An Executable Intermediate Representation for Retargetable Compilation and High-Level Code Optimization

Rainer Leupers, Oliver Wahlen, Manuel Hohenauer, Tim Kogel
Aachen University of Technology (RWTH)
Integrated Signal Processing Systems
Aachen, Germany
Email: leupers@iss.rwth-aachen.de

Peter Marwedel
University of Dortmund
Dept. of Computer Science 12
Dortmund, Germany
Email: marwedel@cs.uni-dortmund.de

***Abstract*— Due to fast time-to-market and IP reuse requirements, an increasing amount of the functionality of embedded HW/SW systems is implemented in software. As a consequence, software programming languages like C play an important role in system specification, design, and validation. Besides many other advantages, the C language offers executable specifications, with clear semantics and high simulation speed. However, virtually any tool operating on C specifications has to convert C sources into some intermediate representation (IR), during which the executability is normally lost. In order to overcome this problem, this paper describes a novel IR format, called IR-C, for the use in C based design tools, which combines the simplicity of three address code with the executability of C. Besides the IR-C format and its generation from ANSI C, we also describe its applications in the areas of validation, retargetable compilation, and source-level code optimization.**

I. Introduction

The growing importance of the C programming language and its derivatives (e.g. [1], [2], [3]) in embedded system design implies that a large amount of tools for translating C specifications into other formats are required: For instance, compilers for translating C programs into assembly programs, and C based hardware design tools for mapping C specifications into equivalent HDL specifications. In order to perform such translations, it is very common to use a frontend that, as a first step, translates the original C program into a machine-independent *intermediate representation* (IR).

The most widespread IR format is *three address code* [4]. This format consists of a sequence of simple statements, each of which references at most three variables: two arguments and one result. The main motivation for using three address code is its simple structure. As compared to an original C program, all complex arithmetic expressions, nested control flow constructs, as well as implicit address arithmetic for array or structure accesses are broken down into sequences of primitive assembly-like assignments and jumps. In turn, this strongly facilitates the implementation of tools for processing C programs, such as IR optimization passes, compiler backends, or HDL generators. A three address code IR can also be easily translated into *data flow graphs* (DFGs) which reflect potential parallelism and which are the usual input format for code generation and scheduling algorithms.

For the purpose of hardware synthesis from C, the IR generation can be viewed as a *specification refinement* that lowers an initially high-level specification in order to get closer to the final implementation, while retaining the original semantics. We will explain later, how this refinement step can be validated.

However, the *executability* of C, which is one of its major advantages, is usually lost after an IR has been generated. Executability means that a C specification can be compiled into a machine program for a host machine which can be executed on the host for validation or simulation purposes. Normally, this is no longer possible with the IR. Even though the notion of three address code is intuitively clear, there is no standard format for a three address code IR, but the detailed implementation is typically tool-specific.

The purpose of this paper is to present a new IR format, called IR-C, that *retains the executability* of the C language, while simultaneously offering the simplicity of three address code. The key idea in our approach is to *represent the IR itself in C syntax*. This is possible, since the C language allows for an extremely low-level, assembly-like specification of programs. Therefore, IR-C can still be compiled and executed like the corresponding original C code, from which the IR has been generated.

Note that the executability of IR-C is not required for all types of applications. For instance, in a C compiler IR-C can be used just like any other three address code format as a file exchange format, without the need to compile it onto a host platform.

Applications of the proposed IR format include *validation* and *source-level optimization* (which exploit the executability of IR-C) as well as *retargetable compilation*, which is an enabling technology for architecture exploration for application specific processors (ASIPs).

Even though IR-C is a quite general machine-independent format, it is mostly dedicated to the use in embedded system design tools, due to the following reasons:

- The advantages of IR-C and the compilation methodology built around it have to be paid with lower compilation speed than in machine-specific compilers for general purpose processors. However, it is widely accepted that for embedded systems high compilation speed has lower priority than retargetability and high code quality.
- Due to the frequent use of non-standard ASIPs, it is mostly in the area of embedded systems where one has to deal with retargetable compilation and cross-compilation/validation. As will be explained, these tasks are explicitly supported by IR-C.

The structure of this paper is as follows. After a discussion of related work in section II, we describe the global structure of IR-C in section III. In section IV, we outline how the IR-C representation for some input C program is generated by an ANSI C frontend, and we show how the executability of IR-C is exploited for validation. In section V, we demonstrate how IR-C can be used in practice for compilation and code optimization. Finally, section VI gives conclusions.

II. Related work

Most C compilers generate a machine-independent IR before translating code into assembly. However, virtually all such tools use a custom IR format, and to our knowledge an executable three address code IR with the capabilities mentioned above so far has not been implemented.

A well-known example is the frontend that comes with the GNU C compiler GCC [5], whose IR is neither machine-independent nor executable. Additionally, the GCC is restricted to certain processor classes. Likewise, the retargetable compiler LCC [6] comes with a C frontend, but the IR is given in the form of DFGs and hence is not executable. The SUIF-2 system [7] allows for emitting the internal IR in C syntax, but the generated C code is still more complex than three address code. In contrast, the IR format proposed in this paper allows to get closer to the assembly level in a still machine-independent fashion.

Further related work includes source-level (e.g. C-to-C) transformation techniques. The advantage of C-to-C transformations as opposed to assembly-level optimizations is that if a given program is optimized at the C level, it can still be passed to any C compiler for any target machine. Thereby, a high degree of retargetability is ensured. Examples of C-level transformations include the ADOPT approach [8] for address code optimization in multimedia applications, loop transformations [9], array-to-pointer transformations for optimizing DSP applications [10], as well as pattern matching and rewriting tools, e.g. [11]. However, such approaches use to work on higher level IR formats than three address code. Hence, they can be considered complementary to our approach.

III. IR-C structure

This section briefly describes the overall structure of IR-C. We assume that the reader is familiar with basics of the C language. Since our goal is a simple three address code format, all high-level C constructs, such as for and while-loops, nested if-then-else-statements, switch-statements, complex arithmetic expressions and conditionals, and implicit address arithmetic for array and structure access are replaced by sequences of primitive statements in IR-C.

We generate one IR-C file for each given C source file. The IR-C file is structured into *symbol tables* and *functions*. The IR functions directly correspond to the functions in the original C code, i.e., there is one IR-C function for each C function. Each function is a list of *IR statements*, which exist in five different types:

1. Assignments: An assignment is a three address code C statement with a destination and at most one operation or a function call on its right hand side.

2. Jumps: A jump is a C ”goto” statement with a target label.

3. Branches: A branch is a C if-statement of the general form `if (c) goto label`, where the condition `c` is a variable or a constant.

4. Labels: Labels are directly represented as C labels.

5. Return statements: A return from a function call is either a ”void” return statement, or a return with a value of the form `return x;` for some variable or a constant `x`.

By means of the IR generation technique described in the following section, any C program can be translated into a functionally equivalent IR using only the above five statement types. A major point is that the C language allows to express all IR constructs directly in C syntax. Therefore, any valid IR-C code is simultaneously a valid (low-level) C code.

Information about identifiers is kept in symbol tables. Table entries store the type of a symbol, as well as additional information such as its storage class, e.g. `extern`, `static`, or `register`. Like for IR statements, all symbol table information can be expressed in C, by means of usual declaration lists.

An important difference between IR-C and C is that, for sake of a simpler structure, we only allow for two different scopes of identifiers: either global or local. There is one global symbol table, as well as one local table per function. In order to avoid name conflicts between identical identifiers declared within nested local scopes, a unique numerical ID is appended as a suffix to all local symbols. Naturally, this is not done for global variables in order to enable linking between separately compiled C modules.

IV. IR-C generation and validation

While the IR structure is (and should be) very simple, the generation of the IR for a given C program is a more difficult task. The main challenge is to generate an IR which, when compiled with a C compiler, shows exactly the same functionality as the original C program. This guarantees a clean semantics for IR-C, because it is exactly identical to a subset of the ANSI C semantics. In addition, it ensures executability of IR-C, which is useful for validation and source-to-source transformations.

We have developed an ANSI C frontend as a part of the LANCE compiler system [12] that uses a syntax-directed translation mechanism, using an attribute grammar compiling system [13]. For each rule of the C language grammar there is one function that translates a certain C construct into an equivalent sequence of three address code statements. The general concept is simple: complex statements are split into simple ones by insertion of auxiliary variables and appropriate "goto" constructs. However, care must be taken in the ordering of IR statements and translation of implicit address arithmetic. We exemplify IR generation for the following piece of C code:

```
void f()
{
   int i,A[10];
   i = A[2]++ > 1 ? 2 : 3;
}
```

For the IR for function `f`, we need 8 auxiliary variables, denoted as `t1` to `t8`. These are declared in the local symbol table of `f`, together with the original local variables `A` and `i`. The symbol table in C syntax looks as follows:

```
int A[10];
char *t1,*t3;
int i,t2,t5,t6,t7,t8;
int *t4;
```

The first step in IR generation is to compute the value of `A[2]`. If we assume that an integer value occupies four memory words[1], the array index 2 needs to be scaled by 4 and must be added to the base address of array `A` in order to obtain the effective address "`A + 8`" of `A[2]`. When implementing this scheme in three address code, it is important to take into account that in C all constants added to pointers are implicitly scaled, e.g., adding a constant `c` to some integer pointer `p` actually increments `p` by `4 * c`. Therefore, we perform all address arithmetic in IR-C exclusively on `char` pointers, since characters are guaranteed to occupy only a single memory word. This leads to the following IR-C code segment:

```
t3 = (char *)A;   // cast base to char*
t2 = 2 * 4;       // compute offset
t1 = t3 + t2;     // compute effective address
t4 = (int *)t1;   // cast back to int*
t5 = *t4;         // load value from memory
```

The post-increment of `A[2]` is implemented as follows. The address stored in `t4` can be reused. Note that auxiliary variable `t5` still contains the original value of `A[2]`. This is necessary, since its value *before* the increment is required in the comparison.

```
t6 = t5 + 1;     // increment
*t4 = t6;        // store back into A[2]
```

Next, the condition `A[2] > 1` is evaluated and the result is stored in another auxiliary variable `t7`. The conditional expression itself is translated by means of a branch and two labels. Depending on the comparison result, auxiliary variable `t8` is loaded with either 2 or 3. When control flow joins (label `L2`), variable `i` finally gets its correct value from `t8`. In total, the IR for this looks as follows:

```
    t7 = t5 > 1;         // compare
    if (t7) goto L1;     // jump if >
    t8 = 3;              // load 3 if <=
    goto L2;             // goto join point
L1: t8 = 2;              // load 2 if >
L2: i = t8;              // move result into i
```

[1]The size and the memory alignment of all data types are passed as configuration parameters to the ANSI C frontend.

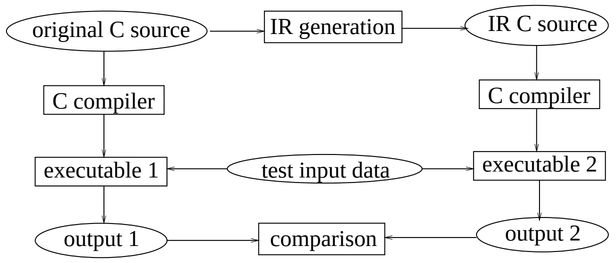


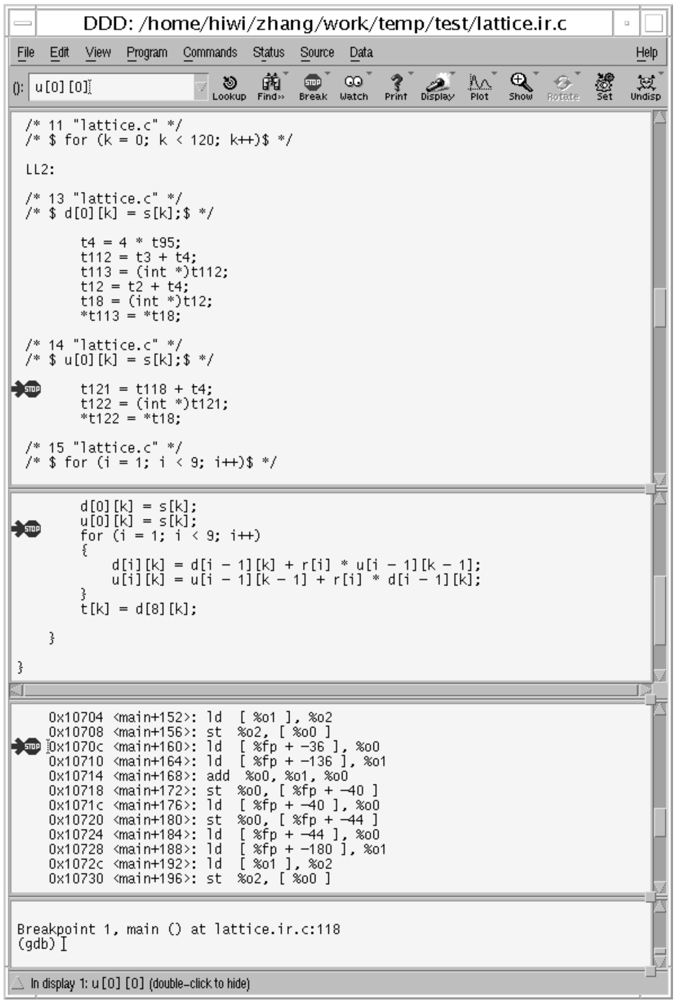
Fig. 1. *Validation methodology*

Using the IR-C intermediate representation allows us to get as close as possible to assembly code, while still retaining machine-independence. Besides the rewriting of high-level control structures and the insertion of auxiliary variables, the IR-C format also explicitly comprises all address arithmetic and cast operations. In addition, the local identifier hierarchy is flattened, local static variables are transformed into unique global variables, and initialization code for local variables is automatically generated. Thus, IR-C already represents much explicit information at a machine-independent level, which any compiler needs to generate.

The C to IR-C translation process can be implemented very efficiently. On a 600 MHz Linux PC, the LANCE ANSI C frontend emits up to 10,000 IR statements per CPU second, including file I/O. Due to the insertion of new variables and statements, the generated IR-C file is typically about twice as large as the original C file. However, this has no practical consequences for IR-C based tools.

Since a C compiler is located at the bottom of the software development tool chain, its correctness is of outstanding importance. In our case, the ANSI C frontend that generates IR-C (as well as all other LANCE tools that manipulate or optimize IR-C) are validated by exploiting the executability of IR-C. Fig. 1 illustrates the methodology. Both the original C program and the IR generated by the frontend are compiled with an existing C compiler for the host machine, which is supposed to be correct. The equivalence of the two executables is checked by means of a comparison between their outputs for a representative set of test input data. Any difference in the outputs indicates an implementation error. For regression tests, this validation process can be easily automated.

Naturally, this approach cannot provide a correctness proof. However, in contrast to custom, non-executable, IR formats, it allows for validating a number of compiler components


DDD: /home/hiwi/zhang/work/temp/test/lattice.ir.c
File Edit View Program Commands Status Source Data Help
(): u[0][0]
Lookup Find» Break Watch Print Display Plot Show Rotate Set Undisp

```
/* 11 "lattice.c" */
/* $ for (k = 0; k < 120; k++)$ */

LL2:

/* 13 "lattice.c" */
/* $ d[0][k] = s[k];$ */

      t4 = 4 * t95;
      t112 = t3 + t4;
      t113 = (int *)t112;
      t12 = t2 + t4;
      t18 = (int *)t12;
      *t113 = *t18;

/* 14 "lattice.c" */
/* $ u[0][k] = s[k];$ */

      t121 = t118 + t4;
      t122 = (int *)t121;
      *t122 = *t18;

/* 15 "lattice.c" */
/* $ for (i = 1; i < 9; i++)$ */
```

```
      d[0][k] = s[k];
      u[0][k] = s[k];
      for (i = 1; i < 9; i++)
      {
          d[i][k] = d[i - 1][k] + r[i] * u[i - 1][k - 1];
          u[i][k] = u[i - 1][k - 1] + r[i] * d[i - 1][k];
      }
      t[k] = d[8][k];

   }

}
```

```
    0x10704 <main+152>: ld  [ %o1 ], %o2
    0x10708 <main+156>: st  %o2, [ %o0 ]
    0x1070c <main+160>: ld  [ %fp + -36 ], %o0
    0x10710 <main+164>: ld  [ %fp + -136 ], %o1
    0x10714 <main+168>: add  %o0, %o1, %o0
    0x10718 <main+172>: st  %o0, [ %fp + -40 ]
    0x1071c <main+176>: ld  [ %fp + -40 ], %o0
    0x10720 <main+180>: st  %o0, [ %fp + -44 ]
    0x10724 <main+184>: ld  [ %fp + -44 ], %o0
    0x10728 <main+188>: ld  [ %fp + -180 ], %o1
    0x1072c <main+192>: ld  [ %o1 ], %o2
    0x10730 <main+196>: st  %o2, [ %o0 ]
```

```
Breakpoint 1, main () at lattice.ir.c:118
(gdb)
```

In display 1: u [0] [0] (double-click to hide)

Fig. 2. *DDD debugger GUI with IR-C debug support*

even without a processor-specific backend and instruction-set simulator.

In practice, a good fault coverage is ensured when using a representative suite of C programs and test inputs. In our case, we used a large test program suite of C applications, including complex program packages like MPEG, JPEG, GSM, BISON, GZIP, a BDD package, a VHDL parser, and a 6502 C compiler. In addition, we have developed a customized interface to the Data Display Debugger DDD [14], a popular graphical frontend to the GNU debugger GDB. This debugger interface allows for monitoring program execution synchronously at the levels of C source, IR-C, and machine code (fig. 2).

V. Applications

A. Retargetable compilation

Apart from a few numerical parameters for type bit width and memory alignment, IR-C is a machine-independent representation. Therefore, IR-C can be mapped into assembly code for different target machines by processor-specific backends. The importance of this concept of *retargetable compilation* [16], [17], [18] in system-level design is well-known. It

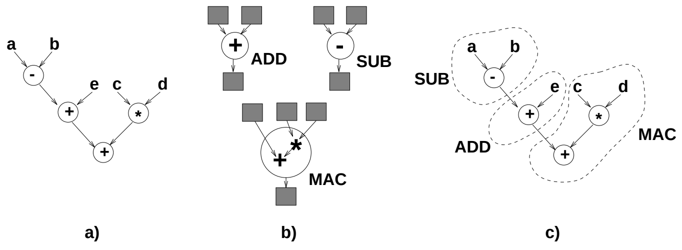


Fig. 3. *DFT covering. a) DFT, b) instruction patterns, c) possible covering*

allows to conveniently study the effects of varying architectural features of an ASIP on code performance, size, and power consumption, thereby enabling processor architecture exploration.

One key issue in retargetable compilers is to maximize the reuse of compiler components, so as to reduce compiler porting effort for new targets. The IR-C tool environment supports this by automatic translation of IR-C into a *machine-independent data flow tree* (DFT) format. DFTs are a special case of general data flow graphs (DFGs), which graphically represent the data dependencies between IR statements. Each edge from node n to m denotes a use of the value generated by statement n in statement m. While three address code IR formats very well support the implementation of machine-independent code optimizations such as constant folding or dead code elimination [21], DFG/DFT formats are generally preferred for backend design, due to their higher expressiveness.

A DFT is a tree-shaped DFG, i.e. a connected DFG without common subexpressions. Due to a lower computational complexity as compared to general DFGs, DFTs are the basic program representation most commonly used in the *code selector* component of a compiler. The code selector maps the machine-independent IR into machine specific assembly instructions. The most popular technique for code selection is *tree pattern matching with dynamic programming* [19]. This can be visualized as a processor of covering a DFT by a minimum cost set of instruction pattern instances (fig. 3).

The IR-C to DFT translator produces maximum size DFTs in a data format fully compatible to widespread *code generator generator* tools like IBURG [20] and OLIVE [17]. As an example, figs. 4, 5, and 6 provide a sample C source, its corresponding IR-C representation, and the generated DFT representation in a textual format. In this way, a retargetable interface to the compiler backend is provided.

The detailed IR to DFT translation is not trivial due to potential undesired side effects during linear ordering of DFTs (e.g. due to function calls modifying global variables, or pointer aliases). Hence, also the DFT generation phase in a compiler should be validated. We achieve this by a tool that emits the DFT representation in C syntax. This permits to reuse the validation methodology from fig. 1 also for the DFT generator.

By generating DFTs in a format accepted by common tools like OLIVE, retargeting time is reduced significantly. The

```
int  A;
void  main( int  a, int  b, int  c)
{  int  x,y,z;
   x = a + b;
   y = b * c;
   z = a − c;
   A += x + y + z;
   return ;
}
```

Fig. 4. *Sample C code*

```
int  A;
void  main( int  a_3, int  b_4, int  c_5)
{  int  x_7,y_8,z_9,t1,t2,t3,t4,t5,t7;
   int  *t6;

         t1 = a_3 + b_4;
         x_7 = t1;
         t2 = b_4 * c_5;
         y_8 = t2;
         t3 = a_3 − c_5;
         z_9 = t3;
         t4 = x_7 + y_8;
         t5 = t4 + z_9;
         t6 = &A;
         t7 = *t6 + t5;
         *t6 = t7;
         return ;
}
```

Fig. 5. *IR-C for code from fig. 4*

```
(WRITE ['t6 = &A;']
 (ADDR ['A' int  ] & 'A'))

(STORE ['*t6 = t7;']
 (READ ['t6' int * ])
 (PLUS ['*t6 + t5' int ]
  (LOAD ['*t6' int ]
   (READ ['t6' int * ]))
  (PLUS ['t4 + z_9' int ]
   (PLUS ['x_7 + y_8' int ]
    (PLUS ['a_3 + b_4' int ]
     (READARG ['a_3' int ] arg no 1)
     (READARG ['b_4' int ] arg no 2))
    (MULT ['b_4 * c_5' int ]
     (READARG ['b_4' int ] arg no 2)
     (READARG ['c_5' int ] arg no 3)))
   (MINUS ['a_3 - c_5' int  ]
    (READARG ['a_3' int ] arg no 1)
    (READARG ['c_5' int ] arg no 3)))))

(VOIDRETURN ['return;'])
```

Fig. 6. *Three DFTs generated for IR code from fig. 5. Symbols in capital letters represent operators matched by the code selector. Brackets contain debug information and node attributes.*

TABLE I
Experimental results for C-to-C optimization

source	orig C	IR-C	perf %	DFT C	perf %
dct	1.03	1.24	-20	1.00	+3
fft	2.90	2.95	-2	2.71	+7
gsm	1.31	1.40	-7	1.30	+1
lattice	1.69	1.92	-14	0.85	+50
me	1.88	2.61	-39	1.29	+31
lms	0.30	0.39	-30	0.40	-33

developer can directly concentrate on the machine-specific code selector when retargeting the compiler to a new machine. Naturally, the design of even more machine-specific compiler components, such as the register allocator, cannot be supported this way, and more effort has to be spent for highly optimizing backends.

B. Source level optimization

Besides compiler development, a further application area of the IR-C format is source-level code optimization. Like in the case of validation, one can exploit the *executability* of IR-C for C-to-C optimization: Since IR-C is emitted as a C subset, any IR-C optimization is a low-level C-to-C optimization by construction, and the output can be passed to any C compiler. While complementary high-level source code transformation tools like the ones mentioned in section II mostly require a high-level IR, the three address code format of IR-C facilitates the design of certain C-to-C transformations that are best implemented at a low level close to assembly code. Simultaneously, we retain the most significant advantage of source-level code optimization, namely machine-independence and retargetability.

In order to demonstrate this, we have performed some preliminary experiments with the IR-C to DFT translation tool described in section V-A. A given original C code is first translated into IR-C code by the C frontend, and the DFG generator transforms the IR into DFT form and emits it in C syntax. Both the original C program and the transformed program are compiled with the same host compiler in order to evaluate the performance.

For several DSP application routines we observed that this transformation process *can increase program performance* as compared to the original C specification. Table I gives results for a discrete cosine transform (dct), Fast Fourier Transform (fft), a routine from a GSM speech encoder (gsm), a lattice filter (lattice), a motion estimator (me), and a least mean square filter (lms). The C programs have been compiled using GNU gcc 2.95.2 on a 600 MHz Linux PC.

Column 2 gives the CPU seconds required by the executables generated from the original C programs. Column 3 gives the performance measured when compiling the generated IR-C code without DFT construction, and column 5 shows the performance for the IR-C code transformed into DFT form as explained in section V-A.

Columns 4 and 6 mention the performance difference of the executables relative to the original C code. As can be seen, the executables generated from the unoptimized IR show a lower performance than in case of the original code. This is presumably due to the fact that the gcc compiler cannot cope well with low-level C code consisting only of three address code statements.

However, the situation changes drastically, when the IR-C code is converted into DFT form. In this case, there is generally a performance increase, as much as 50 % as compared to the original C code.

Obviously, the reason for this improvement is that restructuring the original C code into a canonical form of maximally large DFTs opens up better optimization opportunities for the C compiler. For instance, the larger the DFTs the more effective is the DFT covering based code selection technique mentioned in section V-A (fig. 3).

As indicated by the lms example in table I, unfortunately a performance increase is not guaranteed by this methodology. Due to its dependence on the source program structure, it can also cause a performance loss. Further investigations are required to predict performance gains or losses due to the C to IR-C/DFT conversion.

VI. Conclusions

The growing importance of the C language in embedded system design implies a growing need for tools capable of processing C specifications. Three address code is a very common IR format for such tools, but it is normally not executable like the original C source. This limitation is removed by the executable IR-C format proposed in this paper. Its applications include validation of frontends and IR-level optimization passes, retargetable compilation, as well as source-level code optimization.

Based on IR-C, the stable compiler development tool chain LANCE has been implemented. It comprises an ANSI C frontend, a library of IR optimizations (including both classical "Dragon Book" techniques [21] and several advanced optimizations from [4]), as well as the backend interface mentioned in section V-A. Applications include industrial C compilers for a Network Processor from Infineon Technologies and Systemonic's OnDSP^{TM} platform, an energy-conscious compiler for the ARM7 RISC [22], as well as numerous research compiler prototypes.

The major benefit from IR-C in this context is the support for validation and retargetable compilation, which largely reduces compiler development effort. Development of operational (not yet heavily optimizing) compiler backends for new target machines typically takes about 6 man-months. Implementation and validation of a new IR-C optimization module takes only 1-2 man-months.

The suite of tools built around the proposed IR-C format is continuously being extended. The application of IR-C for source-level (C-to-C) optimizations is still at an early stage and will be further investigated.

References

[1] G. Arnout: *SystemC Standard*, Asia South Pacific Design Automation Conference (ASP-DAC), 2000

[2] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao: *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, 2000

[3] C Level Design Inc.: *http://www.cleveldesign.com*

[4] S.S. Muchnik: *Advanced Compiler Design & Implementation*, Morgan Kaufmann Publishers, 1997

[5] Free Software Foundation: *http://www.gnu.org*

[6] C. Fraser, D. Hanson: *A Retargetable C Compiler: Design And Implementation*, Addison-Wesley, 1995, *http://www.cs.princeton.edu/software/lcc*

[7] The Stanford Compiler Group: *http://suif.stanford.edu*

[8] S. Gupta, R. Gupta, M. Miranda, F. Catthoor: *Analysis of High-Level Address Code Transformations for Programmable Processors*, Design Automation & Test in Europe (DATE), 2000

[9] H. Falk, P. Marwedel: *Control Flow driven Splitting of Loop Nests at the Source Code Level*, Design, Automation and Test in Europe (DATE), 2003

[10] C. Liem, P.Paulin, A. Jerraya: *Address Calculation for Retargetable Compilation and Exploration of Instruction-Set Architectures*, 33rd Design Automation Conference (DAC), 1996

[11] M. Boekhold, I. Karkowski, H. Corporaal: *Transforming and Parallelizing ANSI C Programs Using Pattern Recognition*, High Performance Computing and Networking Conference, 1999

[12] LANCE compiler: LS12-www.cs.uni-dortmund.de/lance

[13] K.M. Bischoff: *Design, Implementation, Use, and Evaluation of Ox: An Attribute-Grammer Compiling System based on Yacc, Lex, and C*, Technical Report 92-31, Dept. of Computer Science, Iowa State University, 1992

[14] Data Display Debugger (DDD): *http://www.gnu.org/software/ddd*

[15] V. Zivojnovic, J.M. Velarde, C. Schläger, H. Meyr: *DSPStone – A DSP-oriented Benchmarking Methodology*, Int. Conf. on Signal Processing Applications and Technology (ICSPAT), 1994

[16] C. Liem: *Retargetable Compilers for Embedded Core Processors*, Kluwer Academic Publishers, 1997

[17] A. Sudarsanam: *Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors*, Ph.D. thesis, Princeton University, Department of Electrical Engineering, 1998

[18] R. Leupers, P. Marwedel: *Retargetable Compiler Technology for Embedded Systems – Tools and Applications*, Kluwer Academic Publishers, 2001

[19] A.V. Aho, M. Ganapathi, S.W.K Tjiang: *Code Generation Using Tree Matching and Dynamic Programming*, ACM Trans. on Programming Languages and Systems 11, no. 4, 1989

[20] C.W. Fraser, D.R. Hanson, T.A. Proebsting: *Engineering a Simple, Efficient Code Generator Generator*, ACM Letters on Programming Languages and Systems, vol. 1, no. 3, 1992

[21] A.V. Aho, R. Sethi, J.D. Ullman: *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, 1986

[22] S. Steinke, N. Grunwald, L. Wehmeyer et al.: *Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory*, International Symposium on System Synthesis (ISSS), 2002