. I'm going to try and write it up.

6: From Constants to Functions

Currently, SECONDITIONS and SECHANGES deal with pointers or constants. However there could be a way to have the write values be a function of the read values. Thus instead of creating a const node, one has the out line as a function of the in line. Then, when it goes into a SW node, the functions are composed together. Branch operations will create constraints between the initial values. This area is very exciting! You can get every write operation to be the minimized function of every read operation. Furthermore, every branch is transformed into a precise constraint for the reads!

5: Machine Learning Component

The machine learning component wasn't addressed in my paper. The goal is to find hot paths. Technically, this means finding a memory equivalence class that is run often and has small SECONDITIONS and SECHANGES. One way to do this is to disregard the arguments and only deal with the commands. The reason for this is one doesn't know what argument is a pointer or a constant unless on creates a reduced SEDIAGRAM. Furthermore, its my belief that the control flow will be adequately captured if only the commands are taken account of. So one can construct a suffix tree in $O(n)$ time of a trace containing only commands. Long branches with many matches make for good candidates for hot paths.

Another method is to look at loops and for each run through the loop again take only the commands. Then put each list into a hash table. Then send paths with the highest hash count to the SEDATABASE for processing.

One wants hot paths with a very high number of transient operations. One can count the number of NEW and FREE operations, stack pointer manipulations, and math operations. Hot paths with high scores make for good candidates to the SEDATABASE.

4: Bounds on Combination

Good news. Let N be the number of SECONDITIONS to be combined. Let m be the number of unique memory equivalence classes. Let s be the number of nontransient heap operations. There's a way to combine the SECONDITIONS such that the combined graph has space $O(Ns)$ and can make matches in time $O(((\log N)m + \log s)s)$. Thus, if you have the space, you can quickly determine if there is a match. This is because the constant values can be sorted. We reiterate $O(Ns)$ is the worst case. On average, the space used is going to be much better because one can take advantage of the redundancy of the constants.

What this means is that given the best 1000 traces, their SECONDITIONS can be combined together in a really efficient way. For example, if the input is a linked list then the memory equivalence classes group the lists by their size

and then number values of the linked lists are actually sorted in the combined SECONDITIONS construct.

3: Floats

An interesting question is how to handle floats. The first thing to note is that a float always produces a *Const* line. However the corresponding SECONDITIONS will need work because the num values will be floats. It's unrealistic to think that two traces will have exactly the same float value. Thus it is an open question on how to modify the num values in SECONDITIONS to handle floats. One method is to specify an average error value threshold between the num values. Another method is to have special programmer code that compiles into the SECONDITIONS. Another idea is as follows. For example, say you produce the SECODE for the top 1000 float traces. When a new trace comes in, the SECHANGES of the trace with the closest float values to that of the new trace is used to compute the side-effects. Due to efficiency, you might want to test only a small fraction of the float nums.

2: Combining Conditions and Changes

I'm going to go ahead and write a followup paper entitled "Greedy Combination of Conditions and Changes in Assembly Theory". It will detail the greedy algorithm to merge the SECONDITIONS and SECHANGES constructs. This will codify what I'm talking about in post 1. I thought the idea to be relatively simple but actually there are some cases that make the endeavor quite tricky.

1: Welcome to Assembly Theory

It apppears that there's no reason why SECONDITIONS and SECHANGES can't be combined together. This means given the best thousand traces, they'll turned into a two part code of conditions and changes and then the constructs will be compressed together by their likeness, all in $O(n \log n)$ time. Each time a trace that comes in with memory-isomorphic match to one of the thousand, then its side-effects will be computed with a single table.