CPU Hot Path Recompilation

Sam Epstein*

January 3, 2025

Abstract

We propose a $O(n \log n)$ time complexity algorithm to recompile long sequences of assembly instructions into a novel code that captures a minimal set of conditions and side effects. This new code can be used to cause the state of the computer to jump forward, increasing the efficiency of the CPU. Transient memory, register, and stack operations are removed during recompilation. This also includes redundant memory operations and all math operations. The new code is completely pointer friendly. With FOR loops, the most common paths in each loop are compiled together to create a condition and side effect directed acyclic graph. This is accomplished on raw assembler code, without any extra programmer notations. Floating point operations are handled directly or approximately.

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.

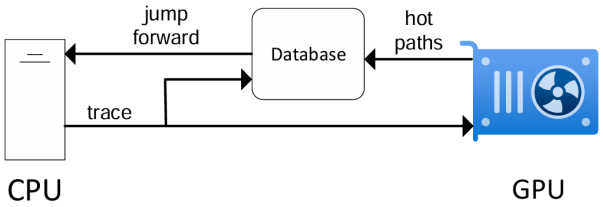

Figure 1: The proposed setup for exploiting hot paths.

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 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."

*samepst@jptheorygroup.com

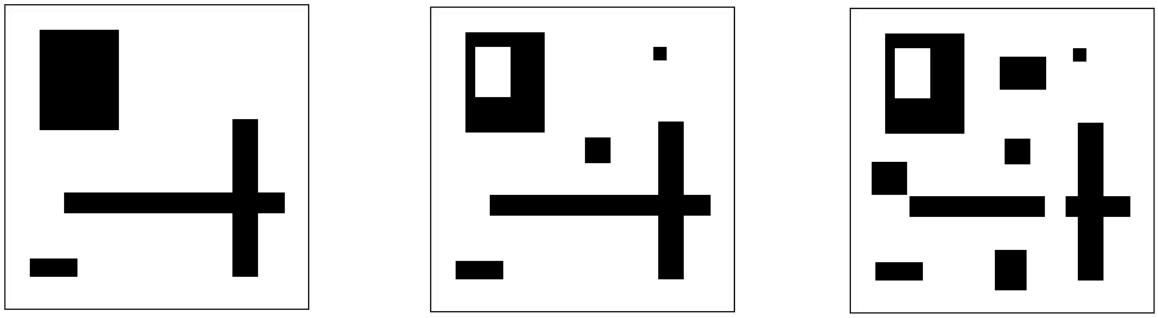

Figure 2: The evolution of the state of the computer, represented as a binary image.

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 D that contains a minimal set of conditions and side effects of C. This new compiled code is called SECode, short for side effect code. The CPU runs the new SECode D and if all the conditions are satisfied, then the side effects of D are incorporated into the system. Otherwise, the changes of D are discarded and system returns to running C. The SECode D is highly optimized. The following components of C are removed:

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

Very large FOR loops are handled by only compiling the most common subset of 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.*

For example, take trace assembler code C that sorts a doubly linked list, with 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, $(1, 2, 3, 4, 5)$. 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.

2 Setup

In this section, the overall setup is discussed. There are two phases, 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 by so called SEDiagrams. We will use the term graph and SEDigrams interchangeably. SEDiagrams can be reduced to the exact minimal set of conditions and side effects of the original code. Then the SEDiagrams is recompiled into SECode. The entire process can be completed in $O(n \log n)$ time. This configuration can be found in Figure 3

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

There are two ways to implement the execution of the SEDatabase. The first method is as follows. The virtual memory is separated into two regions. The first region contains the regular memory of the computer. The second region contains memory needed by the SEDatabase. When a section of SECode is reached, progress on the regular computer is halted. The SECode is run, using the second region as memory. The SECode can either *fail* or *finish*. The SECode fails if a condition is not met. In this case, the virtual memory in the second region is cleared and the computer resumes its operation. If the SECode finishes, then the changes specified in the second region is applied to the first region, and the computer resumes normal operation.

The second method splits the virtual memory into three regions. The first region contains the regular memory of the computer. When a match to a segment of SECode is found in the SEDatabase, two separate CPUs start operating. The first CPU executes the original code, with changes only saved in the second virtual memory region. Some work is needed to keep the call stack properly synchronized. The second CPU executes in parallel to the first CPU, the SECode. If the SECode finishes before the execution of the first CPU, then the changes in the third virtual memory region is applied to the first region and normal operations resume.

If the SECode fails or finishes after the first CPU has completed, then the second CPU stops. The first CPU finishes any computation, and the changes from the second virtual memory region is applied to the first and normal operation resumes. After the overall operation is completed, regardlesss of the outcome, the second and third virtual memory regions are wiped.

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

3 Related Work

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.

4 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 7 addresses how the SEDatabase handles floating point numbers. We assume the reader is familiar with operations of registers and the stack. Asside from the stack and frame pointer, all registers are denoted by $n, for some number n. We assume a small, but unspecified, number of registers. The following MIPs operations will be used in this paper.

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, and DIV. 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). SEDatabase handles branch on equal exactly how it handles branch on not equal, BNE, so this operation is not included.

Branch on Less Than

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.

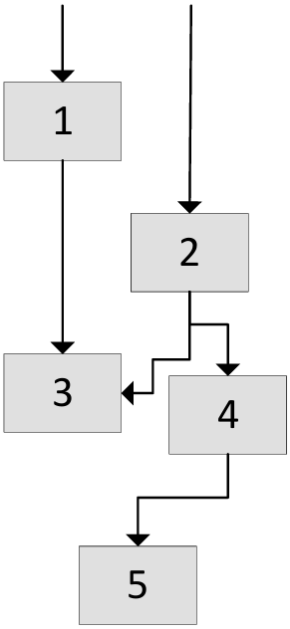

Figure 4: An abstract SEDiagram. The nodes of the SEDiagram are compiled into SECode from top to bottom, in the order specified.

5 SEDiagram Overview

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 or one node is 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, removing redundant operations. The SEDiagram is then compiled into SECode, translating the nodes in vertical order, from top to bottom.

6 Floating Point Numbers

7 Machine Learning

References

[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.

[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.

[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.