# 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 [6] 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, 5 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 [9]

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*
© 2018 Association for Computing Machinery.
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 [6] 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 [6] introduces run time code generation to optimize kernel performance. The technique benefits from more information regarding the execution and the environment during runtime.

### 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.

### 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

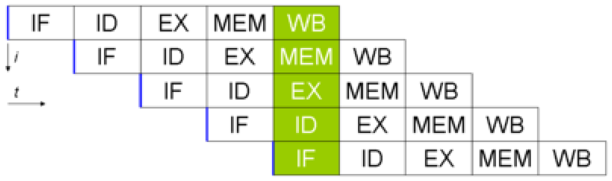


**Figure 1: Wikipedia, the free encyclopedia, "Atomic force microscopy," https://en.wikipedia.org/wiki/File:Fivestagespipeline.png**

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.

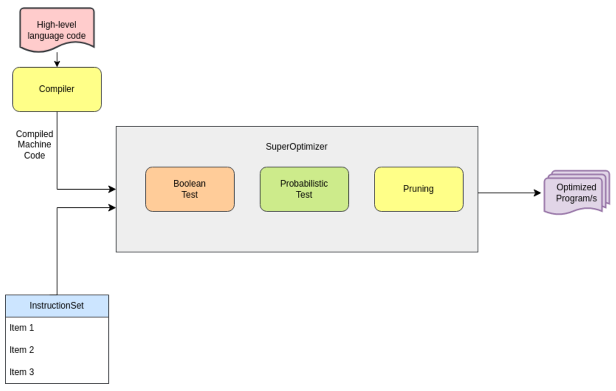
In such scenarios, the specialists have advanced knowledge of the Instruction architecture of the processor and the special-purpose processors. They decide on which part of the code shall be executed on the processor and which part would be redirected towards special purpose processors. The special purpose 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. In such optimizations, each main processor instruction is coupled with one co-processor instruction that is independent of the main processor instruction and are executed in parallel.

## 4 SUPEROPTIMIZATION

Superoptimization refers to finding the shortest algorithm for a specific function by dynamically analyzing the code generated at runtime. Its a run-time method and takes into account the environment where the program runs and the state of the program at the point of application. The search space is the complete instruction set of the given CPU architecture. The program component that performs this search and modifies the program accordingly is called a "SuperOptimizer". The next few sections focus on understanding the architecture, variants, application and limitations of such superoptimizers.

### 4.1 Understanding Superoptimizer

Superoptimizer as described in [5] uses three stages : 1) Boolean Test, 2) Probabilistic Test and 3) Pruning. Boolean Test simply performs a boolean verification if the given candidate in the search procedure returns the exact same output as expected from the original program. The test generally uses a binary bit-exactness tests to a certain pre-defined set of minterms to conclude the result. The other two stages are steps taken to reduce the search time. Probabilistic test is conducted using a set of test vectors that maximizes the probability that an incorrect program fails thereby reducing the size of the search space. Pruning helps further improve search time by filtering out instructions that are obviously redundant.



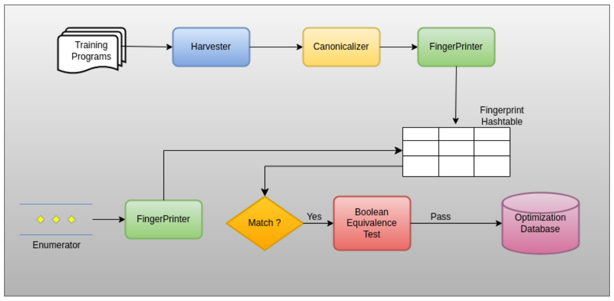
#### 4.1.1 *Limitations and Applications.*

The standard Superoptimizer described above has an obvious limitation that the searched program is limited to the specific architecture. Moreover, in spite of optimizations through probabilistic test and pruning, the search time can be impractical. Another added limitation is that the method does not describe any technique to handle pointers and pointer arithmetics which is a significant part of any system code including kernel.

Due to these limitations, such a superoptimizer is used in register-based operations, design of RISC architectures and also optimizing little tasks that do not use pointers extensively. Effectively, it can be used as a tool to help assembly language programmers in optimizing the programs and also along with other better superoptimizers [2] as a preparatory step.

### 4.2 Peephole Superoptimizers

The diagram below (derived from [2]) shows the components of a Peephole superoptimizer.



A peephole superoptimizer is basically intend to replace a sequence of instructions with an equivalent faster sequence of instructions. One proposal [2] uses automatically generated set of pattern-matching rules from a database of thousands to millions of optimizations. The goal with this superoptimizer is to achieve optimizations that are not explored by the compilers and possibly also not found by human optimizers.

The optimizations are computed offline and fed into a search database which can be accessed over a network instead of storing locally - referred to as network-based search engine. The type of instructions considered include loop-free assembly sequence.

The system to achieve peephole superoptimization has several stages briefly described below:

#### 4.2.1 *Harvester.*

This stage considers several reprensentative applications identifying harvestable instruction sequence as target sequences. The choice of the target instruction sequences is based on two constraints - 1) The sequence must have a single entry point and 2) After the first instruction, no other instruction in the sequence should be a jump target (ensuring the 1st condition).

#### 4.2.2 *Canonicalizer.*

In this stage, all instruction sequences that are equivalent to other sequences are filtered out to reduce the number of sequences under consideration. For example, A sequence is said to be *canonicalized* if registers and constants are named in the order of their appearance.

#### 4.2.3 *FingerPrinter.*

Further filtering is done in order to eliminate equivalence. The result of every instruction sequence in the target group is hashed and any instruction sequence that produces a result with a matching hash is eliminated as unnecessary. The hash is also called the *Fingerprint* of the instruction sequence. A hash table is constructed with hashes for all target instruction sequences. The *fingerprint* is an index into the vector of instruction sequences and is provides much faster access compared to instruction sequence search.

*4.2.4 Enumerator.* A few additional restrictions when applied to the target instruction sequence group, further reduces the search space. These restricted set of instructions are said to be enumerable instruction sequences. An enumerable instruction set is a sequence that has at most two exit points allowing for at most one conditional instruction in a sequence. The number of distinct registers is restricted to 4 since most instruction sets under 8 instructions per set use less than 4 registers in practice. The number of constants per instruction set as a common practice is reduced to 2. The number of indirect memory access is also reduced by the constraint on the number of registers used. Also the accessible memory is sandboxed and every instruction set is provided with a suitable offset that places all memory accesses within the sandbox territory. These measures though restrictive produces highly efficient superoptimizations and produce quicker results. Such enumeration steps provide exponential performance benefits.

*4.2.5 Boolean Equivalence Test.* After finding a suitable enumerated instruction sequence from the fingerprint hash table, it is necessary to evaluate the correctness of the sequence. This is achieved through Boolean Equivalance Test. This tests is carried out in two steps:

- **Execution Test:** Run the instruction set over a set of test vectors and compare the results.
- **Boolean Equivalance Test:** This test further tests the equivalence accurately using a SAT solver taking into consideration the current the state of the system defined by the values in the registers and the current memory map.

*4.2.6 Optimization Database.* The optimizations discovered in the process can be stored into a database which is indexed by the original instruction sequence in its canonical form and the set of live registers. With this database, the steps to optimize a code sequence becomes simpler and involves steps like, harvesting a sequence, canonicalizing it and searching the database to find if an optimization exists. This can be a running database that gets updated if an optimization is not found and is discovered through the longer process. In summary, the database can act as an intermediate cache for optimizations.

## 5 SUPEROPTIMIZERS IN MODERN COMPILERS

The GNU compiler provides a simple Command Line tool that can produce superoptimized code from a set of assembly instructions on a specific architecture. The tool supports several architectures and acts as a helping aid to developers and professional optimizers as an initial step before they carry out more elaborate optimizations. The documentation for this tool is available at [12] and can be experimented on any linux distribution.

SPARC, MC68000, MC68020, M88000, POWER, POWERPC, AM29K, I386, I960 (for i960 1.0), I960B (for I960B 1.1), PYR, ALPHA, HPPA, SH are supported by this tool.

An example usage is depicted below:

```
$ make CPU=-D<cpu_name_from_list> superopt
$ superopt -f <goal-function> | -all [-assembly]
 [-max-cost n] [-shifts] [-extracts] [-no-carry-insns]
  [-extra-cost n]
```

As we can see this tool is easy to use and universally available. However, the key limitation of superopt CLI tool is that its too slow. The time complexity of the underlying algorithm is exponential with respect to the instructions on the architecture. Some instructions sets with just 7 instructions might take a week to produce the optimization result. Hence this is still not practical for many use-cases.

Here is an example of working with GNU Superopt. The example generates 6 possible sequences of assmembly instructions for i386 architecture to execute the ABS instruction.

```
$ superopt-i386 -fabs -as
Searching for { r = (signed_word) v0 < 0 ? -v0 : v0; }
Superoptimizing at cost 1 2 3 4

1: movl %eax,%edx
sarl $31,%edx
addl %edx,%eax
xorl %eax,%edx

2: movl %eax,%edx
sarl $31,%edx
addl %edx,%eax
xorl %edx,%eax

3: movl %eax,%edx
sarl $31,%edx
xorl %edx,%eax
subl %edx,%eax

4: movl %eax,%edx
sarl $31,%eax
addl %eax,%edx
xorl %eax,%edx

5: movl %eax,%edx
sarl $31,%eax
addl %eax,%edx
xorl %edx,%eax

6: movl %eax,%edx
sarl $31,%eax
xorl %eax,%edx
subl %eax,%edx
[6 sequences found]
```

The max-cost paramater can be set to convey the maximum number of instructions the optimizer can use limiting the number sequences to those that can be executed with this cost. For example, reducing the max-cost for the ABS example above to 3 leads to failure since the superoptimizer cannot find any such sequences.

```
$ superopt-i386 -fabs -as -max-cost 3
Searching for { r = (signed_word) v0 < 0 ? -v0 : v0; }
Superoptimizing at cost 1 2 3 failure.
```

## 5.1 A Synthesizing Superoptimizer

Another effort worth mentioning regarding superoptimizations in modern compilers is described in [10]. [10] along with the source at *https://github.com/google/souper* presents a synthesizing superoptimizer that generates optimizations using SMT solvers which generalizes the binary SAT solver to consider real numbers, integers and other data structures including arrays, bit-vectors and more ([11]). The superoptimizer was called Souper and was able to work with LLVM and Microsoft Visual C++ compilers.

TODO: A PRACTICAL SOUPER EXAMPLE HERE

## 5.2 Superoptimization of WebAssembly Bytecode

## 6 OTHER RELATED WORKS AND FUTURE DIRECTION

In this paper, the initial works of Massallin [5], peephole superoptimizers [2] and the implementations in currenProjectt GNU compilers are presented. In addition to this [8] presents a goal-directed superoptimizer that uses an automatic-theorem prover to generate optimal code for several architectures. The general problem with the superoptimizers has been that the programs generated are relevant to specific processor architectures. Traditional techniques lacked scalability and using them on large programs pose several practical challenges. However, owing to improved computing with the advent of cloud computing and evolution of several statistical techniques including machine learning, have recently provided opportunities to look into superoptimization of programs. Some areas where more research has been carried out and future directions are present here.

## 6.1 Applications and Use-cases

One area that has been explored with superoptimization is enhancing the use-cases where superoptimizers can be applied. Generally used as an aide to professional optimizers and developers, it can be extended if there is a way to apply across multiple architectures. The binary translation using superoptimizer as presented in [3] as a solution to translate optimized assembly from one architecture to another.

## 6.2 Scalability

These techniques allow for scalable superoptimization. The paper [7] presents an approach presents a search algorithm called LENS that increases the size of the code synthesizable by the superoptimizer. The problem at hand is to search for a program that is semantically equivalent to a given program, but faster according to a specific performance model that uses a set of test cases comprising of test-inputs and test-outputs. The paper [7] models as a graph search problem with each program state represented as a node. The starting node is a set of instructions and the goal node is the optimized equivalent or the target program.

## 6.3 Statistical Approaches

[4]

## 6.4 Machine Learning Approaches

Over the last decade, machine learning and deep learning techniques have made unbelivable advances and have seen use-cases in various fields. It is only but natural, that the domain of systems optimization cannot be left out. A large number of really good code easily accessible from open source repositories creates a huge data set which the authors of [1] call **Big Assembly**. [1] describes an approach called SILO (Self Imitation Learning Optimization) that is trained on a subset of the Big Assembly and progressively improve superoptimization ability with this training.

[1]

## 7 CONCLUSIONS

Synthesis kernel introduced the concept of runtime code generation and superoptimization to enhance the kernel performance. Some of these concepts are relevant even to this day when applied with the modern approaches and new use-cases can be discovered.

## REFERENCES

[1] J.Lacomis C.Le Goues E.Schwartz G.Neubig A.Shypula, P.Yin. 2022. Learning to Superoptimize Real-World Programs. (2022).

[2] Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. (2006).

[3] Sorav Bansal and Alex Aiken. 2008. Binary Translation Using Peephole Superoptimizer. (2008).

[4] Alex Aiken Eric Schkufza, Rahul Sharma. 2016. Stochastic Superoptimizer. (2016).

[5] Henry Massalin. 1987. Superoptimization - a look at the smallest program. (1987).

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

[7] R.Bodik D.Dhurjati P.M.Phothilimthana, A.Thakur. 2016. Scaling up Superoptimization. (2016).

[8] Greg Nelson Rajeev Joshi and Keith H. Randall. 2001. Denali: A Goal-Directed Superoptimizer. (2001). http://www.research.compaq.com/SRC

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

[10] J.Ketema G.Lup J Taneja J.Regehr R.Sasnauskas Y.Chen, P.Collingbourne. 2018. A Synthesizing Superoptimizer. (2018).

[11] Adapted from ACM SIGDA newsletter Prof. Karem Sakallah Various Authors. 2006. *Satisfiability modulo theories.* https://en.wikipedia.org/wiki/Satisfiability_modulo_theories

[12] Opensource Maintainer Thien-Thi Nguyen Various Authors. 1995. *GNU Superoptimizer.* https://blog.regehr.org/archives/1676