Implementation of Invariant Code Motion Optimization in an Embedded Processor Oriented Compiler Infrastructure

Ivan Považan, Marko Krnjetin, Miodrag Đukić, Miroslav Popović

Abstract—**In order to achieve better performance results for the generated code various compiler optimizations are performed during the compilation process. Even though it is expected that optimization would improve the code quality, for embedded processor with small amount of resources applying certain optimizations can sometimes lead to generation of larger number of instructions (i.e. larger code size and slower execution). One such optimization is a loop optimization - invariant code motion. In this paper we present improvements to the implementation of this optimization in an embedded processor oriented compiler infrastructure which give good results in cases of high register pressure. The enclosed test results for translated code with and without proposed improvements show that our new approach generates code with better performance in all cases.**

Index Terms—**Compiler optimization, invariant code motion, high register pressure**

I. INTRODUCTION

Having a high quality compiler for certain processor is not an uncommon practice. It is actually quite the opposite and nowadays there is a high demand for fast, robust and reliable compilers that can generate code with high quality. Quality of the code most often refers to code size (number of instructions in memory) or code speed (number of cycles needed for the code execution).

In the embedded systems domain both characteristics are highly desirable and an ideally generated code has the performance near to the hand-written assembly code. In other words, the generated code is small and fast. However, designing and developing such compilers is not a trivial job to do. Especially if the quality of the source code is taken into account. Compiler has to understand and recognize certain constructs from the source code and to perform the best compilation strategy in order to generate the best code, despite the way those are written in the starting language. There are two approaches for dealing with this problem:

- Source code adaptations
- Compiler optimizations

First approach relates to adapting the source code for certain compiler according to compiler specific coding guidelines which can help compiler in code generation.

On the other hand, more ideally, compiler should automatically perform the same and should not depend on the source code. Therefore compilers implement various optimizations which are performed automatically on the source code, or are available through compiler switches to the user, in order to achieve the best performance. Complier optimizations are not independent and applying one can affect the other which can result in better or worse code. Choosing the best combination of optimizations which would generate the best code, is a complex problem. Furthermore, compilation passes like instruction selection, resource allocation and instruction scheduling also need to be interleaved and interconnected for embedded processors to achieve better results, which additionally complicates choosing the best compilation strategy [1]. Clearly for DSP applications the most critical part of the source code, which need to be condensed down as much as possible, are loops. Optimizations like loop unrolling, software pipelining, loop tiling, invariant code motion, induction variable detection, hardware loop detection and similar, generally improve the quality of the generated code by significant percentage. However, the question is: is this always true?

Most of mentioned loop optimizations are machine independent optimizations and are performed on high level of intermediate representation of the translated code. This means that some target architecture specifics like: available resources or available machine instructions are not considered at that point in the compilation process, and applying such optimization can have undesirable effects. Moving the code outside of the loop usually increases the number of interferences between operands and can cause spills which increases the overall number of instructions. In this paper we present improvements for the implementation of invariant code motion optimization which solve these problems.

The paper is organized in six sections. This section gives an introduction to the problem domain. Next section includes description of the used compiler framework, while the third describes initial algorithm for invariant code motion and introduced improvements. Following section encloses test results and discussion about improvements acquired with modifications. Fifth section includes related work and finally

Ivan Považan, RT-RK Institute for Computer Based Systems, Narodnog fronta 23a, Novi Sad, Serbia (e-mail: ivan.povazan@rt-rk.com)
Marko Krnjetin, RT-RK Institute for Computer Based Systems, Narodnog fronta 23a, Novi Sad, Serbia (e-mail: marko.krnjetin@rt-rk.com)
Miodrag Đukić, Faculty of Technical Sciences, Trg Dositeja Obradovića 6, Novi Sad, Serbia (e-mail: miodrag.djukic@rt-rk.uns.ac.rs)
Miroslav Popović, Faculty of Technical Sciences, Trg Dositeja Obradovića 6, Novi Sad, Serbia (e-mail: miroslav.popovic@ rt-rk.uns.ac.rs)

the last section gives a conclusion.

II. RTCC

Implementation of invariant code motion optimization, described in this paper is a part of embedded processor oriented compiler infrastructure called RTCC [2]. It is a class library which includes compiler modules for creating back-ends for embedded processor compilers. The framework defines sets of modules for:

- *analyses* – control flow, data dependency, call graph, liveness, alias, single static assignment;
- *optimizations* – common sub-expression elimination, dead code elimination, constant propagation, hardware loop detection, invariant code motion, induction variable detection, copy propagation
- *compilation passes* – instruction selection, resource allocation, instruction scheduling, code generation
- *modelling target architecture* – resources, instructions, hazards, latencies, parallelization rules
- *IR (intermediate representation)* – translation from *FE IR* (front-end intermediate representation), definition of *BE IR* (back-end intermediate representation)

IR is defined in form of graph of connected basic blocks that include list of operations. *HLIR* - high level intermediate representation includes high level operations which are independent of the target architecture, while *LLIR* - low level intermediate representation or low level operations include target specific instructions. Creating a custom compiler requires using an existing front-end and modelling the target processor with RTCC modules, where some of them can be used directly while others can be derived and adjusted according to target architecture specifics.

Source code translation is performed in the compiler's FE until IR is reached. After that FE IR is transformed into RTCC HLIR and compilation resumed in the BE of the compiler. Control flow, data dependency, call graphs, and other analyses are then performed on the HLIR in order to provide additional information used by other compilation passes and high level code optimizations. Afterwards, HLIR is lowered to LLIR with instruction selection module by applying rewriting rules which translate abstract representation into target specific instructions. When LLIR is reached, low level optimizations, instruction scheduling and resource allocation are performed. Finally, the assembly code is generated.

One of the available optimization modules, within RTCC library, is invariant code motion module described in this paper.

III. Invariant Code Motion

A. HLICM – High level invariant code motion

Invariant code motion is a loop optimization technique which detects computation within a loop which result does not change over loop iterations, and moves that computation from the loop body in order to improve performance of the generated code. This motion is also called *hoisting*. The technique is most often used on HLIR in order to hoist invariant expressions, but can be also used during low level passes. Initially, RTCC included HLICM described in paper [3] which was implemented to work only as high level optimization.

In order to describe the HLICM algorithm we first need to define how to detect invariant computation; then where we can move the code; and finally how to determine if code is movable. These three procedures are defined according to [4] as follows:

Definition 1: The definition $d: t \leftarrow a1\ op\ a2$ is loop-invariant within loop L if, for each source operand a_i:

1. a_i is a constant,
2. or all the definitions of a_i that reach d are outside the loop
3. or only one definition of a_i reaches d, and that definition is loop-invariant
4. *op* some operation

Definition 2: The preheader basic block p of the loop L is immediate dominator of head basic block for the loop L

Definition 3: The definition $d: t \leftarrow a1\ op\ a2$ can be hoisted to the loop preheader p if:

1. d dominates all loop exists at which t is *live-out*
2. and there is only one definition of t in the loop
3. and t is not *live-out* of the loop preheader p

Fig. 1 shows simple example with invariant code which can be detected on HLIR. The output generated after HLICM module is displayed in Fig. 2 which shows that the algorithm can successfully detect such scenarios.

```
L1:
  i = 0
L2:
  if (i > 10) jmp L3
  a = 0x101
  array[i] = a & b
  i++
  jmp L2
L3:
```

Fig. 1. ICM example before HLICM optimization

```
L1:
  i = 0
  a = 0x101
  t = a & b
L2:
  if (i > 100) jmp L3
  array[i] = t
  i++
  jmp L2
L3:
```

Fig. 2. ICM example after HLICM optimization

However, there are scenarios where HLICM does not work as expected. Fig. 3 shows slightly modified example which is not covered by HLICM and will not be optimized as it can be. The reason for this is that global variables are not covered in the initial algorithm, and if they are volatile or changed over their address, the optimization cannot detect if they are

invariant, due to the lack of information. In this particular example *t = mem[a]* is invariant, because *a* is changed after the loop.

```
a = 0x101 // global scope
…
L1:
  ptr = &a
  i = 0
L2:
  if (i > 10) jmp L3
  t = mem[a]
  array[i] = t & b
  i++
  jmp L2
L3:
  *ptr = 10
```

Fig. 3. ICM example with a global variable

Furthermore, Fig. 4 shows slightly different situation where high level comparison *(i < 200)* cannot be hoisted due to dependency of one of the source operands.

```
L1:
  i = 0
L2:
  if (i > 300) jmp L3
  a = 100
  if (i < 200)
    array[i] = a & b
  i++
  jmp L2
L3:
```

Fig. 4. High level representation of ICM example with a comparison

```
L1:
  load 0, i
  load 300, t0 // upper bound
L2:
  cmp i, t0
// gt - greater than
  jmp gt L3
  load 100, a
  load 199, t1
  cmp i, t1
  and a, b, t2
// le - less or equal than
  storec le t2, mem[array+i]
  add, i, 1, i
  jmp L2
L3:
```

Fig. 5. Low level representation of ICM example with a comparison

If we observe Fig. 5 we can see low level representation of the example from Fig. 4. Focusing on comparison and its low level representation we can see that there are additional opportunities for increasing code efficiency:

```
HLIR:  if (i < 200)
LLIR:  load 199, t1  // invariant
   cmp i, t1
```

Low level load becomes another potential operation for invariant code motion. Following subsection proposes improvements for these cases.

B. LLICM – Low level invariant code motion

Fig. 3 shows an example where global variable is used in invariant computation. In order to detect invariant code in such examples initial approach needs to be extended with information from alias analysis module which determines if a variables location has been accessed, and whether its value has been changed over its address or not. According to this we are adding an extension to the *Definition 1* as follows:

5. if there is a_n $(n \in i)$ that is a static storage duration variable - a_n must not be not volatile, and its value is not changed over address before usage

By implementing this extension the LLICM can detect and safely hoist loading of *a* variable outside of the loop in the given example, since its value is changed over the address after the loop. However if the change **ptr = 10* was placed inside of the loop body, LLICM would not hoist the *t = mem[a]* since the alias analysis would provide information about the change for variable *a*.

To answer the problem of not detecting hoistable code from Fig. 4 we simply need to perform LLICM algorithm on low level representation as well, just before instruction scheduling and resource allocation, in order to enable hoisting low level instructions that are invariant.

Clearly with both proposed improvements LLICM can further decrease the size of the loop body.

C. RPICM – Register pressure invariant code motion

Hoisting of invariant computations is not harmful if the resulting operands are not used within a loop. If that is not the case, number of live operands will grow for each motion which can lead to increased pressure on registers. When the optimization is applied on HLIR there is no information about used resources since the allocation is not yet performed. However, with our modifications introduced for LLICM we have gained the opportunity to control when to perform the hoisting.

In order to do this we introduce RPCIM algorithm as a modified version of LLICM which clones the LLIR before

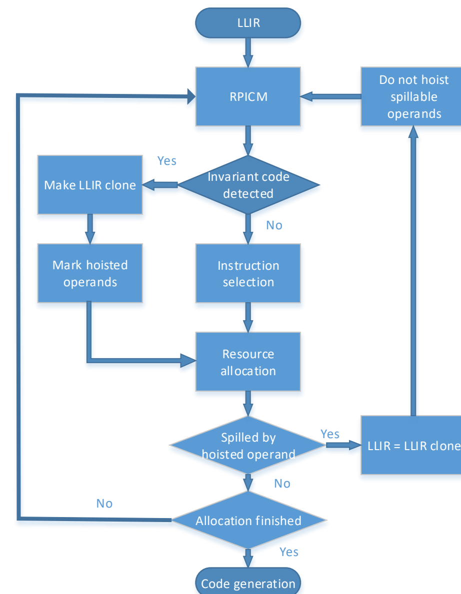

Fig. 6. RPICM algorithm

hoisting, performs the code motion and stores indication about hosted operands. Afterwards, instruction scheduling and resource allocation are performed. If a spill occurs during resource allocation, the module checks if the spilled operand is in interference with hoisted operand. If that is the case the current LLIR is replaced with LLICM clone and the process is repeated without hoisting the offending operand. In this way we will not hoist operands that would introduce spills otherwise. Fig. 6 shows the algorithm.

IV. RESULTS

Described improvements are added to existing port of RTCC compiler for a custom DSP class processor. Some of the important features of this DSP are: Harvard, load/store architecture, VLIW, two memory banks, two arithmetic units which allow instruction level parallelism, address generation unit, with limited amount of data registers: 4 general purpose, 3 temporary and 4 accumulators which makes it a good candidate for testing high pressure on registers.

In order to test the performance with and without applying the modifications to invariant code motion optimization for testing and verification we have used following test and measured number of cycles required for the code execution:

- Execute group of tests that include loops from DejaGnu test suite – 293 tests
- DSP benchmark projects which include
    - FFT algorithm
    - CDMA algorithm
    - Bi-quad filter
    - FIR filter

TABLE I
COMPARISON RESULTS IN NUMBER OF TESTS THAT AFFECTED THE GENERATED CODE POSITIVELY OR NEGATIVELY FOR DEJAGNU TEST SUITE

DejaGNU - 293 tests			
	HLICM	LLICM	RPICM
No effect	249	181	189
Worse code	16	29	16
Improvement	28	83	88

TABLE II
COMPARISON RESULTS OF OVERALL EFFICIENCY REDUCTION AND IMPROVEMENT BETWEEN PROPOSED TECHNIQUES FOR DEJAGNU TEST SUITE

DejaGNU - 293 tests		
	HLICM vs LLICM	HLICM vs RPICM
Worse code	4.44%	0.00%
Improvement	18.77%	20.48%

TABLE III
COMPARISON RESULTS IN NUMBER OF EXECUTION CYCLES BETWEEN PROPOSED TECHNIQUES FOR TESTED DSP APPLICATIONS

DSP applications			
	HLICM	LLICM	RPICM
FFT	56587	55989	55057
CDMA	314116	274097	274097
BIQUAD	120475	120475	120475
FIR	248157	248157	248157

TABLE IV
COMPARISON RESULTS OF OVERALL IMPROVEMENT BETWEEN PROPOSED TECHNIQUES FOR TESTED DSP APPLICATIONS (-% MEANS FASTER CODE)

DSP applications		
	HLICM vs LLICM	HLICM vs RPICM
FFT	-1.07%	-2.78%
CDMA	-14.60%	-14.60%
BIQUAD	0.00%	0.00%
FIR	0.00%	0.00%

Table I shows comparison results between initial implementation of ICM – HLICM, and two new proposed techniques LLICM and RPICM in number of DejaGNU tests that:

- did not have any effect after ICM was applied
- generated slower code after ICM was applied
- generated faster code after ICM was applied

We can see that the best performance is gained with the RPICM and the worse result with the initial implementation of ICM. However, Table I also shows that LLICM introduces worse performance for 13 test cases. This happens because applying ICM on low level instructions increases the opportunity for detecting and hoisting invariant code. Table II emphasize these results furthermore, by giving an overall comparison of reduction and improvement of performance compared to the total amount of tests between proposed approaches. It can be concluded again that RPICM gives the best results and also fixes the issues introduced with low level variant of the optimization. On the other hand, Tables III and IV include execution cycle count comparison results for DSP application tests, between ICM techniques and show overall gain in percent. By analyzing the displayed results it can be concluded that new techniques generate overall better results for tested DSP applications.

Tables also show that even though RPICM is the best technique overall, it still does not solve the problem of spilling operands hoisted during high level invariant code motion.

V. RELATED WORK

It is known that expression or instruction hoisting methods can de-optimize the translated code. This especially comes to the fore on target architectures with limited amount of resources and optimizations like common sub-expression elimination or invariant code motion. Such code movements extend the liveness of operands and increase the number of interferences between them causing a potential spill to happen – RP (register pressure). There have been many attempts to deal with this problem.

Some of them target the solution of the problem on high level passes which can cause high register pressure. Ma and Carr propose predictive algorithm for register pressure on a

loop in [5]. The algorithm works on a high level before unroll-and-jam loop optimization is applied and manages to get improvement in performance by reducing the number of potential spilling.

On the other hand changes of the low level passes can also improve performance. Proposed techniques like live range splitting and rematerialization presented in [6] and [7] are resource allocation techniques that deal better with high register pressure. Similar is noticeable with proposed register pressure-aware instruction scheduling proposed in [8]. Likewise, our approach includes modifications to the low level compilation, but does not include changes to resource allocation nor to instruction scheduling algorithms. By providing feedback information about hoisted operands, spill candidates and their interferences, and by allowing invariant code motion, instruction scheduling and resource allocation to be invoked multiple times we can achieve better results in high register pressure scenarios which is shown in enclosed results.

VI. Conclusion

Results demonstrate that improvements introduced to invariant code motion optimization give overall much better performance results than the initial implementation. Also results for the proposed technique for controlling when to perform the optimization show that in cases of high pressure on registers compiler can successfully detect solution which gives the best performance in all test cases.

Due to the fact that there are cases when the optimization introduces overhead in the generated code, described implementation can still be improved. This is true for cases when invariant code motion happens on HLIR and therefore one of the future improvements would be tracking the hoisted operands on HLIR and applying similar algorithm to the one described in this paper. However, that approach would probably have a larger impact on compilation time.

Acknowledgment

This research was partially funded by the Ministry of Ministry of Education, Science and Technological Development of the Republic of Serbia under project No: TR-32031. This research was performed in cooperation with the RT-RK Institute for Computer Based Systems.

References

[1] T. Kisuki and P.M.W. Knijnenburg and M.F.P. O'Boyle and H. A. G. Wijshoff: "Iterative Compilation in Program Optimization", Netherlands, 2000

[2] M. Đukić, M. Popović, N. Četić, I. Považan, "Embedded Processor Oriented Compiler Infrastructure", Advances in Electrical and Computer Engineering, Volume 14, Issue 3, 2014, pp: 123 – 130, DOI: 10.4316/AECE.2014.03016

[3] S. Radonić, M. Đukić, N. Četić, "Jedno resenje kompajlerske optimizacije za premestanje invarijantnog koda van petlje", TELFOR, Belgrade, 2014

[4] A. W. Appel, "Modern compiler implementation in C", Second edition, pp. 418-425, Cambridge University Press, 2004.

[5] Y. Ma and S. Carr. "Register Pressure Guided Unroll-and-Jam", In The 2008 Open64 Workshop, 2008.

[6] V. Makarov "Fighting register pressure in GCC", Proceedings of GCC Summit, 2004

[7] P. Briggs "Rematerialization" '92 Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation, pp. 311-321, USA, 1992

[8] R. Govindarajan, H. Yang, J. Amaral, C. Zhang, and G. Gao "Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures", IEEE Transactions on Computers, pp. 4-20, January 2003.