Embedded Processor Oriented Compiler Infrastructure

Miodrag DJUKIC, Miroslav POPOVIC, Nenad CETIC, Ivan POVAZAN
Faculty of technical sciences, Novi Sad, Serbia
miodrag.djukic@rt-rk.uns.ac.rs, miroslav.popovic@rt-rk.uns.ac.rs, nenad.cetic@rt-rk.com, ivan.povazan@rt-rk.com

***Abstract*—In the recent years, research of special compiler techniques and algorithms for embedded processors broaden the knowledge of how to achieve better compiler performance in irregular processor architectures. However, industrial strength compilers, besides ability to generate efficient code, must also be robust, understandable, maintainable, and extensible. This raises the need for compiler infrastructure that provides means for convenient implementation of embedded processor oriented compiler techniques. Cirrus Logic Coyote 32 DSP is an example that shows how traditional compiler infrastructure is not able to cope with the problem. That is why the new compiler infrastructure was developed for this processor, based on research in the field of embedded system software tools and experience in development of industrial strength compilers. The new infrastructure is described in this paper. Compiler generated code quality is compared with code generated by the previous compiler for the same processor architecture.**

***Index Terms*—digital signal processors, embedded software, fixed-point arithmetic, software tools.**

I. INTRODUCTION

Processors used in embedded systems usually have architectural elements that are not observed in general purpose processors. Non-orthogonal instruction set, separate memory banks, vectorization, address generators, etc. are just some examples. Efficiency of such processors is highly dependent on proper utilization of those architectural elements. Creating a satisfactory compiler for that kind of processors proved to be a hard task. Standard compiler techniques give limited results, which mostly do not match efficiency requirements for embedded systems. That is why many embedded processors are still programmed on assembly level [1]. In the recent years, research of special compiler techniques and algorithms for embedded processors broaden the knowledge of how to achieve better compiler performance in those cases. However, industrial strength compilers, besides ability to generate efficient code, must also be robust, understandable, maintainable, and extensible (also observed in [2]).

A typical example of embedded processor with irregular architecture is Coyote 32 DSP [3], from chip vendor Cirrus Logic. It is a fixed-point arithmetic DSP based on Harvard architecture. Data memory bank is split into two so that parallel data access can be achieved. The processor has non-orthogonal instruction set with instruction level parallelism (ILP), where one 32 bit instruction can contain up to 6 operations, if operands are in certain resources. It also supports indirect addressing with address generators (AGU), hardware supported loops and call stack, as well as several DSP specific instructions and architectural elements, such as MAC (multiply-accumulate) and saturation unit.

An important part of the tool chain initially made for Coyote 32 DSP is the C compiler, which also supports Embedded C language extension [4]. This compiler will be called the baseline compiler in this paper. The baseline compiler is built on infrastructure for general purpose processors with several modifications for embedded processors. In practical usage the baseline compiler generated code that is good enough in earlier stages of application development process, but in all cases, at the end of that process, most of the code was written in assembly. Efforts to improve generated code quality were not very successful. Clearly, compiler techniques for general purpose processors where not sufficient for achieving desired code quality (in this paper mainly code size), whereas implementation of embedded processor oriented compiler techniques in the used infrastructure proved to be difficult. Changes of the baseline compiler source code that were made in order to support some embedded processor oriented techniques significantly reduced understandability and maintainability of the compiler.

Therefore, the goal was set to develop a new compiler infrastructure that will be a good base for building industrial strength compiler for embedded processors. Such compiler must satisfy these key characteristics: generation of efficient code, understandability, maintainability, and extensibility.

In other words, the infrastructure must provide means for convenient implementation of embedded processor oriented techniques. With these goals in mind, a new infrastructure was created, and a new compiler for Coyote 32 DSP was made using it. Development of infrastructure and compiler was based on research in the field of embedded system software tools and experience in development of industrial strength compilers. This paper describes the resulting compiler, discusses its advantages from the point of compiler developers and compares the infrastructure of the novel compiler with the baseline compiler.

The next section gives overview of related work, and explains design criteria that were used for compiler development. The third section specifies the main compiler design elements introduced in the compiler framework and how they improve compiler development process. In the fourth section the compiler is evaluated in several ways and results are discussed. The fifth section contains conclusions and ideas for future work.

This work was supported in part by the Serbian Ministry of Science TR32031, 2011-2014.

Digital Object Identifier 10.4316/AECE.2014.03016

II. Related Work

How compilers for embedded processors differ from compiler for general purpose processors is subject of research in [5-7]. It is emphasized that for embedded processors code quality is more critical, whereas programs are generally smaller. As a consequence, it is much more acceptable to let the compiler run for longer time (maybe even hours or days) if that would produce better code. That increases plausibility of optimizations based on multiple compilation strategies, such as general compilation parameters variation presented in [8], variation of high level code, presented in [9], and compilation phase ordering variation presented in [10]; as well as whole program optimizations in general.

Organization of compiler into compilation phases (or passes) has many advantages, and it is common practice in many compiler designs. However, decisions made in one phase influence other phases, and in order to better optimize the code communication between phases must be improved. It is observed in [5-6] that this phase coupling is especially important in relation of instruction selection, register allocation, and instruction scheduling phases, due to non-orthogonal instruction set with instruction level parallelism. A good overview of several approaches that combine some (or all) of those three phases is given in [11]. Some of more recent papers on the subject are [12], which analyses coupling of scheduling and register allocation, [13-14], which analyses coupling of all three phases. The results always show significant improvement of code quality, so phase coupling is considered to be an important aspect of compilers for embedded processors.

Phase coupling is not restricted to abovementioned three phases. In [15] coupling of scheduling phase and offset assignment optimization oriented towards better usage of AGU was proposed. Optimizations for better utilization of AGU are important when targeting embedded processors that have them. An overview of manual and compiler techniques for AGU usage optimization is given in [16].

Also in [5-6], the importance of graph-based code selection, as opposed to tree-based, in compiler for embedded processors is discussed. Complex instructions in embedded processors, such as MAC (Multiply-ACcumlate), often require complex patterns that are a problem for conventional tree-based code selectors. Some graph patterns and pattern matching techniques are described in [17-18].

In [19] it is analyzed how programmer can improve quality of compiler generated code by inserting in the source code hints to the compiler. Some element of C language, such as restrict keyword and static in parameter array declaration, are a form of a programmer inserted hint, because they describe non-functional properties of the code which are intended to help the compiler perform some optimizations. Those hints provide information that is very hard, or even impossible, for compiler to obtain. Sometimes compiler specific hints exist, usually expressed through pragmas or attributes. Also, a particular way the code is written can influence compiler optimizations, which is discussed in [9]. For two functionally identical programs but with slightly different code, the compiler can generate very different outputs because some compiler optimization was more successful for one of them. Often it is easier to have simpler optimization algorithms that relay on some form of hinting than to try to develop, or implement, powerful algorithms that are able to optimize large space of input code. It is observed in [7] that embedded programmers are mostly willing to condition their code for a particular compiler. In order to make that easier for them it is proposed to extend optimizations with additional mechanism that recognizes cases, which the optimization algorithm cannot cover well. Those cases are then reported to the programmer in form of suggestions, i.e. programming tips, which also indicate how to write code in order to fully benefit from the compiler optimizations and processor architecture. This feature is now also becoming available in compilers for general purpose processors, but mostly due to non-standard architectural elements recently finding their way into general purpose architectures (an example is vectorization by Intel compiler [20]).

III. Compiler Framework Description

A. The baseline compiler description

Several key elements of the baseline compiler are designed according to [21]. Compilation flow is organized in passes and it is shown in Figure 1. C language frontend from Edison Design Group was used. Information produced by the frontend is loaded in backend in form of abstract syntax tree, and subsequently converted to target independent tree-like IR. Code selection pass is generated by the IBURG tool ([22-23]) from tree cover rules and their costs. Code selections works on tree-like IR. It finds minimal cost covering and then generates list of target operations. Algorithm for graph coloring by simplification with coalescing (described in [21]) is used for register allocation.

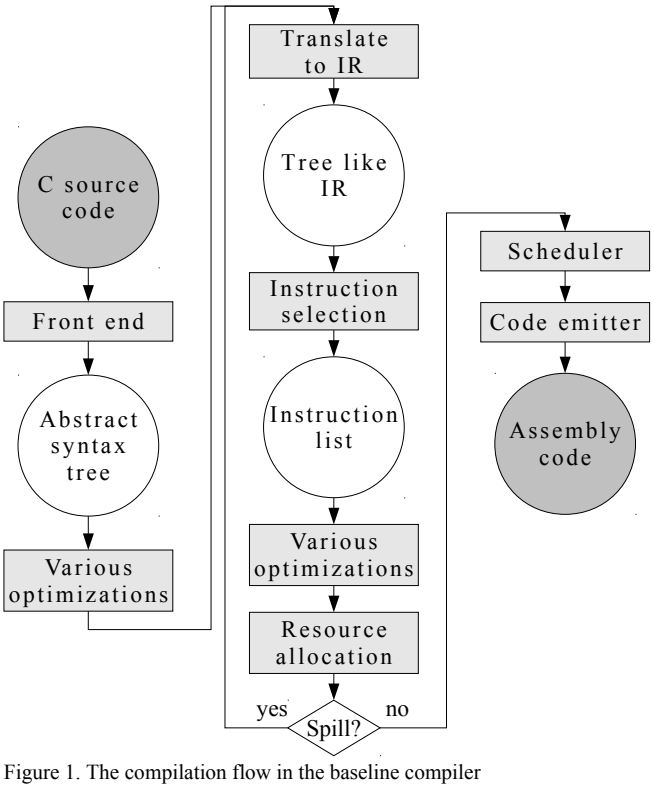

Figure 1. The compilation flow in the baseline compiler

On top of the described structure several extensions were made in order to adapt compiler to DSP platform target. On

the high level of abstraction, when working with abstract syntax tree, two passes where added: the pass for discovering loops that can be implemented as hardware loops, and the pass for usage of address generators. On the low level, when working with instruction list, a scheduling pass, based on list scheduling approach [24], was added after code generation phase. The scheduling pass contains the code for finding instruction level parallelization opportunities, and performing operation mutation, if necessary. Register allocation is slightly modified by adding heuristic that try to allocate resources so that there are more opportunities for instruction parallelization by the scheduling pass. Heuristic works without feedback from the scheduling pass. Finally, several small passes where added that performed some target dependant peephole optimizations [25], although it is important to note that peephole optimizations in this paper have wider meaning, because they do not require instructions to be successive (as described in [24]).

The baseline compiler was used for code development in several projects which had a successful outcome. The code it generated was good enough in early stages of development process, when code size and speed are not yet important. However, during the course of compiler usage several problems where noted. Algorithms for discovering hardware supported loops and usage of address generation unit where too rigid. Not all of the cases encountered in the practical usage where covered, and users would find out that a particular loop is not made to be hardware one, quite late in the development process. The level of instruction parallelism in the generated code was not very high, comparing to hand-written assembly code (see section "Results", for measurements). Maintaining the compiler was also problematic. Three different IRs added to complexity of the design. The compiler maintainer needed to know to work with all three code representations (two IRs and AST), understanding their program organization and their semantics. Code that works with tree-like IR and AST proved to be harder to debug and work with, which is also observed in [26]. Code selection phase was especially difficult to maintain. Additional knowledge was required for understanding tree-pattern rules description imposed by the IBURG tool. Each change in the rules required regeneration of the code selection pass, and because the code was automatically generated, it was more difficult to understand and debug. Finally, the used IRs where focusing on a function level, and did not express inter-procedural dependences. Because of that, attempts for adding global optimizations were requiring serious reworks throughout the compiler code.

B. The novel compiler infrastructure description

The work on the new compiler was started with the main intention to surmount shortcomings of the baseline compiler, but also having in mind the four goals named in the previous section. It was decided to develop a compiler infrastructure first, and then to build Coyote C compiler on top of it. Although there was no initial plan to reuse the infrastructure for some other language or target, there are two reasons for favoring that approach. One reason is the assumption that building compiler on top of infrastructure is more systematic and therefore easier to maintain, and the other reason is anticipation that the infrastructure might be used for some tools other than compiler, such as compiled simulator [27].

The compiler infrastructure is organized as a class library, as opposed to concept of automatic code generation and domain modeling [28]. C language frontend from the baseline compiler was reused. Abstract syntax tree that is output of the frontend is loaded into the backend. However, during loading, abstract syntax tree is immediately converted to the intermediate representation (IR), that way almost eliminating usage of abstract syntax tree from the compiler (Figure 2).

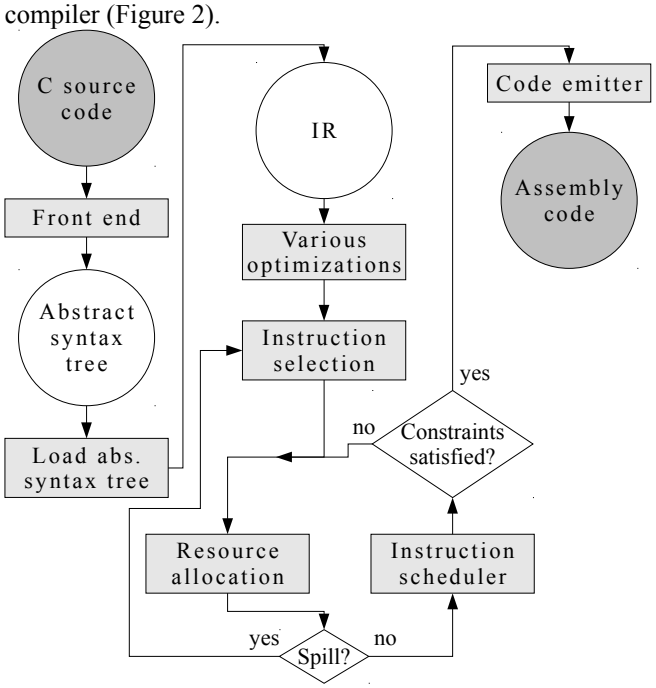

Figure 2. The compilation flow in the new compiler

There are two IRs used in the new infrastructure, one closer to the source language and one closer to the target processor. The both abstraction levels, however, have the same underlying structure. That is why effectively it is a single IR with two levels of abstraction, called: HL (High Level) IR and LL (Low Level). The only difference is that LL IR contains target processor operations (assembly code is emitted directly from LL IR), whereas HL IR has target independent operation. This design makes development and maintenance of compiler much easier. A compiler programmer needs to know only one set of procedures for handling IR and its elements.

Structural organization of IR is a hybrid one. On basic block level it is linear, whereas basic blocks are organized in a function control flow graph. Functions are connected with call edges and thus form a function call graph. Inter-procedural dependencies are integral part of the IR, enabling easier expression of global optimizations. Basic blocks contain list of instructions, where each instruction represents a single operation. This is important in context of ILP because target machine instructions can contain multiple operations. During compilation process the IR can have both HL and LL elements at the same time. As a consequence, partial code lowering can be performed, and IR can be written to a file seamlessly at any moment, which is very useful for debugging purposes.

Since no formal description of the processor architecture was available, the target machine was modeled manually. The compiler infrastructure requires aspects of target processor to be described in several different classes by inheriting the base classes and defining appropriate attributes and methods. Target processor resources are first to be modeled. The infrastructure provides a way to express basic resource groups (different memory banks, different register sets) and their sub-groups. Sub-groups are very important for expressing resource constraints involved in instruction level parallelism, which is done later in the process.

Instruction operation codes are listed in two enum types, one for HL IR and one for LL IR. Each operation code must be accompanied by its representation string, which is used for writing IR in a file. That representation string is not necessarily used for final code emission of LL IR operation codes, because one operation sometimes does not have the same string representation in all contexts, which is for Coyote 32 DSP mostly related to whether an operation is in parallel with some other instructions. The final code emitter module needs to take care of all those special cases. Information about what kind of operands a certain operation can have is not attached to the instruction definition. It is up to the compiler programmer to take care of that at places where IR instructions are created, i.e. to make sure that IR instructions are created with appropriate operand kinds. In addition to representation string, some abstract attributes must be attached to instructions, such as whether it is a branch instruction, a memory read or write, etc. Those attributes enable many of common compiler analysis and optimizations (such as liveness analysis, dead code elimination, control flow analysis, common sub-expression expression elimination, data dependency analysis, alias analysis, etc.) to run on IR regardless of operations (HL or LL ones) it contains.

The new compiler infrastructure incorporates support for expressing and applying rewriting rules, as means for transforming IR and performing graph-based code selection. For every rule a new class is defined, inherited from the base class for rules. The object of that class represents the rule, and it is used for applying the rule. Each rule class has two IRs in it: the first represents pattern being matched, and the second one represents template, which is semantically equivalent piece of IR that is going to replace the pattern in the main IR. Both the pattern and the template are created as pieces of IR. Although the IR is linear, it is interpreted as graph when matching rule patterns. For each pattern the local data dependency graph (DDG) is built and it is matched on the main IR DDG. DDG nods and edges need to match, but also operands attached to edges need to match. An operand in the compiler infrastructure is defined by its kind (a resource operand, a constant, and some subcategories), type, and, depending on the kind, some characteristics such as: value, resource class, size in memory, etc. It is possible to specify in each rule to which extend operands need to match. For example, in some rules operand type needs to match, while its resource class does not.

For most rules when pattern is matched, it means that the rule can be applied. However, if additional conditions (e.g. that some constant is in certain range, etc.) need to be checked, a special virtual function is provided that needs to contain code which checks them. Module for applying rules calls a rule's virtual function for additional conditions check, after the rule's pattern is matched. For some usual conditions, such as abovementioned range check for a constant, predefined functions are provided in the infrastructure for programmer's convenience, but evidently there is no restriction on what kind of additional checks can be performed.

The main advantages of this approach to specifying rewriting rules come from the fact that rules are expressed directly in C++, not in some additional language. There is no additional syntax learning, since the knowledge of handling IR is practically enough for writing rules. Compilation of rules is fast, because there is no intermediate step that generates code from some other language. For the same reason debugging is easier, because rules can be inspected and step through directly. One disadvantage is that the rules are very specific for the compiler infrastructure. Also, a separate language for rules might provide more flexibility and expressivity. However, implementation of Coyote DSP compiler on this infrastructure showed that expressivity is good enough for the given purpose.

Figure 3 shows an example of a rule. The rule that matches integer addition is given for four different compilers. First three rule forms are GCC form (taken from GCC MIPS port), LLVM form (taken from LLVM MIPS port), and IBURG form (from the baseline compiler), respectively. The fourth form is the one used in the new compiler. Although shown rules are not targeting the same processor, they are still comparable because they have the same meaning, only in different context, which is imposed by different compiler infrastructures. It is important to note that GCC and LLVM rules also contain partial information about string and binary formats of target instructions. In the baseline compiler, and the new compiler, information about string format is located in another place, and binary format is not emitted at all. The new compiler matching rule is the only rule that is entirely written in C++ code and does not require any special preprocessing (MAKE_PATTERN and MAKE_TEMPLATE are simple C++ macros, and their meaning is quite clear). Rewrite rules can be used for code transformations other than code selection. The fact that a single IR structure is present through the whole back-end makes this possible. In Coyote compiler rewriting rules are also used for strength reduction, algebraic simplifications, some target specific transformations on HL IR, and peephole optimizations on LL IR. There are currently 1032 rules for code selection and 108 rules for other purposes.

Every instruction has as an attribute a list of names of all the rules involved in its generation. That list is used for debugging purposes in order to be able to determine the instruction's origin. When a rule is being applied, let I be the set of instructions matched by the rule's pattern, called input instructions, and O the set of all instructions generated from the rule's template, called output instructions. Then, creation of rule names list for output instructions is described in (1).

```
(define_insn "*addsi3"
  [(set (match_operand:GPR 0 "register_operand" "=d")
   (plus:GPR (match_operand:GPR 1 "register_operand" "d")
   (match_operand:GPR 2 "register_operand" "d")))]
  ""
  "addu\t%0,%1,%2"
  [(set_attr "alu_type" "add")
   (set_attr "mode" "<MODE>")])
```

a) GCC matching rule

```
class ArithLogicR<string opstr, RegisterOperand RO,
                  bit isComm = 0,
                  InstrItinClass Itin = NoItinerary,
                  SDPatternOperator OpNode = null_frag>:
  InstSE<(outs RO:$rd), (ins RO:$rs, RO:$rt),
         !strconcat(opstr, "\t$rd, $rs, $rt"),
         [(set RO:$rd, (OpNode RO:$rs, RO:$rt))],
         Itin, FrmR, opstr>
{
  let isCommutable = isComm;
  let isReMaterializable = 1;
}

def ADD : MMRel,
          ArithLogicR<"add", CPURegsOpnd, 1, IIAlu, add>,
          ADD_FM<0, 0x20>;
```

b) LLVM matching rule

```
accumulator: PLUS_NO(accumulator, accumulator) = 103 (1);

CAsmOperand* InstSelect::Reduce(NODEPTR_TYPE p, int nonterm)
{
  int rulenumber = burm_rule(STATE_LABEL(p), nonterm);
  short* nts = burm_nts[rulenumber];
  NODEPTR_TYPE kids[10];

  burm_kids(p, rulenumber, kids);

  switch (rulenumber)
  {
    case 103:
      ...
      CAsmOperand *src1, *src2, *dst;
      ...
      src1 = Reduce(kids[0], nts[0]);
      src2 = Reduce(kids[1], nts[1]);
      ...
      return dst;
  }
  ...
}
```

c) IBURG matching rule

```
class CLowerAddNumType : public CSingleInstTypeMatchRule
{
public:
  CLowerAddNumType(CRewriter* rewriter)
  : CSingleInstTypeMatchRule(rewriter)
  {
    m_ruleName = "CLowerAddNumType";

    CBaseType* numType
      = (CBaseType*)m_types->createNumType();

    COperand* dest = m_ir->createResOp(numType, ACCUM);
    COperand* src1 = m_ir->createResOp(numType, ACCUM);
    COperand* src2 = m_ir->createResOp(numType, ACCUM);
    COperand* flag
      = m_globals->getGlobOp(CGlobOps::ARITH_FLAG);

    MAKE_PATTERN(ADD_OC, DEST(dest), SRC(src1, src2));
    MAKE_TEMPLATE(OPKIND_ADD_ACC_ACC, DEST(dest, flag),
      SRC(src1, src2));
  }
};
```

d) The new compiler matching rule

Figure 3. Four different forms of pattern matching rule that matches integer addition

$$\forall o \in O, list(o) \leftarrow \sum_{i \in I} list(i) + r \quad (1)$$

In (1) addition on lists represents their concatenation, and *r* is name of the rule currently being applied. By looking at those lists it is possible to see, at any moment, order of rule applications that produced a certain instruction.

Up to this point in compilation process, instruction level parallelism and pipeline hazards are not taken into consideration. One operation is represented by one IR instruction, and delay of each instruction is assumed to be 1. It is up to scheduler module to sort out pipeline hazards and exploit instruction parallelism. Target processor in this case has a shallow pipeline and only a couple of very simple cases of pipeline hazard. Therefore, the main task of the scheduler is to parallelize operations.

Instead of list scheduler algorithm used in the baseline compiler, which orders operation list based on some priority, then iterates through the list, selecting operations to try to put in the same cycle, in the new infrastructure percolation scheduling is used [29]. Percolation scheduling iterates through operations and tries to move them upward or downward as far as possible. When moving an operation, scheduler looks for the cycle within a basic block where it can place that operation, considering all data and control flow dependencies. It can be an empty cycle, but also a non-empty cycle with operations that can go in parallel with the operation which is being moved. The scheduler needs to have knowledge about which operations can fit into a single instruction cycle and under which conditions. This knowledge is expressed in a way very similar to expression of rewriting rules. Every parallelization rule contains information about a single parallelization case: list of operations that execute in parallel and list of resource constraints that need to be satisfied in order for that parallelization to be possible. If there are a lot of parallelization cases this approach requires a lot of parallelization rules to be written, but, on the other hand, it provides more explicit and precise control of what can go in parallel or not.

One example of parallelization conditions on Coyote DSP would be double memory write, which is possible only if the following conditions are met:

a) It targets two different memory zones (they are called X and Y memory zones)

b) Both sources are of accumulator register type

c) Both memories are accessed using index registers

d) Source writing to X memory zone must be one of the first 4 accumulators, while source writing to Y zone must be one of the second 4 accumulators, and

e) Index register for addressing X zone must be index register 0 or 1, while for addressing Y zone it must be index register 4 or 5.

All of the above conditions, except a), can be expressed by set of resource constraints, which are emitted by the scheduler. Condition a) is already indirectly determined in input program using memory zone qualifiers that are part of Embedded C language extension. Resource constraints can place an operand in a certain resource group (or subgroup), imply that two operands have to be in the same, or different,

resource group, that two operands have to have same resource index within different groups, etc. Every constraint has a priority which can have range of values, but in essence divides constraints in two groups: mandatory and optional. Set of all constraints is the input for resource allocation module.

Resource constraints can be emitted from any place in the compiler, but three main points from which constraints originate are instruction selection, scheduler, and interference graph construction. Instruction selection and interference graph construction generate only mandatory constraints. Constraints related to instruction particularities are emitted from instruction selection module, where each edge in interference graph is represented by one constraint. Scheduler, as mentioned, generates only constraints which, if satisfied, enable a certain instruction level parallelisms to happen. Therefore, resource allocation module might not satisfy all the constraints. After resource allocation, scheduler must reevaluate the code, propose another instruction order and send new constraint set back to resource allocator. This loop can be repeated several times, until all constraints are satisfied. It represent coupling of scheduling and resource allocation phases in the new compiler infrastructure.

Two common architectural elements of DSPs are hardware loop and address generators. Recognizing loops that can be implemented as hardware loops and figuring out efficient utilization of address generators are difficult problems, in general case. In order to successfully utilize those architectural elements for wide range of possible inputs, some advanced algorithms must be used, like [30], and even then there might be cases which are not going to be covered. For hardware loop detection the new compiler covers only loop cases encountered in DSP Stone test set [31]. Algorithm that covers those cases proved to be relatively simple. For every loop in the source C code that is not translated into hardware loop an info message is emitted, telling a programmer the reasons why it did not happen. A programmer can than change the code accordingly, if the loop can be made into hardware loop at all. This mechanism represents tradeoff between optimization complexity and programmer effort. The tradeoff is acceptable if there are not too many cases where programmer needs to spend his effort. This turned out to be true for the new Coyote DSP compiler because when compiling five HD audio post-processing applications used for benchmarking (see Results section) all loops in the code where translated into hardware loops, except those loops that cannot be made into hardware loops anyway. Very similar approach was taken for utilization of address generators. Simpler algorithm is used, covering only some cases (also from DSP Stone test set), but for every memory access that does not use address generator an appropriate info message is emitted. It usually requires programmer to manually change indexing to pointer access, with explicit pointer updated, or to change access order. Unlike hardware loop detection, this optimization failed to utilize address generator in seven places for the same five application benchmark set, but effort of adjusting the code according to info messages was acceptable.

The new compiler also performs three global optimizations. The first one is automatic in-lining, where compiler chooses functions to be in-lined. In-lining aims to remove the barrier to other optimizations, imposed by function call. By in-lining, that barrier is completely removed. When in-lining is not suitable, the second global optimization tries to reduce the calling costs by inter-procedural register allocation. The third global optimization is merging of constants. It counts all constants used in the code and for each constant creates single memory location which will be referenced from every place where that constant is used. For the target processor this optimization is additionally significant because, in general case, it is more efficient to set values in register by loading them from memory then by using immediate operand in instruction. For example, storing 32 bit constant in register requires two instructions that set higher and lower 16 bits by immediate operand value, whereas only one load from memory is needed. Therefore if constant is placed in data memory, as opposed to program memory, the program memory consumption will be decreased even if there is a single use of some constant in the whole code. The user can select memory bank where constants will be placed. All of the global optimizations will give better results when the whole program is passed to the compiler at once, but they do work even if only a part of the program is compiled.

The compiler infrastructure provides the module for cloning IR. Creating a duplicate of current IR state is useful for trying different compilation strategies. In the new Coyote DSP compiler each function is cloned at a certain point in the backend. Compilation strategies vary for each clone, and at the end the best resulting code is kept. This mechanism provides a simple and easy solution for making some difficult decisions in optimization algorithms. Instead of developing or implementing an advanced algorithm that tries to make the best decision (which is often not possible), it is much easier just to try all alternatives. Currently the whole compilation is performed in a single process, but performing of each compilation strategy is independent task and the work can be easily parallelized. Utilization of multi-core host architectures can be achieved relatively easily, and with serialization of IR the work can even be distributed among different computers.

IV. RESULTS

To quantify the improvement, several benchmarks were performed on both baseline compiler and the novel compiler. The main measurement of code quality was code size, whereas code execution speed was not measured. However, bearing in mind that no optimization that favor speed over code size (such as loop unrolling) was being used by any of the two compilers, it can be concluded that code size and execution speed are tightly correlated.

Five HD audio post-processing applications where used for benchmarking: two audio volume post-processing applications, two multichannel virtualization applications, and one complex post-processing application (comprising several different post-processing algorithms).

TABLE I. THE COMPILED CODE SIZES

Application	Baseline compiler	New compiler	Relative improvement
Volume pp 1	1239	954	30%

Virtualization pp 1	549	467	18%
Virtualization pp 2	3259	2791	17%
Volume pp 2	4428	3751	18%
Complex pp	4733	3975	19%

Table I shows code sizes for all five applications with all optimizations turned on in both compilers. It is evident that for the most applications the new compiler generates around 18% smaller code, with the peak improvement of 30% for the first application.

TABLE II. CONTRIBUTION OF GLOBAL OPTIMIZATIONS

Application	Without global optimizations	With global optimizations	Relative improvement
Volume pp 1	997	954	5%
Virtualization pp 1	492	467	5%
Virtualization pp 2	2895	2791	4%
Volume pp 2	3801	3751	1%
Complex pp	4129	3975	4%

Table II shows contributions of global optimizations, namely inter-procedural register allocation, automatic in-lining, and constant merging, with constants being stored to data memory. The contribution is expressed as improvement relative to the code size when all optimizations are turned on in the new compiler. The results show that the global optimizations reduce the code size by around 4 to 5%. For the second volume post-processing application improvement of only 1% is observed, but it is attributed to the fact that the application has smaller number of constants and functions relative to its size.

TABLE III. CONTRUBUTION OF SMALL OPTIMIZATIONS IMPLEMENTED AS REWRITE RULES

Application	Without optimizations by rw rules	With optimizations by rw rules	Relative improvement
Volume pp 1	1067	954	12%
Virtualization pp 1	493	467	6%
Virtualization pp 2	3012	2791	8%
Volume pp 2	4166	3751	11%
Complex pp	4353	3975	9%

In Table III it can be seen how much code size is reduced by optimizations implemented as rewrite rules. Those optimizations include strength reduction, algebraic simplifications, and target specific transformations on HL IR and LL IR. Contribution of those small optimization rules is also expressed relative to code size when all optimizations are turned on. The rules decrease the code size by 6 to 12%. When this result is evaluated in the context of optimizations implementation effort, which is very small when the rules mechanism is used, it can be concluded that implementing these optimizations as rules has good benefit/effort ratio.

TABLE IV. DETAILED RESULTS FOR MULTIPLE COMPILATION OPTIMIZATION

Application	Strategy #1	Strategy #2	Best of both per function
Volume pp 1	998	1008	954
Virtualization pp 1	487	469	467
Virtualization pp 2	2831	2803	2791
Volume pp 2	3792	3801	3751
Complex pp	4020	4010	3975

Contribution of applying multiple compilation strategies in the new compiler is analyzed separately. The compiler was set to use only two different strategies. The difference was in register allocation phase: strategy 1 in Table IV was more focused on removing move instructions, whereas strategy 2 in Table IV was favoring resource constraints that improve instruction parallelism. Table IV shows three numbers for each test application. In the first column there is code size when whole application was compiled with only first of the two strategies. The result when compiling only with the second strategy is given in the next column. The result in the final column represents the code size when for each function in the application, the smallest code from the both strategies is chosen. These results show feasibility of this approach, but there is a lot of room for adding new strategies. That will be the part of the effort to further improve the compiler.

TABLE V. PARALLEL INSTRUCTION PERCENTAGE

Application	Baseline compiler	New compiler	Average parallel instruction percentage for application group
Volume pp 1	8%	14%	15%
Volume pp 2	10%	13%	
Virtualization pp 1	15%	19%	20%
Virtualization pp 2	13%	18%	
Complex pp	17%	21%	not available

Finally, improvement in instruction level parallelization is analyzed. For that purpose percentage of parallel instructions (instructions which have more than one operation) was measured for every application, as it was measured for assembly coded applications in [32]. Table V shows that code generated by the new compiler has higher percentage of parallel instructions. In [32], based on analysis of large set of assembly hand-coded applications for Coyote DSP, it is concluded that parallel instruction percentage varies for different application groups. The fourth column in Table V contains average parallel instruction percentage for virtualization applications and volume application (there were no measurements for complex applications in [32]). The numbers show that the code generated by the new compiler has parallel instruction percentage very close to hand-coded assembly code. It is also evident that the biggest improvement in parallelization is achieved in the first volume post-processing application, which is the biggest reason for the significantly higher overall improvement observed for that application in Table I.

V. CONCLUSION

The proposed compiler infrastructure proved to be adequate for implementing several key techniques, named in this paper, for embedded processors. Very efficient compiler for a particular DSP target was developed based on that infrastructure, but for a more general conclusion some other target processors should be explored.

There is still room for improvement of the described compiler infrastructure. It should mainly go in direction of introducing more code analysis modules (base on embedded system specific static code analysis described in [33]), further exploring coupling of scheduling and resource allocation phase, and introducing more different compilation

strategies in multiple-compilation mode. Besides that, the future research will be aimed at the possibilities of compiler helping programmers to transform floating-point C code to fixed-point code by applying conversion algorithm such as [34], and guiding them to optimize the code by more elaborate performance feedback.

REFERENCES

[1] J. A. Fisher, P. Faraboschi, C. Young. Embedded computing: A WLIW approach to architecture, compilers, and tools. pp. 9-16, Morgan Kaufmann, 2005.

[2] L. H. Hamel, "Industrial strength compiler construction with equations", ACM SIGPLAN Notices, Volume 27, Issue 8, pp. 43-50, 1992, doi: 10.1145/142137.142145

[3] Cirrus Logic 32-bit DSP Assembly Programmer's Guide, Cirrus Logic Inc., 2013. [Online]

[4] JTC1/SC22/WG14, Programming languages - C - Extensions to support embedded processors, Technical report, ISO/IEC, 2006.

[5] R. Leupers, "Code generation for embedded processors", in Proc. 13th International Symposium on System Synthesis (ISSS'00), Madrid, 2000, doi: 10.1109/ISSS.2000.874046

[6] R. Leupers, "Compiler design issues for embedded processors", IEEE Design & Test of Computers, Volume 19, Issue 4, pp. 51-58, 2002, doi: 10.1109/MDT.2002.1018133

[7] M. Wolfe, "How compilers and tools differ for embedded systems", Proc. 2005 international conference on Compilers, architectures and synthesis for embedded systems, New York, 2005. doi: 10.1145/1086297.1086298

[8] G. Fursin, O. Temam, "Collective optimization: a practical collaborative approach", ACM Transactions on Architecture and Code Optimization, Volume 7, No. 4, Article 20, 2010, doi: 10.1145/1880043.1880047

[9] D. R. White, A. Arcuri, J. A. Clark, "Evolutionary Improvement of Programs", IEEE Transactions on Evolutionary Computation, Vol. 15, No. 4, pp. 515-538, 2011, doi: 10.1109/TEVC.2010.2083669

[10] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, J. W. Davidson, "Practical Exhaustive Optimization Phase Order Exploration and Evaluation", ACM Transactions on Architecture and Code Optimization, Vol. 6, Issue 1, Article 1, 2009, doi: 10.1145/1509864.1509865

[11] S. Bashford, R. Laupers, "Phase-Coupled Mapping of Data Flow Graphs to Irregular Data Paths", Design Automation for Embedded Systems, Volume 4, Issue 2-3, pp. 119–165, 1999, doi: 10.1023/A:1008966522714

[12] S. Rajagopalan, S. P. Rajan, S. Malik, S. Rigo, G. Araujo, K. Takayama, "A retargetable VLIW compiler framework for DSPs with instruction-level parallelism", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 20, Issue 11, pp. 1319-1328, 2001, doi: 10.1109/43.959861

[13] M. Eriksson, C. Kessler, "Integrated Code Generation for Loops", ACM Transactions on Embedded Computing Systems, Vol. 11S, Issue 1, No. 19, 2012, doi: 10.1145/2180887.2180896

[14] G. W. Grewal, C. T. Wilson, "Mapping reference code to irregular DSPs with the retargetable, optimizing compiler COGEN(T)", Proceedings 34th ACM/IEEE International Symposium on Microarchitecture, pp. 192-202, 2001, doi: 10.1109/MICRO.2001.991118

[15] Y. Choi, T. Kim, "Address assignment in DSP code generation - an integrated approach", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 22, Issue 8, pp. 976-984, 2003, doi: 10.1109/TCAD.2003.814955

[16] G. Talavera, M. Jayapala, J. Carrabina, F. Catthoor, "Address Generation Optimization for Embedded High-Performance Processors: A Survey", Journal of Signal Processing Systems, Vol. 53, Issue 3, pp. 271-284, 2008, doi: 10.1007/s11265-008-0165-y

[17] V. Bertin, J. Daveau, P. Guillaume, T. Lepley, D. Pilat, C. Richard, M. Santana, T. Thery, "FlexCC2: An Optimizing Retargetable C Compiler for DSP Processors", Lecture Notes in Computer Science Volume 2491, pp. 382-398, 2002. doi: 10.1007/3-540-45828-X_28

[18] D. Ebner, F. Brandner, B. Scholz, A. Krall, P. Wiedermann, A. Kadlec, "Generalized instruction selection using SSA-graphs", ACM SIGPLAN Notices - LCTES '08, Vol. 43, Issue 7, pp. 31-40, 2008, doi: 10.1145/1379023.1375663

[19] G. Chen, M. Kandemir, "Optimizing embedded applications using programmer-inserted hints", Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific, Vol. 1, pp. 157-160, 2005, doi: 10.1109/ASPDAC.2005.1466149

[20] M. Deilmann, A Guide to Vectorization with Intel C++ Compilers, Intel Corporation, pp. 20-21, 2012. [Online]

[21] A. W. Appel. Modern compiler implementation in C, Second edition, pp. 219-240, Cambridge University Press, 2004.

[22] C. W. Fraser, R. R. Henry, T. A. Proebsting, "BURG – Fast optimal instruction selection and tree parsing", ACM SIGPLAN Notices, Volume 27, Issue 4, pp. 68-76, 1992, doi: 10.1145/131080.131089

[23] C. W. Fraser, D. R. Hanson, T. A. Proebsting, "Engineering a simple, efficient code generator generator", ACM Letters on Programming Languages and Systems, Volume 1, Issue 3, pp. 213-226, 1992, doi: 10.1145/151640.151642

[24] K. D. Cooper, L. Torczon. Engineering a compiler, pp. 595-604, 569-579 Morgan Kaufmann, 2004.

[25] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: principles, techniques, & tools, Second edition, pp. 549-553, Addison-Wesley, 2007.

[26] T. A. Johnson, S. I. Lee, L. Fei, A. Basumallik, G. Upadhyaya, R. Eigenmann, S. P. Midkiff, "Experiences in Using Cetus for Source-to-Source Transformations", Languages and Compilers for High Performance Computing, pp. 1-14, 2005, doi: 10.1007/11532378_1

[27] M. Djukic, N. Cetic, R. Obradovic, M. Popovic, "An approach to instruction set compiled simulator development based on a target processor C compiler back-end design", Innovations in Systems and Software Engineering: Volume 9, Issue 3, pp. 135-145, 2013, doi: 10.1007/s11334-013-0220-0

[28] D. Guilan, Z. Suqing, T. Jinlan, J. Weidu, "A Study of Compiler Techniques for Multiple Targets in Compiler Infrastructures", ACM SIGPLAN Notices, Volume 37, Issue 6, pp. 45-51, 2002, doi: 10.1145/571727.571735

[29] A. Nicolau, R. Potasman, "Realistic scheduling: compaction for pipelined architectures", MICRO 23 Proceedings of the 23rd annual workshop and symposium on Microprogramming and microarchitecture, Orlando, pp. 69-79, 1990, doi: 10.1145/255237.255252

[30] P. Lokuciejewski, D. Cordes, H. Falk, P. Marwedel, "A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models", Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09), Washington, pp. 136-146, 2009, doi: 10.1109/CGO.2009.17

[31] V. Zivojnovic, J.M. Velarde, C. Schlager, H. Meyer, "DSP-stone: A DSP-oriented benchmarking methodology", Proceedings of International Conference on Signal Processing Applications and Technology, 1994, pp. 715-720.

[32] I. Považan, M. Popovic, M. Đukic, and M. Krnjetin, "Measuring the quality characteristics of an assembly code on embedded platforms", Telfor Journal, Volume 4, No. 1, 2012, doi: 10.1109/TELFOR.2011.6143798

[33] H. M. Kienle, J. Kraft, Thomas Nolte, "System-specific static code analyses: a case study in the complex embedded systems domain", Software Quality Journal, Vol. 20, Issue 2, pp. 337-367, 2012, doi: 10.1007/s11219-011-9138-7

[34] A. Barleanu, V. Baitoiu, A. Stan, "Digital filter optimization for C language," Advances in Electrical and Computer Engineering, Vol. 11, no. 3, pp. 111-114, 2011, doi: 10.4316/AECE.2011.03018