International Journal of Research in Engineering, Science and Management
Volume-2, Issue-5, May-2019
www.ijresm.com | ISSN (Online): 2581-5792


# A Review of Conventional and Upcoming Approaches for Compiler Analysis and Code Optimization

M. Shobha[1], Soumyashree Dhabade[2], C. Sowmya[3], Sini Anna Alex[4]
[1,2,3]*Student, Dept. of Computer Science and Engineering, Ramaiah Institute of Technology, Bengaluru, India*
[4]*Assistant Professor, Dept. of Computer Science and Engg., Ramaiah Institute of Technology, Bengaluru, India*

***Abstract*: Present day compilers have profusion of code modification. But all most all compiler follow same old method to optimize code and they apply predetermined sequence like scanning, lexing, analyzing syntax and analyzing semantics followed by generating intermediary code and finally generating target code. Thus optimizing functions without interpreting whether code is getting modified or not we can't assure that after compilation it uses less resources and faster execution. In order to overcome this problem, techniques like Reverse-Inlining (Procedural Abstraction), Cross Linking optimization, trace inlining, Optimizing Leaf Function, Combined code motion and allocation of registers etc are used which are more efficient and generated better machine codes. As trace inlining technique helps in analyzing enforcement and size of low level code generated. Various inlining rules on the benchmark suites DaCapo 9.12 Bach, SPECjbb2005, and SPECjvm2008 are assessed, proved that compilers based on tracing attains nearly 51% better enforcement than method-based Java HotSpot client compiler. Moreover, the positive effect of the large compiler scope on other optimizations of compiler is conveyed. One of the techniques used to deicide good optimizing sequence for programs is artificial neural network.**

***Keywords*: Compiler Analysis**

## 1. Introduction

Process of transforming a program where it considers resultant of foregoing step as input to next step and modifies code in such manner that it absorbs minimum expedients like CPU, memory etc. with superior accomplishment is called optimizing code. Machine dependent and machine independent are kinds of code optimizations. Transforming given code as stated in target device architecture, modifying induced code is called as Machine dependent optimization. It uses CPU and registers, instead of using corresponding memory label it uses established memory label. Machine independent optimization [1] method optimizes transitional code for better final code.

This paper focuses on new techniques which supports for the emerging computer architectural designs. Very large instruction word (VLIW) Architecture, allows exploiting instruction level parallelism, meaning execution of more than one instruction simultaneously which are not inter dependent. In Parallel execution code execution doesn't consume more time and hence code will be more efficient. So in this paper we will also discuss about the trending techniques of compilation.

Traditional compilers have been in use for generating machine codes until the emerging architectural changes, since these were hardware dependent thus there was a necessity to build compilers that are not hardware dependent and use to compile source codes for type of microprocessor designed to control functionality of devices has accentuated coders to need for controlled potency consumption, real-time execution and agility of code.

In static compilation, compiler processes the program code method after method, a control-flow graph (CFG) is constructed for each method, the graph is optimized and the native code gets generated based on CFG traversal. Suganuma et al, proposed a just-in-time dynamic compiler(JIT), where it accesses the runtime profile information, selects code only if it influences the overall runtime and inline those parts of method bodies only [2]. Gal proposed trace-based compilation approach to building dynamic compilers, wherein cyclic code paths that are frequently executed are identified and recorded. These traces are assembled dynamically into a tree-like data-structure. The major advantage is that the trace tree only consists of code areas that are relevant. Edges which appear in static CFG but that are not executed at runtime, are not considered in the trace representation, and are assigned to an interpreter in case of execution. The control flow merge point in the trace tree being absent greatly simplifies optimization algorithms which increases performance and reduces the machine code that gets generated.

We will see the difference between the traditional compilers and modern compilers in their way of optimizations and compiler analysis, also the modifications done to the conventional compilers in order to meet the emerging changes.

## 2. Literature survey

- Phase Ordering optimization techniques: Here instead of following fixed sequence for modifying whole program, apply it to discrete portion by this it selects superior sequence for modification automatically.
- Phase ordering with genetic algorithm [3]: It is used in

two experiments. First experiment involves process of finding better optimization sequence. To do this evaluate optimization sequence (chromosome) through compiling available benchmarks with sequence, recording their time taken for execution, calculating their fastness by normalizing time taken for running with running time observed by compiling benchmarks at O3 level. This corresponds to "best overall sequence". Second experiment involves finding superior optimization ordering refers to "best sequence per benchmark"

- Automatic Feature generation [4]: Components of system are training data generation, attribute generation and machine learning. Extracting intermediate codes of compiler and optimal values for heuristic is task of data generation. This does process iteratively to obtain best heuristic value for each program. The extracted codes will be passed to feature search which explores characteristics of intermediate codes and generates vector by considering evaluated values of all programs, pass generated vector to machine learning tool. Job of tool is to fetch characteristics from base attribute list and calculate value for that characteristic and add characteristic with best heuristic value to list, if two characteristics have same quality or value then add short characteristic to the list. And this process will be repeated iteratively. List gets generated in final is called as latest attribute list.
- Trace Inlining: In previous work, found on Oracle's JavaTMHotSpot client compiler a trace-based JIT compiler [5] was implemented. It focused on how trace inlining is performed and its advantages. Application of several trace inlining rules was presented for trace-based JIT compiler. Influence of rules on peak enforcement, size of generated low level code and compilation time for DaCapo 9.12 Bach benchmark suite was evaluated.

## 3. Discussions

Traditional compilers aim to remove redundancies, inefficiency and reordering the data and operations. It uses a number of techniques for this purpose like data flow analysis where we try to understand the flow of the data. If a variable is defined we try finding out the way it‘s used. It can be either forward analysis or backward analysis, in forward analysis we supply the value of variable to succeeding code whereas, in backward analysis we can supply information about some succeeding properties "back in time", like in case of dead code elimination we can withdraw variables if they are never read in future.

Local optimization is used to improve the code and done with the following methods.,

- Local constant folding where Expressions with operands holding fixed values can be evaluated at time of compilation and this causes faster execution, reduction in size of code.
- Local constant circulation, fixed values allocated to a variable can be circulated through the flow graph and replaced at use of the variable.
- Local sub-expression elimination, while executing the expressions present in the program variable being accessed in current step whose value is being computed in previous step and if it's not modified in such scenarios compiler can skip process of recounting value for the same variable.
- Local strength reduction is frequently applied in customary compiler optimization to allow replacement of an expression by its meaning. That is, it replaces more resource consuming operations (multiplication) by less resource consuming operation (addition).

Global optimization, different from local optimization is applied on global parameters. Data flow analysis is used for code optimization during this step. The various methods followed are:

- Redundant (common) Sub-expression elimination, in this kind of elimination redundant expressions is identified globally and are calculated once and replaced by the result everywhere else.
- Dead Code Elimination: After all the steps are performed as mentioned so far, the optimized code consists of some used lines of instructions which are eliminated in this step.

Loop Optimization [6], if each time the result computed within a loop is same then computing it in each iteration is not required. If an induction variable is found within a loop that is a variable whose value on each loop iteration is proportional to the calculation of the iteration index. When such variables and the value they compute are found, often high cost operations are replaced by low cost operations or the variable itself can be deleted.

Reverse-Inlining (Procedural Abstraction) is used to achieve code size reduction. This method replaces the function call with function definition, thus increasing the speed of execution.

Some of the modern compiler optimization and compiler analysis are discussed below. It also uses data flow analysis but with alias analysis that is it is used to find if data storage can be retrieved in many other ways. If two pointers are pointing to the same memory location they are said to aliased. Advanced compiler analysis applies the above mentioned aliasing in determining the existence of the references at the exit of the block.

- Combined code motion [7] and register allocation, Code motion tries in placing the such instruction together than are independent of each other and can be ran simultaneously, thus implementing parallel execution. The Register Allocation and Code Motion (RACM)minimizes register pressure by implementing code motion as explained above, followed by duplicating the code and finally storing the value of variables from register to memory.
- Cross linking optimization, this optimization can be applied globally or locally. This is used where functions

consist of switch statements with same tail codes, which is replaced after the switch block.

- Address Code Optimization, speeds up instructions and execution time by reordering data in memory to minimize and simplify computation of address.
- Type Conversion Optimization, involving the physical size of the data known as data processing, and semantics of data processing operations. Substructure for a different of data types is the main goal of most compilers; as such compilers insert many implicit data type conversions, such as zero extension.
- Multiple Memory Allocation(MMA) is one of the technique used in trending compilers as in this technique instructions are saved and uploaded in many registers simultaneously.
- Upcoming compilers are also expected to minimize code length, increase programs speed to run by accurate usage of memory which is appreciably seen in the techniques they adopt.

Artificial neural networks [8] used to find out superior sequence modification, represented as reticular nervous. Kinds of feed forward network are one layer with one insert and one resultant layer. Multilayer with one insert and one resultant layer and includes multiple underlying layers. Recurrent layer is like multilayer but includes minimum one feedback loop. As mentioned before resultant of each layer becomes input to next layer, link across neurons represented by weighted vectors. ANN works in iterative manner where for each iteration it takes characteristic of code of present state and does evaluation to determine superior optimization.

Enactment of model is done by using a dynamic compiler 4cast-xl that builds ANN, integrates it to Jikes RVM's optimization driver and performs phase ordering optimizations. And this dynamic compilation has to be done repeatedly by ensuring the steps given below:

1. Generating attribute vector of current method's state.
2. Initiating outline of code.
3. Determine superior sequence to modify by using ANN.

Following is an overview on brief description of VM's structure. Class loader does following functions of parsing, loading and verifying class files and run-time data structures are also provided to different parts of VM. Then during bytecode pre-processing step, loops are detected and tracing-specific data structures are created. To implement trace recording quite a few steps are followed by Java HotSpot VM template interpreter that is key is duplicated (a copy made) and instrumented which results in a usual and a trace recording interpreter. With respect to the initial executions, normal interpreter is used. Counter of invocation of that trace anchor increments each time basic interpreter comes across a trace anchor. Execution switches to trace recording interpreter, when counter exceeds specified threshold. Trace-based JIT (just in time) compiler is based on HotSpot client compiler. Compiler records traces into a trace graph when traces have been recorded often enough. The interpreters and other compiled traces then directly invoke generated machine code. If a prerequisite condition for optimization is violated, System reverses optimizations to trace recording interpreter. During reversing the optimizations all values that are still in current compiled frame are saved first and then one or more interpreter frames replaces that compiled frame and then execution is continued in trace recording interpreter. The trace recording interpreter takes charge here after as, a partial trace that starts at the point of reverse optimization, instead of from where the trace anchor is recorded.

In trace recording approach, every individual thread maintains tracing stack for traces that gets recorded. Instructions that can alter control flow is recorded always at stack's top. Method invocation also gets notified in caller's trace. Receiver's class also gets stored if it is a virtual method. A new trace for caller function is recorded onto stack and recording of the caller is resumed. On completion of caller's execution, trace of called function gets popped and stored in a trace repository. A pointer to caller's function trace is used to link caller and called function by recording in caller's trace and then the recording of called function is continued. Information which is context dependent over the methods gets preserved by this linkage. Trace doesn't get stored but counter gets incremented in the previously recorded trace, when a trace that is saved is again recorded. Traces are considered to be different depending upon the execution flow or the caller function traces. Hence, accurate information of call regarding every executed path is recorded using trace-linking. Loop and recursive modules are not linked to their parent trace to reduce the number of traces that gets recorded. Assuming that all the traces for a particular anchor has been recorded after recording is done a specific count of times for that anchor, machine understandable instructions which is optimized is generated from those compiled traces. Commonly executed operations are executed straightly by the interpreter's assembler and complex operations are executed in C language-based runtime environment. Efficient operations done by many threads simultaneously enables Java thread structure to jump between two interpreters independently. A local buffer is maintained by each thread to achieve the best performance. The data structure used to store recorded traces allows data to be read and denies locks and atomic instructions during the reading process. But locks data structure during writing threads only when a new trace is found.

In trace inlining, static and dynamic inlining methods [9] are supported by trace-based JIT compilers with the help of the recorded trace information. Trace inlining replaces function calls with its code itself but traces that are commonly executed are only inlined instead of entire methods. Unlike method-based compilers, context dependent information is contained in the recorded traces which helps to avoid inlining of function definition parts that are unnecessary for the current caller but are executed frequently overall. Aggressive inlining of virtual

methods takes place since methods of only a specific type of receiver gets invoked at a call site.

First of all, the maximum size of the trace, which is dependent on the relevance of the call site, that has to be placed in inline at the current call site is computed. Then, the invoked traces at the current call site areinlined based on certain heuristics. So, the method invocation is replaced with the trace graph that is built from the traces that should be inlined. Then, return instructions are replaced with jumps to the next line of command after the call and exception-throwing instructions will be attached tohandlers of exception. Usually, it's only the traces that get invoked by the current caller that are inlined. However, in certain cases, all caller function traces are considered as candidates for inlining, if the callee traces were compiled before the caller's trace recording had even begun. Based on dead code elimination technique [10], certain candidates for inlining can be rejected based on specific parameters that is passed to the callee by the caller.

By taking into consideration CHA [11] and context dependent trace information, in case of virtual calls, the actual receiver class for the calling site can be determined. As recorded receiver classes are used if multiple target methods are found by CHA along with an additional run-time check such that the trace deoptimizes to the interpreter in case of type mismatch. In a similar manner, since no context-specific information is available for the loop traces, all recorded traces are considered as candidates for inlining. However, elimination of few traces is possible based on parameters and locals. Traces with no further lead existence in the caller functions trace graph for the inlined loop is also eliminated.

A special kind of inlining guarded by method guard was implemented to overcome the issue that a method could not be inlined if the same method was always invoked by the call site but by different receiver types. In the above process the invoked method is compared with the expected method by accessing the virtual table of the receiver. Multiple receiver types can be checked for by extending the type guard to a switch structure if the same method was always invoked by the call site but by receiver of type interface. The stated process is definitely cheaper when compared to the interface lookup. The switch like structure can also be extended to efficient inlining of polymorphic calls. Platform specific methods are inlined by the JIT compiler using heuristics of the compiler. The same method based inlining is conducted yet the trace-based compiler performs aggressive inlining of java traces and traces are also smaller than methods.

Further departing edges of similar part of the code is compared with the repeatedly implemented departing edge determined. Edges to unreachable parts of code and with hundred times the minimal frequency are removed from the trace graph. The process has many downfalls and must be taken care of, the tracing information is based on the program behavior that may change over time, removing major execution paths might proceed to repeated deoptimizations. In the case of loops, the exit will have to be filtered out since in spite of loop body, occurrences of loop entry are compared with occurrences of loop exits, which would restrict the scope of compilation and again rise the occurrences of deoptimization.

In order to discuss about trace inlining heuristics, execution frequency of each block of trace graph, using recorded traces, is evaluated. Then, estimated frequency is then divided with a reference value to estimate relevance of each block which in turn will be used to compute the call site's relevance. So, that reference value is chosen utilizing one of the algorithms mentioned below.

- Simple: Estimated execution frequency of each block of trace graph is divided by completeimplementation occurrences of complete merged execution paths of graph. Value lies in domain 0 to 1, excluding 0.
- Most frequent trace: Execution frequency of each block is divided by execution frequency of merged trace that gets executed most number of times. So, call sites residing in blocks shared among multiple traces would have a greater relevance due to higher execution frequency, while values in range ]0,1] would be the relevance of a call site residing only in individual traces.
- Path-based: In this approach, firstly successor block, with respect to root block, that is executed most number of times is determined. Then, marking block as visited process is performed over and over until either a loop header orleaf nodes is reached. Then, leastimplemented node of all visited nodes will be used to determine the other blocks aptness that are present in trace graph. Hence, values would be in range of [1,∞], for call sites deemed to be important, while [0, 1] would be range of values for calls that are less important.

Following are the configurations of the dynamic inlining heuristics that generated least size of machine code and increased performance.

- *Minimum code:* Depending on relevance of call site, inlining size of 35 bytecodes is modified by this heuristic. Inlining size is decreased for relevance less than 1 and increased for relevance greater than 1. This heuristic results insmall size of machine code and increased performance on combination with path based algorithm.
- *Balanced:* Inlining size remains unchanged for blocks with relevance value less than 1 which conceivably are important calls that are inlined, while inlining size is increased by 40 bytes of code by this heuristic for blocks having relevance greater than 1. This heuristic results in stability between size of code and performance on combination with path based algorithm.
- *Performance:* Depending on the relevance of call site, substantial size of 150 bytecodes is used for inlining by this heuristic. The inlining size is not increased beyond the increased value. Inlining size is decreased for relevance less than 1. This heuristic optimises the performance on combination with path based algorithm.

- *Greedy:* Similarly, large inlining size of 250 bytecodes is used by this heuristic. This heuristic prevents inlining of traces beyond maximum value. The inlining size is decreased if relevance is less than 1. This heuristic results optimizes the performance on combination with path based algorithm. This heuristic on combination with algorithm that computes the frequent executed paths provided better results.

Amount of machine code generated is more on invoking small traces in comparison to inlining them hence the above heuristics make sure that the assessors are always inlined. If compilation of calleee merged to substantial size of machine code then inlining is avoided by the heuristics since after a certain point increasing scope of compilation is discouraged. Inheriting of relevance of the parent block by the callee is used by our heuristics to minimize size of machine code and inlining of traces in a nested manner.

To evaluate JIT compiler deployed from traces, it is enforced for Java VM of A-32 design from Oracle [20]. Benchmark suites DaCapo 9.12 Bach, SPECjvm2008, andSPECjbb2005 were chosen for evaluating the above discussed heuristics. Results evaluated is with respective to results for client compiler based upon the methods.

- Server/client approach is simulated by this SPECjbb2005 benchmark where functions are implemented in database of memory that is subdivided into warehouses. Relative to peak performance, client compiler gets outperformed by trace-based configurations because of greater aggressiveness of inlining of traces. Configuration greedy slightly causes an increase in performance but at same time size of machine code is more. Configuration, minimum code, is well organized with respect to machine code and time utilized for compilation and also performance is decent. Irrespective of number of warehouses used client compiler is outperformed by trace configurations. Performance is utmost when there are 4 warehouses since benchmarking system has 4 cores and each thread processes one warehouse respectively.
- Nine benchmark categories are included in SPECjvm2008 benchmark to measure peak performance. Client compiler gets outperformed by tracing configurations. Highest speed-ups is witnessed on derby and serial benchmarks by tracing configurations. Due to small size, benchmarks scimark, mpeg audio and crypto indicate nearly no increase in performance and is similar to client compiler. However, it's only traces that gets inlined and not whole methods which decreases compilation time and machine code that gets generated.
- Fourteen Java applications is included in DaCapo 9.12 Bach benchmark suite. On the whole, less machine code gets generated with respect to all inlining heuristics except greedy configuration, yet overall performance increases. Greedy configuration profits benchmarks luindex, pmd, and sunflow, due to its large inlining size. Highest speed up is achieved by tracing configurations for jython benchmark, which is responsible for execution of virtual calls, since compiler makes use of recorded trace configurations. In terms of compilation time, configuration based on traces is well organised that even aggressive configuration greedy takes similar amount of time as client compiler.

Optimization is positively affected by inlining of traces due to increase inscope of compilation. For SPECjbb2005 benchmark, canonicalization [12] is increased because of trace inlining. For SPECjvm2008 benchmark suite, optimizations is hardly affected due to unavailability of larger compilation scope. Increased performance is equally spread over all optimizations listed on benchmark DaCapo 9.12 Bach.

Code with effective performance and optimizations are produced by server compiler but involve 13times extended compilation on benchmark suites DaCapo 9.12 Bach, SPECjbb2005 and SPECjvm2008. Trace-based compiler only attains up to 67% of performance of server compiler in SPECjbb2005 benchmark but benefits largely from server compiler optimizations. 85% of performance of server compiler is attained for SPECjvm2008 benchmark suite by configuration greedy. Server compiler exhibits higher performance on crypto, mpeg audio, and scimark benchmarks that are loop exhaustive, due to advanced optimizations. However, server based compiler gets outperformed by trace based compiler on compress and sunflow benchmarks, by aggressive trace inlining. In DaCapo 9.12 Bach benchmark suite, on an average 93% of performance of server compiler is attained by trace based compiler due to presence of benchmarks that are more complex and less loop exhaustive. However, irrespective of basic optimisations performed by trace based compiler, server compiler's performance gets outperformed in sunflow, pmd and lu index benchmarks due to context-sensitive and aggressive inlining of traces.

## 4. Conclusion

This paper discusses difference between the traditional compilers and modern compilers in their way of optimizations and compiler analysis, also the modifications done to the conventional compilers in order to meet the emerging changes. It also discusses the java compiler based on traces that manages inlining of traces during compilation of JIT instead of at times of trace recording, which enforces trace inlining to be more selective because of the extra information available. Besides, variant components of the method can be inlined based upon the site of call because of context-sensitive nature of traces. Moreover, its proposed that it's only the traces that are implemented often that gets compiled to machine code by eliminating infrequently executed traces. Evaluation with benchmark DaCapo 9.12 Bach, SPECjvm2008 and SPECjbb200proved that effective performance along with the generation of decent amount of machine code can be attained by proper trace inlining. Furthermore, larger compilation

scopes attained by trace compiler increases efficacy of compiler optimizations and effective performance.

## References

[1] David Padua,"Compilers and the Furture of High Performance Computing" , IEEE 22nd International Conference on High Performance Computing (HiPC), 2015.
[2] M. Bebenit, Jikes RVM, "Trace Based Compilation in Interpreter-less ...", Semantic Scholar, 2008.
[3] Michael R. Jantz; Prasad A. Kulkarni, "Exploiting phase inter-dependencies for faster iterative compiler optimization phase order searches", 2013.
[4] Edwin Bonilla; Michael O'Boyle, "Automatic Feature Generation for Machine Learning Based Optimizing Compilation Hugh Leather", International Symposium on Code Generation and Optimization,2009.
[5] Hiroshi Inoue; Hiroshige Hayashizaki; Peng Wu; Toshio Nakatani, "A trace based Java JIT compiler fitted from a method-based compiler," International Symposium on Code Generation and Optimization, 2011.
[6] Grigoris Dimitroulakos; Christakis Lezos; Konstantinos Masselos," MEMSCOPT: A source-to-source compiler for dynamic code analysis and loop transformations," Conference on Design and Architectures for Signal and Image Processing [LOOP], 2012.
[7] Slavko Radonić; MiodragĐukić; NenadČetić, "One solution of loop invariant code motion compiler optimization," 22nd Telecommunications Forum Telfor (TELFOR),2012.
[8] L. Prechelt, "Exploiting domain-specific properties: compiling parallel dynamic neural network algorithms into efficient code," IEEE Transactions on Parallel and Distributed Systems, 2010.
[9] J. Cavazos; M. F. P. O'Boyle, "Automatic Tuning of Inlining Heuristics", ACM/IEEE Conference on Supercomputing ", 2005.
[10] Hiral H. Karer; Purvi B. Soni, "Dead code Elimination Technique in eclipse compiler for Java", International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT), 2015.
[11] Jason Sawin; Atanas Rountev, "Assumption Hierarchy for a CHA Call Graph Construction Algorithm", IEEE 11th International Working Conference on Source Code Analysis and Manipulation, 2011.
[12] Shin-Ming Liu; R. Lo; F. Chow, "Loop induction variable canonicalization in parallelizing compilers," Conference on Parallel Architectures and Compilation Technique, 1996.