Ideas from the Synthesis Kernel in Modern Systems: Optimization techniques and SuperOptimizers

Sridhara Madhu Mohan Nelemane
University of Bamberg
Bamberg, Germany

ABSTRACT

The paper from Henry Massalin [1] was considered futuristic for its time. Several ideas were proposed in this paper towards developing a much more sophisticated operating system. These ideas particularly focussed on performance of the operating system and an implementation called Synthesis kernel was used to drive the concept home. This seminar paper refers to the same research and implementation and tries to provide an understanding some of the same ideas in the modern context. In particular the paper focuses on techniques of SuperOptimization utilized in Runtime Code Generation phase of the Synthesis Operating system.

CCS CONCEPTS

• **Operating Systems Design-Performance**; • **Optimization - Techniques**; • **SuperOptimization**; • **Synthesis Kernel**; • **Runtime Code Generation**;

KEYWORDS

datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
Sridhara Madhu Mohan Nelemane. 2018. Ideas from the Synthesis Kernel in Modern Systems: Optimization techniques and SuperOptimizers. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX).* ACM, New York, NY, USA, 3 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Before the invention of hyperthreading and powerful processors with big working memories as we have today, the processors had meagre processing power and very low RAM capacity. In such processors, running complex algorithms were challenging. Operating system itself consumed significant resources and optimization was one of the key research topics for operating system experts. It was under these circumstances that Henry Massalin proposed ideas for Operating system optimization. These ideas are extremely relevant even today. However, it needs some investigation with a modern perspective as demonstrated in the blog [2]

The paper introduced the concept of Runtime Code generation helping build purposefully optimized code. The techniques included among other ideas, constant folding, constant propagation, and procedure inlining. The paper also proposed the idea of superoptimizer which searches for the smallest of possible programs to achieve a given objectives. These and other ideas are explored in later sections of this paper.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

2 SYNTHESIS KERNEL AND OPERATING SYSTEMS DESIGN

The basic premise of the Synthesis kernel proposed in [1] is that operating system performance can be enhanced through few strategies of optimization during runtime code-generation. Instead of static code, runtime code generation provides an opportunity to dynamically add optimizations in code based on runtime conditions of the system.

2.1 Key concept and Ideas for performance enhancement

The paper [1] proposes two important techniques

2.2 Runtime Code generation and Optimization

- Runtime Code Generation

3 TRADITIONAL OPTIMIZATION STRATEGIES

This section revisits some of the traditional optimization strategies at the level of Assembly language to fully utilize processor capabilities. Most general purpose and specialized processors implement multi-stage instruction pipelines. The most common issue in the pipelines is pipeline stalls. This happens when an instruction has to wait for another instruction to pass a certain stage. For example, a Mov instruction waits for the previous Add instruction to write to the register which is used in the Mov instruction. Several optimization techniques are deviced to avoid such stalls.

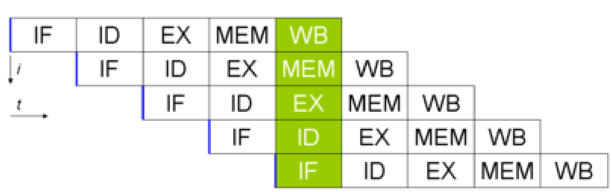

Figure 1: Wikipedia, the free encyclopedia, "Atomic force microscopy," https://en.wikipedia.org/wiki/File:Fivestagespipeline.png

3.1 Avoid Decision statements

Conditional statements like 'if....else...' translate in assembly to a set of JMP instructions. The JMP instructions require a few additional steps like storing the current Program Counter and updating the Stack Pointer so that the program can resume graceful execution when the control returns to the point from where the JMP was called. These additional steps cost CPU cycles and when possible should be avoided. Several hacks are available to counter these through Arithmetic and Logical operations to obtain results similar to what the conditional statements do. In case it is impossible to remove the conditional statements, there should at least be an effort to move the condition statements out of a loop construct so that the overheads related to JMP instruction are not repeated several times.

Example:

```
int maximum(int a, int b) {
    if (a > b)
        return a;
    else
        return b;
}
```

Can be replaced by:

```
int maximum(int a, int b) {
   return -(((b - a) >> 31)*a + ((a - b) >> 31)*b);
}
```

Several modern compilers are now capable of replacing such conditional statements with logical and arithmetic instructions.

3.2 Precompute Constants and Constant Expressions

Sometimes, code can contain variables that for certain use-cases are not subject to change. Such variables should be eliminated and replaced with constants and constant expressions. For the purpose of clarity, such constants can be implemented using Macros which are computed during compilation.

3.3 Procedure Inlining

Similar to Conditional instructions, procedure calls involves several intermediate steps like storing the program counter, updating the stack pointer and then calling a JMP instruction to the procedure being called and a similar set of steps while returning to the caller. This can be avoided with procedure inlining. Procedure inlining provides an elegant way of structuring the code in a modular fashion while not losing the performance gain by executing the instructions in sequence without the overhead of function call. This is generally accomplished by either a prefix for the procedure definition or an annotation placed before the procedure depending on the type of higher-level language used.

3.4 Loop Unrolling

Loops are often the most compute-intensive sections of the code. Pipeline stalls in loops can be hazardous for the performance of programs since the delay is multiplied by the number of iterations. Hence avoiding pipeline stalls in loops is extremely critical. Loop unrolling is done in three steps:

(1) Preload registers that are used in computation
(2) Precompute one sample before the loop begins
(3) Align compute of older samples with loading of new samples to avoid pipeline stalls

The method best illustrated in the program below. Note that though this technique optimizes the loop, it is based on the assumption that the processor has enough registers available at this point to be utilized in the loop.

```
Loop :
    load R1,(R6)
    load R2,(R6+1)
    mac R1,R2,R3      //..2 t i c k s
    store R3,(R9)
    add R6,2
    inc R7
EndLoop
```

The unrolled version is as below:

```
load R1, (R6)
load R2, (R6+1)
add R6, 2
Loop:
    load R4, (R6)
    mac R1, R2, R3 ..2ticks
    load R5, (R6+1)

    add R6, 2
    store R3, (R9)

    load R1,(R6)
    mac R4, R5, R7
    load R2, (R6+1)

    add R6, 2
    store R7, (R9)
    inc R9
Endloop
```

3.5 Utilize Special Architecture Capabilities

Some processors also come with specialized hardware in the form of co-processors, special Multiply-Accumulate units, fast I/O units and others. The compilers may not fully utilize these special facilities provided for the processor. Such handling requires customization of the compilers or disabling compiler optimization and manually modifying sections of code to utilize the capabilities better. For example, most multimedia SOCs provide an additional DSP or a Co-Processor for specialized computation and they can do computations independent of the registers and ALU of the main processors. The compiler may not take advantage of such feature while optimizing and hence require manual intervention by specialist programmers.

In such scenarios, the specialists have advanced knowledge of the Instruction architecture of the processor and the special-purpose processors. They decide on which part of the code shall be executed on the processor and which part would be redirected towards special purpose processors. The special purpose processors could be Signal processors, controllers or just a coprocessor with additional MAC units to enable parallel processing. The programmer can use a switch like #pragma to separate code that should run on the special purpose processor.

4 SUPEROPTIMIZATION

4.1 Understanding Superoptimization

4.1.1 The Method.

4.1.2 Limitations.

4.1.3 Applications.

4.2 Peephole Superoptimizers

5 SUPEROPTIMIZERS IN MODERN COMPILERS

6 OTHER RELATED WORKS AND FUTURE DIRECTION

7 CONCLUSIONS

REFERENCES

[1] Henry Massalin. 1992. Synthesis: An Efficient Implementation of Fundamental Operating System Services. (1992).

[2] John Regehr. 2019. *It's time for a Modern Synthesis Kernel.* https://blog.regehr.org/archives/1676