OPTIMIZING THE EXECUTING TIME OF COMPILER OPTIMIZATION USING CODE SEGMENTATION TECHNIQUE

Kaushik Roy¹, Brindha K², Sudha Senthil Kumar³

¹SITE School, VIT University Tamilnadu, India.
²Email: kroy.compsc@gmail.com

Received on 12-05-2016
Accepted on 05-06-2016

Abstract

Optimization compiler means that the compiler tries to minimize the execution time, memory usages, and code length and power requirement. But within that parameter the execution time is the most important parameter whereas the memory usage is almost negligible nowadays. After the popularity of portable computer systems like laptops, palmtops, Tablets etc. another major optimization parameter has become the power consumption. That means using minimum numbers of resources how the program can be successfully executed without alteration of the final outcome. Peephole optimization is widely used optimization technique still yet in the field of compiler optimization. It is the technique based on string pattern matching using regular expression and replacing a piece of code with the equivalent piece of code which is shorter and faster than the original one. However the efficiency of a compiler depends on the machine architecture, target application and compilation environment. So optimization technique is to get a way which could balance all those parameters.

Nowadays excluding the peephole optimization technique there are lot of other which is dedicated to get a balanced strategy among those parameters. So in this paper we have discussed about those methods with their pros and cons. As far in this field no technique is perfect for all situations, different strategies are healthy for different situations. We have also discussed those situations where a strategy, depending on parameter can give us a better result compare to others and also cannot be a good for any other parameters.

In this paper we have also proposed a new technique of parallel execution in multicore system. Basically the technique depends on the dependencies of instruction on each other and according to those dependencies we have segmented the code, so that they can be executed parallel without affecting the final outcome. The details is in our proposed algorithm section.

Keywords: Compiler optimization, Code Segmentation.
I. Introduction:

With the evolution of complex processors day by day optimizing a code becoming more critical. Compilers are no more required to generate simple matrices for instruction count. Nowadays optimizing means the balanced combination of utilization of execution resources, registers, pipeline to avoid unnecessary stalls and sometimes need to consider the cache misses and branch mis-predictions also. The optimization would be more effective when a compiler can predict the future pressure on resources, like register pressure, height dependence etc. and according to that can contemplate those parameters according to their availability on a specific machine architecture. Most of the compilers basically follow the “predictive heuristic” method to understand and apply these code transformations [1,2]. The heuristic method predicts whether a particular optimization technique would be beneficial or not for a code fragment. The full benefit can only be taken when it predicts all the chances of applying the optimizations on a code fragment. Basically it is a complex and also expensive to predict most of the time which leads it to less popularity among the compiler designers. Also for the modern architectures most of the parts of a code remains unrealized which could give us more significant potential. To get a solution of this many researchers proposed to compile a same code multiple times with different optimization configuration. This can give us a brief view about the shortcomings of predictive heuristic optimization and explore much promises about this iterative compilation method [3,4,5]. But this process follows basically the exhaustive search technique via full execution which leads long execution time. So this technique is not suitable for most of the practice environment [6].

The optimization can be realized into three stages (a) Frontend: Optimizing the source code which is basically depends on a programmer. Most of the cases a well formed optimized code executes faster than an un-optimized code with the same logic. (b) Intermediate code: According to a specific programming language syntax compiler optimizes the assembly level code according to machine architecture. (c) Backend: Optimizing the object code or machine specific code during execution which is extracted from the assembly language. Most of the time frontend optimization techniques are dependent on the programming language and the experience level of a programmer on that specific language.

But for most of the programming language there are some well-defined frontend optimization techniques using which it lead us not only optimizing the front end but also it help the compilers to predict a better optimized assemble code and when your assemble code is itself optimized the there is also a huge chance to generate optimized machine level codes. After all machine level code depends on the machine architecture and when the architecture of a machine is in
optimal in use then it could be easily realized that the code has a chance to become the optimized one a lot. At the same time according to the functionality a compiler can be considered a combination of front end and back end. In front end it translates the source code to intermediate code and the back end translates the intermediate code to assembly language according to the target machine.

This model is helpful because adding a new language or machine needs writing one new program (either for front end or back end) to handle the characteristic that all language are available on all machine. Now optimizing the front end means many common optimization need to be done for each front end and the optimizing back end means the duplication of code for different machine again and again. So basically the most convenient choice for optimization is to exploit the intermediate part. Peephole technique is the oldest technique to a optimized code from assembly level code where the inefficient instructions are replaced in target machine code to optimize the response time, memory usage, resource use.

We also discussed about the previously existing peephole techniques in our related work section. Nowadays the machines are with multicore or multi-processor architecture, but still the compliers can exploit the only one processor. But for a systematic shared memory multiprocessor the compiler can parallelizes the code according to their data locality, parallelism and the granularity of parallelism. To achieve the parallelism it uses simple cache model and dependence analysis to derive the loop optimizes: loop permutation, tiling, fusion and distribution [7]. In this paper we also consider this part to optimize. The execution time can be much more reduced if we could make involved the multicores as much as possible. We are proposing to segment the code in different parts according to the dependencies among them and execute them parallel in multiple cores. In the Experimental Result part we also have discussed some cases so that the efficiency can be realized. To realize those segments first it is better to understand the existing optimization techniques in front end as our algorithm depends on the front end most and the efficiency of the code also will be achieved according to the compact coding.

I.A. Specific existing optimizations in front end:

A.1. Address optimization: In some architecture, referencing a global variable by constant address requires two instructions. Now this leads to expensive load. But using a pointer it can be accessible also which reduces the code size also. So in a program if there is a less no of global variable is unable to know during compilation some times. So the compiler and link should be cooperating with each other. If there is no register defined for global variable pool, then loading a pointer would no longer a cheaper strategy the directly global addressing.
A.2. Bit Field Optimization: Most of the architecture supports a series of load/store instructions to storing bit fields. But keeping those bit field in the registers, the runtime performance may be enhanced.

A.3. Loop Collapsing: Sometimes some nested loops can be replaced by a single nested loop with the help of pointer. As, we all know that pointer facility is only available in C. However if we can collapse the multiple nested loops into single one than the run time performance can be improved a lot and this leads us to loop unrolling technique.

A.4. Instruction Combining: Sometimes combining some statement in source code level can improve the intermediate code in terms of space. But this type of instruction combination is safe for basic block only.

A.5. Constant Folding: Constant Folding means the expressions consist with constants are evaluated during compile time and replaced by compilers. Basically it reduces code size and also evaluation time during runtime. But this type of constant folding is only possible for integer values, because before runtime the rounding mode is most of the time unknown to the compiler. As it is very simple optimization technique, so most of the compilers follow this technique without any thread.

A.6. Constant Propagation: If any constant is assigned to a variable which remains unchanged during runtime, then all the other user variables which are depending on the unchanged variable can be evaluated up to some extend during compiler technique. That means the assigned constant propagates through flow graph and can be used instead of that variable which is assigned by that constant. Some compiler follows this for integer constants only, also within a basic block. The constant propagation also is handled by pointer assignments for some compilers.

A.7. Common Sub-expression Elimination: If an expression is already computed and the value of the operands are still unchanged then the re-computation of that expression will be avoided. If the previous expression is a sub-expression for the current expression, still it replaces the previously computed value for that sub-expression. Through theoretically this CSE elimination is possible for operators, data types and storage classes still in reality only few compilers take this risk within basic block.

A.8. Dead Code Elimination: Sometimes the elimination of unreachable code is useful in terms of code size and storage complexity.

A.9. Forward Storage: Accessing or storing a global variable within a iterative loop can devastate memory bandwidth a lot. Using a temporary variable we can keep out that global variable out of loop. Now that temporary variable would be better if it is a register variable, otherwise same wastage of memory cycle would arise. For an example:
A.10. Loop fusion: Sometimes to improve runtime performance it is better to fuse more than one loop into one, if it is possible. Though loop fusion reduces loop overhead, it does not always improve runtime performance.

A.11. Hosting: The expression which is not directly related to a loop can be hoisted out from the loop. At least it is better to execute once then executing again and again for each iteration.

A.12. If optimization: Sometimes more than one adjacent or nested id statements with same conditions can be combined to gather, it optimizes the condition checking overhead again and again for the same condition.

A.13. Integer Mod/Multiply Optimization: Direct integer division function is expensive for most of the architecture. So any modules expression whose divisor is in power of two can be replaced with condition and shift instruction and this improves run time performance. EX- ‘x%8’ can be represented as ‘(x<0)? (temp==0)? 0 :( temp|~7)) : temp’ here temp= ‘x&7’. In a same way the multiplication function can be replaced with shift for a multiplicand which is in power of two. Ex- ‘j*4’ replaced by ‘j<<2’

A.14. Tail Recursion: Sometimes avoiding trail recursion through go to statement can avoid the overhead of call and return the function again and again also avoids the unnecessary use of stack.

II. Related Works

From the beginning of the compiler, the optimization was the most matter of concern among researchers. This journey of optimization continues more than fifty years. We could segment the in following order

II.A. Machine Specific Peephole Optimizers:

First Peephole optimization is invented by William McKeeman in 1952. In this technique at a time two assembly code instructions are considered as a small window. The two phases in this technique are matching and replacing. Simple hand written rules were there and according to that the assembly instructions were matched and if there was any matched redundant code than it was replaced with the redundant one[9].
In 1981 Lamb introduces optimizer with three phases: pattern rules, a translator and pattern-matcher frame work. The assembly level code is generated in a double linked list formation for easy operating. The optimizer uses compiled version of the patterns and it is simplified again for matching the rules. It also contains abstract symbols or variables that are epitomized during pattern matching. Escape is defined as a condition which must met over some variable(s) occurring in the pattern.

Pattern Matching can only be successful when all the instructions in the preconditions part can be connected to contiguous code instructions with any specific conditions over variables get to be valid. It starts with last instruction and backwardly it reaches to the first one through rule matching. The major advantage is that the newly inserted rules in precondition part can be instantly performed[10].

II.B. Machine Independent Peephole Optimizations:

The previous peephole techniques were machine dependent, therefore they can be only used on a specific machine. So if the machine changes it needs to change the code whole again. This re-targetable optimizes can be used for different machines, as they are independent in nature though they are little bit slower than their counter part.

Jack W. Davidson and Christopher W. Fraser developed their own first re-targetable peephole optimizer, known as PO in the year 1979 and 1980 respectively [11, 12]. PO used to receive two inputs – one the assembly code and another one is the machine description. The effect of the machine generated assembly code are represented as register transfer patterns. Then a bi-directional translation grammar is made between assembly code instructions and register transfers. Now each pair of assembly code is analyzed and then converted to equivalent register transfer patterns using the lastly generated bi-directional grammar. This register transfer patterns are optimized by PO. At the end the best possible single assembly code is replaced with the input assembly code. Instead of operating on the input assembly code, PO operates on register transfer which leads most of the possible optimizations without exhaustive searching. Late 1980’s a lot of compiler follows this technique or heavily inspired by PO and one of the still well know is the GNU C compiler GCC [13, 14, 15, 16].

Tanenbaum [17] introduced a little bit different approach where optimization focused on the intermediate code, but not on the object code. Basically this one is independent from front end and back end phase of different compilers. This technique facilitated the portability of a compiler. This technique used to follow some simple hand written pattern matching rules. The authors also claimed that this one is the faster than the Davidson and Fraser model. But major disadvantage of this model is that in some cases optimization is not possible before code generation.
Davidson and Fraser later focused on automating the development of re-targetable optimizers where OP was used during compile time to generate a ‘training set’ of patterns which would help to find patterns for optimization automatically[18,19].

Lamb’s HOP[20] technique followed the hashing to perform matching and replacement of instructions. This was possible because in optimization rules, a fixed format for a specification was used. In this case the input instructions and the matching rules were stored on two hash tables separately. HOP followed the byte-to-byte comparison to maximize the matching speed.

In 1991 an optimizer tool implemented by Whitfield and Soffa [21] which incorporated two tools one was optimization specification language and an optimizer generator. It was for both traditional and parallelizing optimizations.

Ganapathi et al.[22] introduced the tree matching pattern using tree manipulating language, twig. Attribute grammar parsing was used for describing the target machine code. Pattern matching was also done here using attribute evaluation and every pattern had one precondition and a replacement.

II.C. Combining Optimization with Code Generation:

To gain more speed in executing it was observed that it was better to reduce the execution time and optimization phases.

In the new version of HOP(1986)[20] Fraser and Wendt introduced ‘recycling’ system which used to generate the code in one phase avoiding in separate phase to save time.

In the year 1995 Ancona[16] introduced a similar model where the peephole optimization was integrated with re-targetable code generation phase. The intermediate code optimizer was programmable using some user defined optimization and translation rules which were written as a form of macros in a simple high level language. The intermediate code optimization and target code generation were used at the time of optimization development, test phase and then translation rules.

At the end a code optimizer was generated according to the tested rules for a specific target machine by target code generator.

II.D. Space Exploration techniques:

Cooper et al. [23] proposed the “adaptive compilation” where probed several optimizations during compile time and the outcome of those optimization phase used to calculate with primitive objective function which helped to count the
static number of instructions. Basically this is the reason for which the calculated values were experience dependent.

It was focused on the phases of compilers, not on any other parameters.

Kisuki et al. [24] actualized a compiler that crosses the optimization space for loop unrolling and tiling and runs all the delivered code to pick the best form of a loop kernel.

A balanced code technique between the code size and performance was proposed by Bodin et al. [25] which was also an iterative compiler. But this used to consume huge time to compile as since they seek a restrictively vast optimization space what's more, they include running every rendition of the system in place to gage its performance.

Wolf et al. [26] presented an algorithm for joining five diverse loop transformations, specifically fusion, fission, unrolling, interchanging, and tiling.

For every arrangement of nested loops the algorithm considered different guaranteeing blends of these transformations. The algorithm stopped shy of producing code for every blend of transformations, it utilized an execution estimator which acknowledges the succession of loop transformations as a contention. The execution estimator could produce an appraisal for the execution figured it out by applying the given succession of loop transformations without really changing the code. This algorithm was better option for both compile time and final code quality, but it could not be generalized to achieve more optimization other than those five original loop transformations.

Spyridon et al. [27] proposed optimization space Exploration (OSE) method where it was able to apply multiple optimization sequences at the same time for each of the code segment. After generating the several optimized version of the same code, compiler picked the suitable one as determined by a performance evaluator. But from the practical view point the OSE needed to limit the several configuration for each code segment.

III. Proposed Algorithm

Though today’s life every system is a multicore system still most of the compiler uses the single code during compilation as well as execution. The vision of our algorithm is to exploit the core as much as possible. As we all know that a program is sequence of instructions that must execute sequentially, otherwise the final outcome may differ also. So when we are talking about the parallel execution, we should also keep that point in our mind, so that the sequence would not differ. So basically we are segmenting the code according to the dependency on each other and each of those segment would be given to a core for parallel execution in such a way so that the sequence would not be violated and also the final outcome.
Now according to the programming concept we could say that basically two types of statements are there in a program, one is independent one, and another one is dependent one. So initially we are assuming that we have two code segment, named as independent segment and other is dependent segment (Diagram 1. stage A).

User input, initial initialization, constant assigning, complex constant calculation (Constant Folding), constant propagation, dead code elimination all can be considered under independent segment. Now rest of all other code can be considered under dependent segment, like- loop execution, variable assigning, calculation of different variables etc. Now from the experience we could easily understand that in program the independent part is always limited up to some extent. Though we are executing the independent part and the dependent part on to two separate core, still it could not achieve the considerable optimization ratio than executing in single core, as after executing the independent part the first code would be idle.

![Fig 1: Different Code Segments.](image)

So we are exploiting the first core again to execute a part of a dependent code segment. We have named it as relatively dependent code segment (Diagram 1. Stage B). Now the question is how we are exactly re-segmenting the dependent part and executing it near about parallel? The code segments those are dependent on another one or two dependent segment those are identified as relatively dependent parts and if there is any segment which is dependent on the relatively dependent segment, we would consider that one under the simple dependent segment. Basically we are considering the further dependence of a code under the simple dependent segment, just to distribute the payload on each core as equal as possible, whereas the final outcome would not affect if the execution sequence is not disturbed.

We can easily explain this with the Diagram 2. It represents three graph nodes which are basically the three code segments. If it is i->j, it means jsegment is dependent on i, i.e. only after execution of i, j is allowed to execute.
Fig 2: Inter-relational Diagram among different Code Segments.

We are utilizing the first core to execute the relatively dependent segments only after execution of its pre-dependent parts on the second core. Again if there is any dependent segment which is dependent on any relatively dependent segment that would be execute in second core. The Diagram 3 would help us understand the scenario.

Fig 3: Architectural Abstraction Model of Code Execution.

Here ‘is’ means the independent segment, ‘ds’ stands for dependent segment and ‘rds’ stands for relatively dependent segment. According to the diagram it can be observed that as soon as ‘is1’ completes its execution, the ‘ds1’ and ‘is2’ starts their respective execution. Then as soon as they completes ‘rds1’ and ‘ds2’ starts their execution. In between if ‘ds2’ ends its execution the core2 picks the ‘ds5’ or ‘ds3’ arbitrarily. If it is ‘ds5’, the next segment in core1 will be ‘rds3’and ‘ds3’ in core2. Once ‘ds4’ also finishes its execution in core2 the ‘rds2’ starts execution. This theoretically implies that the payload has been more or less equally distributed. But this type of mechanism is dependent on three factors-first one is the length of the each segment, because that only determines what would be the next segment that would be chosen by core. In the above situation, if ‘ds2’ is a lengthy segment and it ends after the ‘rds1’ then the core1 would be idle for some time, because of dependence. The second factor is the general anatomy of the program. If the dependence in a program arises in such a way that most of the cases the dependent segments and relatively
dependence segments are overlapping with each other, it is difficult to identify for a processor as well as compiler how to distribute those segments among the cores. The third factor is the compiler code segmentation ability. If a compiler is not so good to segment the whole code according to the dependencies among them, then the whole processes will be futile. Basically this would be the most aroused problem during machine level implementation, because it is not so easy to predict the branches and relative dependences all time.

IV. Experimental Results

Due to the time limitation and several hardware limitation we did not implement this technique in machine level. But to realize the concreteness of this proposed algorithm we have executed several programs related to some general well known problems through Open MP environment. Here we segmented the code manually which has leaded this approach to the optimum efficiency. Though it is easily understandable that this optimum efficiency achieving is nearly impossible in practice, while the code segmentation would become automatic through branch prediction and dependency identification mechanism. Here we are representing the simple sorting algorithm, Spanning tree algorithm and shortest path algorithm. To get the more precise value of efficiency we executed them as an iterative way, so that the next input can be taken by the first core while the second one is already calculating the previous one. Intuitionlly it can be assumed that this may interpolate some error into the result of the first iteration, if the array elements are altered during calculation. But during experiment we found that up to near about three hundred data it was not disturbing the execution model. This happened because the calculation was continuous on the second core with the speed of near about 2.2GHz where as the user input was discrete and also dependent on the human speed which is very less compared to a computer processing speed. So in the Diagram 4 we are representing the comparative experimental result, ignoring the user dependent time as much as possible, though the great achievement of our work is that we make the calculation part independent from the user after the first iteration. The following tables and diagram explain the efficiency diagrammatically.

Fig 4: Comparison of Different algorithm according to their execution time.
Fig 5: Line graph between proposed method and conventional method for different algorithm.

In the above two figures we have represented the total execution time of each algorithm. Basically the executing time is evaluated by the formula: (Clock time when it completed its execution - Clock time when it was started). Lastly to get the efficiency in term of any standard unit we calculate it with the following formula:

$$Ef. \ gain = \frac{Execution\ time_{conventional} - Execution\ time_{Proposed}}{Execution\ time_{conventional}} \times 100$$

Fig 6: Efficiency Gain Comparison.

From the above graph it can be understandable the efficiency gain is near about 35% in average for our tested cases.

In this above mechanism we have proposed a model that can change the execution model in near future. But the possibilities of future development in this model can be more evaluative, as we segmented the code manually whereas the segmentation techniques of code in machine level can generate a lot of variations. So the future possibility is to implement this type of code segmentation for multicore in architectural level with some automated techniques.

V. Conclusion

The optimization should be done according to the characteristic of the target machine. Because sometime the same compiler code can be used for different target machines only by changing the machine dependent parameters.
Machine generated compilers are more optimized than the handwritten. However this optimization never assure that it could be perfect one, equivalently for all cases. Also it is impossible to get a fast possible equivalent compiled program for all source code, because indirectly it means that the compiler would solve the halting problem.

According to the today’s architecture some system has more than one functional units (core) and also ALUs which helps to execute multiple instructions at the same time. So in our proposed algorithm we have used this technique to exploit the as many cores as possible simultaneously and we have gained near about 35% of efficiency that the convention execution model. But it is very important that how efficiently the compiler is using maximum number of functional units with proper scheduling of instructions.

References


Corresponding Author:
Kaushik Roy*,
Email: kroy.compsc@gmail.com