Optimizing Binaries for Multiple Gain-factors Using State-based Model

Amit Gupta, Balpreet Singh Pankaj, Rajeev Kumar and P. P. Chakrabarti
Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur
Kharagpur, WB 721 302, India
rkumar@cse.iitkgp.ernet.in

Abstract

One of the most important and challenging aspects of developing software tools for embedded systems is retargeting of a compiler to the custom processor. However, such a compiler may not always optimally utilize the enhancements made in the architecture by the designer and thus there is a need for post-compilation optimizations to the code generated by such a retargeted compiler. In this poster, we present some of the recent results that we have obtained from the compiled binaries for ARM and Intel family of processors. We discuss the possible applications of this novel strategy in design-space exploration of the processors.

I. Introduction

Embedded systems use processors which are optimally designed to suit the application-specific needs. To support fast compiler design for such custom processors and hence support architecture exploration, researchers have proposed retargetable compilers. However, many existing retargetable compilers for ASIPs and domain specific processors such as DSPs generate low quality code due to the highly specialized architecture of ASIPs, whose instruction sets are incompatible with high-level languages and traditional compiler technology. Because this poor code is virtually useless for embedded systems under efficiency constraints, it requires a time-intensive post-compilation optimization which must utilize domain-specific code optimization techniques that go beyond classical compiler technology [7, 8, 9].

Several post-compilation optimization techniques using pattern-matching as a means, are emerging [4, 6]. Peephole optimization remains one of the most practiced approaches for such tasks [1, 2, 3, 5]. However, peephole techniques are limited by a pre-defined window size and look for instructions to be lexically adjacent. In this paper, we present a technique which overcomes these shortcomings by allowing interleaving of any non-dependant instruction and by having no fixed limit on the peephole width.

This new technique allows the user to specify potential instruction replacement patterns as a set of FSMs and a set of replacement actions. The method looks at 'logically' adjacent instructions(instead of lexically adjacent ones) while parsing the assembly code, trying to match instruction sequences, and if a match is found, replaces the original sequence with a more efficient code sequence according to a replacement algorithm. A term called as 'gain factor' is defined for each possible optimization (replacement) and the one with the best gain is chosen by the conflict resolution algorithm. The gain factor for each optimization is calculated based on several parameters like number of cycles reduced, code size improved, power activity reduced etc. with the given replacement.

Also, since a number of FSMs are run simultaneously on a given assembly code, each of them produces its own optimization depending on its specification. As a result, certain optimizations might overlap and we need to select the best among these. For this, a conflict resolution algorithm has been devised The use of FSMs instead of linear sequences gives this technique an inherent ability to neglect any non-dependant instructions between two potential replacement instructions. Depending on the input specification, this technique can match any number of instruction sequences and thus can circumvent the shortcoming of limited peephole width of a peephole optimizer, which is generally three or four instructions. Also, this powerful specification technique allows the user to express several peephole optimizers using one succinct FSM specification.

The technique is a superset of peephole optimizers in the sense that the user may specify some possible replacements using this technique which cannot be specified with peephole optimization techniques. It is a post assembly optimization technique and thus may be incorporated as a post compilation pass for any compiler. In this paper we present the technique and define its design and implementation. Also, the conflict resolution algorithm which was implemented is explained. We then show the results obtained by running the method on i386 and ARM binaries. These results have been produced only for seeing improvement in code size (by gaining in number of instructions or else wise) by using this technique. Other parameters like gain in number of cycles and power consumption would be tested later.

II. Notation, Definition and Model

First, we define some notations which we use in rest of this paper:

1. Instruction (I): An instruction is considered as an ordered pair - *(Operator, Operand1, Operand2, Operand3)*.

2. Basic Block (BB): A Basic Block is defined as a set of instructions, $U(I_i)$, $i=1$ *to* n, such that the sequence of instructions I_1 -> I_2 -> I_3 ... -> I_n have a single entry & exit point i.e. I_1 is the entry point and I_n is the exit point.

3. Instruction set FSM is defined by a 5-tuple, $M= (Q, \sum, \delta, q0, f)$
   Q: Set of States S
   $\sum$: Input Alphabet i.e. I U V, where I is an Instruction, V is set of user defined state variables, updated at each transition of the FSM.
   δ: Transition Function i.e. $S \times T$ -> S, where S is a state, T is a set of all triplets of the form $(I_k, Pre_k, Post_k)$, where I_k is an Instruction, Pre_k is a boolean expression on V denoting precondition for a transition to take place, $Post_k$ is a set of statements of the form $lk = rk$, where $lk \in V$ and r_k is an arithmetic expression on V
   $q0$: Initial State of the FSM
   f: Final State of the FSM

4. Optimization (*OPT*): It is defined as consisting of two sets; a set of instructions to be replaced, IS1, and the set of optimized instructions with which they will be replaced, IS2.

5. Optimization Set (*OPS*): Such a set is the union of all non-conflicting optimizations (OPT) which can all be applied together on the respective Basic Block.

6. Optimization Sets Set (*OPSS*): Such a set is the union of all possible Optimization Sets (OPS) available on a Basic Block.

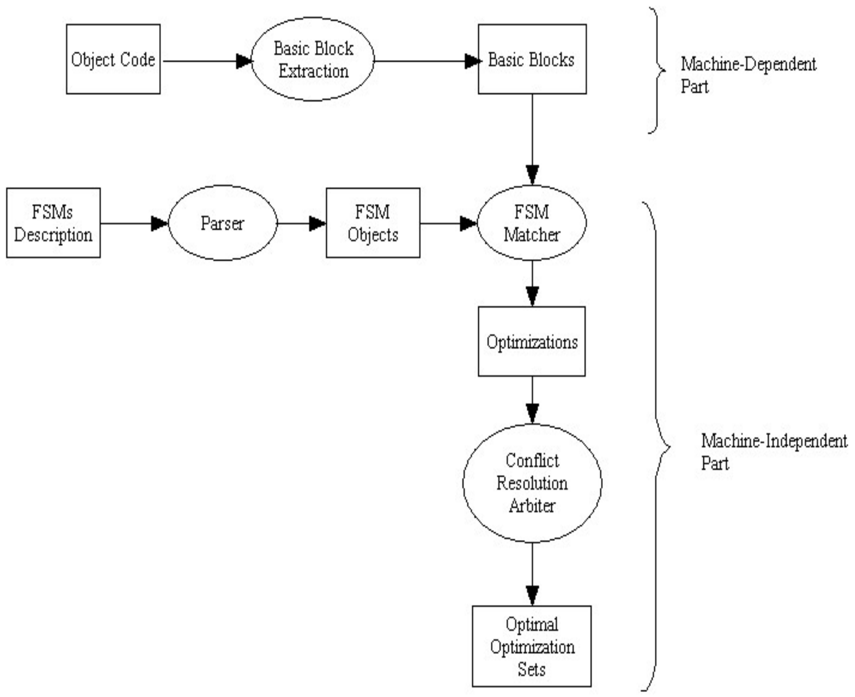

Fig. 1: A Schematic showing the phases involved in post-compilation optimization

III. Implementation

Algorithm

```
1. Form Basic Blocks from object code
2. Read description of all FSMs from input description file and generate the entire FSM Set.
3. For each Basic Block
        For its each instruction
            Run all the FSMs on it
        For all optimizations obtained in this Basic Block
            Make non-conflicting sets //Replacement Algorithm
        Choose the optimal "Optimization Set" on this Basic Block
        Reset all FSMs
```

Running of an FSM: As explained in the definition of FSM, each FSM has a set of states, a start state, a set of state variables, end state and a set of transitions from (or to) each state such that each transition has its own pre-condition (which when true allows the transition to occur) and a post-condition (which needs to be set once the transition has taken place).Every transition in the FSM has a precondition (sometimes NULL) which is a Boolean expression in state variables and allows us to check for the validity of instruction encountered using the state variables. Thus state variables check for validity of instructions parsed (through precondition) and then allow us to carry out one of the following steps:

1. Allow the transition to occur
2. Do nothing (remain in this state) and parse the next instruction
3. Check if the FSM needs to be reset.

Once a transition take place, it has a post-condition (sometimes NULL) to be set. This expression is also in terms of state variables.

The algorithm with which each FSM runs is as follows:
For each instruction obtained

1. Store state variables in a temporary structure.
2. If the FSM has a transition in its current state on the parsed instruction, then go to 3 else go to 6
3. Update the state variables with the values obtained in the parsed instruction and evaluate precondition for this transition. If true go to 4 else go to 5
4. Set the post-condition, transit to next state and go to 7
5. Restore the state variables values and go to 9
6. Check if the current instruction has parameters whose values are same as contained in any of the state variable. If true, **reset FSM** and go to 9 else directly go to 9
7. If final state and final state has no transitions, then go to 9 else go to 8
8. Parse the next instruction and go to 10.
9. **Store the optimization** and **reset the FSM** and Go to beginning of the loop (i.e. parse next instruction)
10. If the FSM has a transition for this instruction in the final state, then update the state variables with the values obtained in the parsed instruction and evaluate precondition for this transition. If false then go to 11 else go to 8.
11. Store the optimization and reset the FSM and go to 1. //starting the FSM again from this already parsed instruction

Reset FSM: Take the FSM to start state and restore the initial values of all the state variables. The FSM is reset under two conditions. First, when the instruction parsed has no transition on the current state of FSM and it sets or uses the value of a register which is contained in any of the state variables, and second, when the final state is reached.

Conflict Resolution (Replacement Algorithm): We have a set of optimizations obtained on the given Basic Block. Let this set be denoted by A1, A2, …, AN. The gain factor of each such optimization takes into account things like reduction in code size, cycles and power requirements. In Figure 2, they can be seen as a1a2, b1b2, d1d2, e1e2. These optimizations cannot be all applied together since they conflict among themselves. e.g. a1a2 can be used with b1b2 and not e1e2 or d1d2.

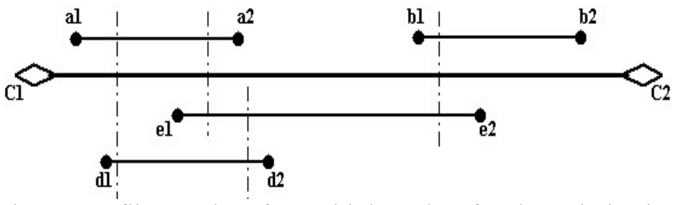

Fig. 2: Conflict resolver for multiple paths of code-optimization

Let Sj = {Ai | each i is disjoint}, then Sj denotes one such overall set of possible optimizations which can be all applied together on this BB. Each such set has a gain factor associated with it. This set is the same as OPS defined in definitions section above. For example, members of S_j would be like {a1a2, b1b2} or {d1d2, b1b2} or {e1e2} . The gain factor of each set is the sum of gain factors of optimizations which make up the set.

Let S= U(Sj), then S is the set of all possible optimization sets, where each Sj has a value of gain factor associated with it. This set is the same as OPSS defined in Definitions above. e.g. S in our example would be { {a1a2, b1b2}, {d1d2, b1b2}, {e1e2}, {a1a2} }

This algorithm then chooses the set having maximum value of gain factor from S and this would be the optimal optimization set for this Basic Block.

IV. Results

The technique was tested to carry post-compilation optimization on i386 (CISC) and ARM(RISC) binaries. FSMs of possible assembly peephole patterns were made and then run on these executables which were produced by GNU Compiler. We obtain results for improvement in code size by replacing existing instructions. Inclusion of other parameters like cycle count and power requirements in gain factor was not done as of now which is an area of current activity.

The results obtained showed clear capability of our model in interleaving any intermediate non-dependant instructions during transitions of FSMs and the use of conflict resolution (replacement) algorithm to select the optimal set of optimizations for a given basic block. The i386 peephole patterns were tested on already optimized binaries produced by GCC (Figure 3 & 4). ARM patterns were applied on binaries which were produced by not allowing GCC to do peephole optimization on assembly (Figure 5 & 6).

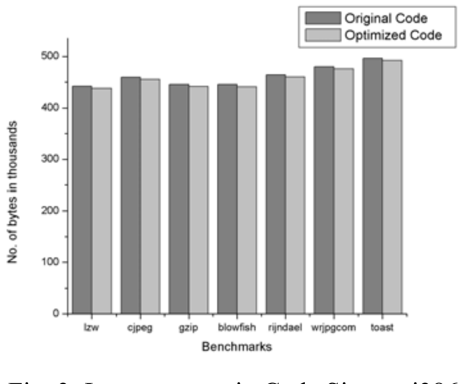

Fig. 3: Improvement in Code Size on i386

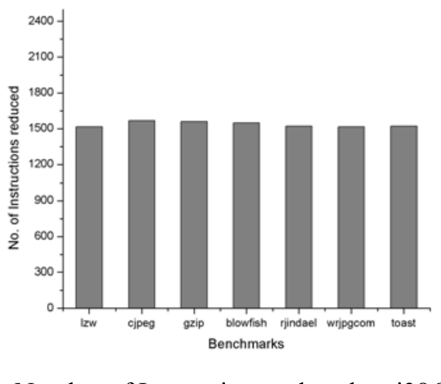

Fig. 4: Number of Instructions reduced on i386

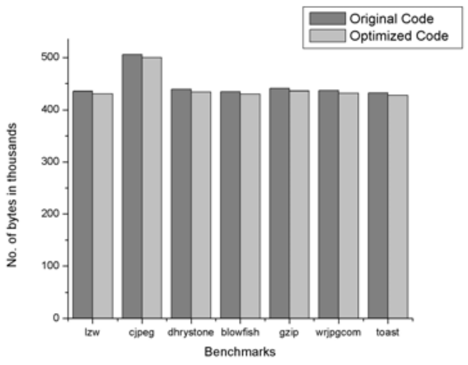

Fig. 5: Improvement in Code Size on ARM

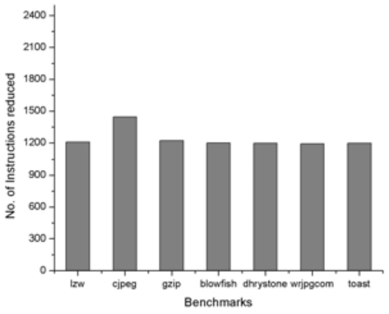

Fig.6: Number of Instructions reduced on ARM

The results obtained though small in number are significant because code was maximally optimized. One point to note is that our model will give results which are as good as the input FSMs supplied of peephole patterns. Hence, better the patterns available better will be the optimization done. For ARM, the patterns available were only restricted to load & store instructions and hence even though we tested the model on its un-optimized binaries, we couldn't get substantial results.

To show the complete capability of this model, commonly recurring patterns in i386 executables were found and made into FSMs. These FSMs were then run on the same executables to check whether the model is able to detect all of them (Figure 7). This observation could be used to replace the recurring set of existing instructions with a new optimized instruction(s). Such an approach can be very useful in Design Space Exploration process for extensible processors in defining newer instructions without modifying the backend of the compiler.

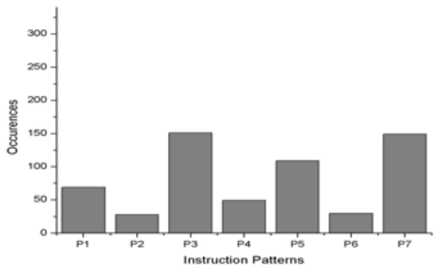

Fig. 7: Number of instances of recurring patterns found in i386

V. Conclusions

In this work, we have presented a novel strategy to optimize the compiled binaries where potential instruction replacement patterns are specified using FSMs. This technique is a superset of peephole optimizers and is powerful enough to be able to represent a number of peephole optimizers in one succinct representation. Whenever there is more than one optimization possible on the same set of instructions, a conflict resolution algorithm chooses the best among these based on multiple gain factors like number of cycles reduced, code size improved, power activity reduced etc.

The technique was implemented and tested for possible optimizations on Intel and ARM binaries. In this work, we used this framework to minimize instruction counts and code-size. However, this technique can be used to optimize power consumption as well as cycle counts. We are also exploring to use this framework to design application specific newer instructions which can replace the existing set of recurring instructions.

Acknowledgment

I would like to acknowledge the contribution of Mrinmoy Ghosh who worked with us initially on this technique.

References

[1] A. S. TANENBAUM, H. V. STAVEREN and J. W. STEVENSON. *Using Peephole Optimization on Intermediate Code*. ACM Transactions on Programming Languages and Systems 4, 1 (January 1982), pp 21-36.

[2] J. W. DAVIDSON and C. W. FRASER. *The design and application of a retargetable peephole optimizer.* ACM Transactions on Programming Languages and Systems 2, 2 (April 1980), pp 191-202.

[3] W.M. MCKEEMAN. *Peephole Optimization.* Communications of the ACM 8, 7 (July 1965), pp 443-444.

[4] R. GIEGERICH. *A Formal Framework for the derivation of Machine Specific Optimizers.* ACM Transactions on Programming Languages and Systems 5,3 (July 1983), pp 478-498.

[5] R. R. KESSLER. *An Architecture Description Driven Peephole Optimizer.* ACM SIGPLAN Notices, 19, 6 pp 106-110 (June 1984)

[6] P. B. KESSLER. *Discovering Machine Specific Code Improvements.* ACM SIGPLAN Notices, 21, 7, pp 249-254 (June 1986).

[7] LEUPERS R. *Compiler Design Issues for Embedded Processors.* Design & Test of Computers, IEEE. 19 , 4(July-Aug. 2002) pp 51 - 58

[8] M. Ghosh. *Compiler Backend Generation and a Post-Compilation Optimization Technique for Extensible Architecture*. MS Thesis. Computer Sc. & Engineering Dept., IIT Kharagpur, 2004.

[9] B. S. Pankaj and A. Gupta. *Porting .NET and a Post Compilation Optimization Technique*. B. Tech. Project Report. Computer Sc. & Engineering Dept., IIT Kharagpur, 2004.