Advanced Code Generation for Network Processors with Bit Packet Addressing

Jens Wagner

University of Dortmund
Computer Science XII
Embedded Systems Group
email: Jens.Wagner@uni-dortmund.de

Rainer Leupers

Aachen University of Technology
Dept. of EE & IT
Integrated Signal Processing Systems
email: leupers@icd.de

Abstract

One important problem in code generation for embedded processors is the design of efficient compilers for ASIPs with application specific architectures. Network processors are highly optimized on special applications. A distinctive feature of network processors are additional instructions for fast protocol analysis and processing. Many protocols in telecommunication frequently make use of bit-stream oriented data formats. The corresponding bit-streams consist of packets of different length. The number of ASIPs which support subword level instructions to accelerate protocol processing is growing. In this paper we will show a novel code generation technique to identify subword-level operations. Furthernore we will show a fast algorithm to find an optimal instruction selection on subword-level instruction for a tree-based data-flow. In our results we will outline that the difference to hand optimized code is rather low on practical benchmarks. The compiler is in an early working stage and first experimental results indicate that C-level programming of the ASIP leads to good code quality without the need for time-consuming assembly programming.

1 Introduction

For modern general purpose processors most of the software development work is done in high-level languages (HLLs). High-level languages are platform independent, therefore software is portable between different architectures at low cost. The GNU C compiler for example supports nearly 100 different architectures and operating systems [1, 2]. Many software projects can be ported between different platforms simply by re-compiling them. The availability of compilers is a main feature of a general purpose system. Therefore compilers are often assisted by processor features, e. g. the instruction set of the Intel Pentium processor family supports function frame construction and destruction by the instruction set [3]. In other words the machine language supports the semantic of the high level language. Because development of compilers and processors goes hand in hand, the availability of compilers for such systems is very high. If an application is developed in a high level language once, it can easily be ported to another platform. If the software contains hardware dependent code, in most cases, only minor changes will be needed. An example for portable software is the Linux operating system. Many applications are portable between different architectures supported by Linux simply by re-compiling the source of the application.

Contrary to general purpose processors there are application specific processors (ASIPs). These processors are optimized for a special application class which gives advantages in speed, power consumption and/or price. Digital signal processors (DSPs) are optimized to process discrete mathematical algorithms in real-time. A DSP is equipped with special data paths, instruction sets and peripheral units. For example the Micronas DSP MAS 3509F is optimized to a single application, MPEG 2 audio layer decoding [4].

However compilers for such ASIPs have been successfully developed for many years. Specific optimizations focus on irregular data paths [5, 6, 7, 8, 9], address code optimization for DSPs [10, 11, 12, 13], and exploitation of multimedia instruction sets [14, 15, 16]. Experiments have shown that such highly machine-specific techniques are a promising approach to generate high-quality machine code, whose quality often comes close to hand-written assembly code. Naturally, at the cost of increased compilation time in most cases.

However, there are other important classes of ASIPs other than DSPs where so far tool support is relatively low. ASIPs for bit-serial protocol processing, which are called *Network Processors* (NPs) are used to process high speed communication protocols. Network processors are

designed to be used in devices such as routers and switches for IP traffic as well as in telecommunication (e. g. ISDN and xDSL). A major system design problem in this area is that the required high bandwidth leaves only a very short time frame (as low as a few nanoseconds) for processing each bit packet arriving at a network node [17]. Even contemporary high-end programmable processors can hardly keep pace with the required real-time performance, not to mention the issue of computational efficiency, with respect to power consumption. In [17] it is outlined that for packet handling in high speed networks only a small time frame exists. Even at the highest CPU cycle rates only a few instructions can be executed per packet. For instance a 1 GHz processor applied to a 10 GBit core router can only execute 10 to 50 machine instructions per packet.

Because the instruction set of an ASIP is completely constructed for a single application domain, its high-level language support is usually low. The instruction-set of an ASIP very often contains assembly commands of such a high complexity that a number of HLL-instructions are needed to describe them. An example for such machine instructions is the support of *bit-packets*. Serial communication protocols mostly employ bitstream-oriented data formats. The bitstreams consist of *packets* of different length, i.e. there are variable length header packets and (typically longer) payload packets. Typical packet processing requirements include decoding, compression, encryption, or routing.

For a compiler, there are two different ways to handle this difficulty: The complexity of the HLL can be enhanced. For this purpose, application domain specific instructions are added. For example in [18] this technique is used to handle bit-packed addressing from C. The disadvantage of this solution is, that the code reusability is very restricted. Another way is to apply algorithms to the compiler to match the behavior of the ASIP instruction set to a HLL program. The advantage of the second way is that the HLL is unchanged and therefore a main part of the software may stay unchanged if it needs to be ported to another system. Moreover, the developer of the HLL software needs less knowledge about the internal behavior of the target processor. The information about the customized hardware of the ASIP is transfered from the HLL into the compiler. Of course such a compiler is often hand tailored to a special processor and is in itself not easy retargetable. In this paper, we focus on a compiler technique to support bit-packet addressing. We will illustrate this technique on samples taken from GSM-kernel of the TU-Berlin [19].

We implemented the algorithm for the instruction set of the Infineon NP network processor [20]. Figure 1 shows a block diagram of the NP. The NP processor supports instructions which allow subword access, e. g. move, load and storage of *bit packets*, for high efficient protocol processing code. The NP instruction set permits the performance of these operations on bit packets not aligned at the processor word boundaries. A packet may be stored in any bit index subrange of a register.

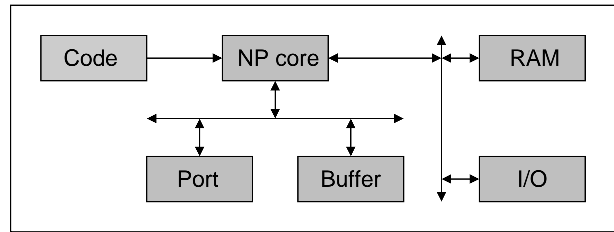

Figure 1. *Infineon NP architecture*

In order to enable packet-level addressing of unaligned data, the NP instruction set permits the specification of offsets and operand lengths. The general instruction format is as follows:

```
CMD addr1.offset, addr2.offset, width
```

Code selectors based on integer size data-flow analysis make only rare use of this class of instructions. The aim of this paper is that we show a fast algorithm to detect bit-packet operations. In a post-pass phase we will insert bit-packet instructions into the source code which make numerous instructions obsolete. A following dead-code elimination phase will remove them.

We implemented an experimental version of this optimization and integrated it into an existing fully functional compiler for a Infineon NP network processor. We benchmarked our compiler with small program fragments. We also compared the resulting code with hand-optimized code.

The remainder of the paper will be structured as follows: In Section 2 we will give an overview of the used algorithms. In Section 3 we describe our implementation and its location in the existing compiler. In section 4 we present our results and in section 5 we outline our conclusions and describe ideas for further work.

2 Algorithm Outline

2.1 Bit-level data-flow analysis and bit-value inference

A given HLL program can be represented as a Data-Flow Graph (DFG). A DFG contains all load, store, move and data-manipulating operations, but no control-flow related aspects. A DFG is defined as a directed acyclic graph $G = (V, E)$, where each node represents an input, output or an opration. Each edge represents a definition-use (DEF-USE) dependency.

A standard representation for the dependencies between statement are DEF-USE chains. DEF-USE chains can be

directly derived from the DFG [21]. Each machine operation of an instruction set should be defined as an instruction name and it's parameters. A parameter of an instruction has one of the following possibilities: a constant number, the definition of a register content (DEF), the usage of a register content (USE), or both – the instruction uses and defines the same register parameter (USEDEF).

Related to the DFG each used operand of a machine instruction may have one or multiple definitions and each definition of an operand may have one or multiple uses. In very unusual cases a USE without a DEF can happen, if side-effects are used: for instance a push of any register is used to increment the stack-pointer. Many instruction sets contain implicit operands, e. g. flag manipulation with an addition. If an instruction contains only definitions which are never used, then the instruction is dead and can be removed from the program. DEF-USE chains are working on whole registers, i. e. DEF-USE dependencies are calculated on the full register bit-width. In many cases the results are suboptimal. Figure 2 shows an example where the fact that variable a does not affect the calculation of c would not be detected by the 'DEF-USE chain'-approach.

```
a = (*p1 & 0x0F);
b = (*p2 & 0xF0)|0x0F;
c = a | b;
```

Figure 2. *Bitoriented data-flow*

An exact bit-level data-flow analysis would give a much better result. In recent years this fact has been analyzed in different works. Yasuura, Tomiyama and Inoue did an analysis of the required register-bank bit-width for a given application [22]. If we expect that on a special ASIP only a single application is executed and we can calculate the complete bit-true data-flow of the application further, it is possible to reduce the size of the used data-types. Therefore the power consumption of the CPU data-paths and operational units can be reduced. However this approach interacts with the user to determine the type information. In this paper we will focus on algorithms without user interference.

Tool-based analysis is done in other work on bit-level data-flow analysis. Brooks and Martonosi [24] who replaced the types of variables, from integer to short to improve speed and to reduce the power consumption of a Pentium MMX processor. Also impressive results are shown from Budiu and Goldstein [25]. They pointed out that 20 % of all most significant bits of MediaBench and SpecINT95 are calculated unnecessarily. In both approaches a bit-value lattice is used. The order of the lattice is the information content of a bit. The elements are: U for *no information*, 0 and 1 if a bit is a constant and X for *don't care*, if the value of a bit does not change the result of the operation. These elements form a lattice (Figure 3). Bit-value lattices are extended to strings by concatenation.

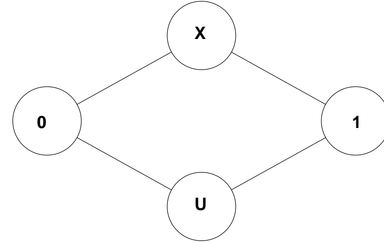

Figure 3. *Lattice by Budiu and Goldstein*

We will extend the bit-value inference approach by adding another two elements to the lattice: $\langle U_n \rangle$, *the content is unknown but we know the location* and $\langle \bar{U}_n \rangle$, *the content is unknown but we know it is the negation of a given location*. Figure 4 shows our lattice $\mathcal{L}$. The lattice forms the following semi order:

$$\langle U \rangle \leq \langle U_n \rangle \leq \langle \bar{U}_n \rangle \leq \langle 0 \rangle \leq \langle 1 \rangle \leq \langle X \rangle$$

X is the top element ($\top$) and U the bottom element of the lattice. This lattice gives us more information than the version from [25]. We will not give formal proof here, but a practical explanation later. Additionally, we need the location information later for the code selection phase in our compiler.

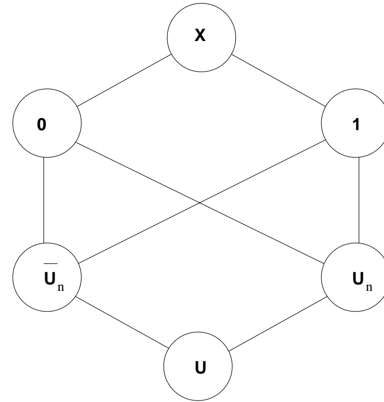

Figure 4. *Extended lattice*

The bit-inference is coupled to the edges. For data-flow analysis we need to calculate how an operation changes the information content of a bit (or a string of bits). Therefore we define two transfer functions for each machine operation. One transfer function in the direction of the edges (↓) which give more information about the result of an operation (the defined parameters of a machine instruction) and one transfer function opposite to the direction of the

edges (↑), which enhances the operators or used parameters. Figure 5 gives an example of a transfer function. To get as much information from the DFG as possible we need to walk along the edges until we reach a fix-point, where we cannott add information to the graph. For a loop-free DFG this is reached after each edge is changed by a single walk-down and walk-up. However, even for nested loops the information about the bit-content can only be improved four times, because our lattice grid has a height of four ($|\mathcal{L}| = 4$)). Therefore in the worst case $|\mathcal{L}| * 2$ changes will be needed until we cannot add any more information to a bit. We call this the fix-point. After $|\mathcal{L}| * 2 * |E|$ iterations any lattice-bit reaches the top element ($\top$). To meet this low complexity we accept only changes where at least a single bit in the DFG enhances its information content. Otherwise it would be possible to construct a DFG where the analysis does not terminate. Figure 7 shows a simple Example for a DFG and figure 6 shows the corresponding C program.

$\downarrow$	$\langle X\rangle$	$\langle 1\rangle$	$\langle 0\rangle$	$\langle U_n\rangle$	$\langle \bar{U}_n\rangle$	$\langle U\rangle$
$\langle X\rangle$	$\langle X\rangle$					
$\langle 1\rangle$		$\langle 1\rangle$	$\langle 1\rangle$	$\langle 1\rangle$	$\langle 1\rangle$	$\langle 1\rangle$
$\langle 0\rangle$		$\langle 1\rangle$	$\langle 0\rangle$	$\langle U_n\rangle$	$\langle \bar{U}_n\rangle$	$\langle U\rangle$
$\langle U_n\rangle$		$\langle 1\rangle$	$\langle U_n\rangle$	$\langle U_n\rangle$	$\langle 1\rangle$	$\langle U\rangle$
$\langle \bar{U}_n\rangle$		$\langle 1\rangle$	$\langle \bar{U}_n\rangle$	$\langle 1\rangle$	$\langle \bar{U}_n\rangle$	$\langle U\rangle$
$\langle U\rangle$		$\langle 1\rangle$	$\langle U\rangle$	$\langle U\rangle$	$\langle U\rangle$	$\langle U\rangle$

Figure 5. *Transfer-function (↓) of an OR-Operation in the domain of $\mathcal{L}$*

```
a = (b & 0x0F);
```

Figure 6. *Simple DFG example*

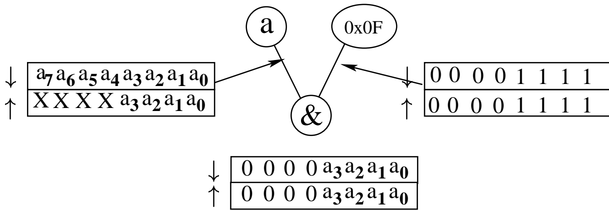

Figure 7. *DFG representation of fig. 6*

In the worst case, for a string S of bits with a length of $|S|$ a change can happen to a single bit only once at each iteration. A change is only valid to a graph if at least a single edge enhances its information content. In worst case a walk over all edges enhances each time a single bit in a single edge. That means the algorithm walks along the edges $|\mathcal{L}| * |S| * 2 * |E|^2$ times till a fix-point solution is found, which gives a complexity of $O(n^2)$ where n is the number of the edges. We will use the information we get from a bit-true data-flow analysis later to enhance the code quality of a given instruction sequence.

2.2 Code selection

Code selection uses the data-flow analysis information of a Basic Block (BB) [23], where a BB is a sequence of statements with one entry and one exit point. Figure 8 shows a BB of C-statements and figure 9 the appropriate DFG.

```
a = *p + 4;
b =  a * 3;
c =  a * 7;
```

Figure 8. *C example for a basic block*

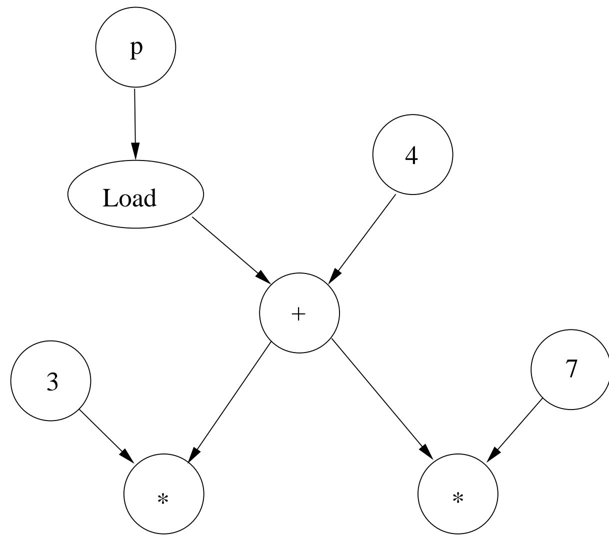

Figure 9. *Example DFG*

Each machine operation of a given instruction set can be considered a small pattern. Code selection can be conceived as a complete coverage of a DFG with instruction pattern. This code selection technique is called *pattern matching*. Optimal pattern matching on graphs is NP-hard [26]. The common solution for this problem is to break up the graph into a forest of trees and to perform the pattern matching on each tree separately. Figure 10 shows the resulting trees for the above example. Ideal tree pattern matching can be done in linear time to the number of nodes and a constant factor given by the grammar definition, but may cause additional machine instructions compared to ideal graph based code-selection. However, for existing homogenous machines, the code quality is similar.

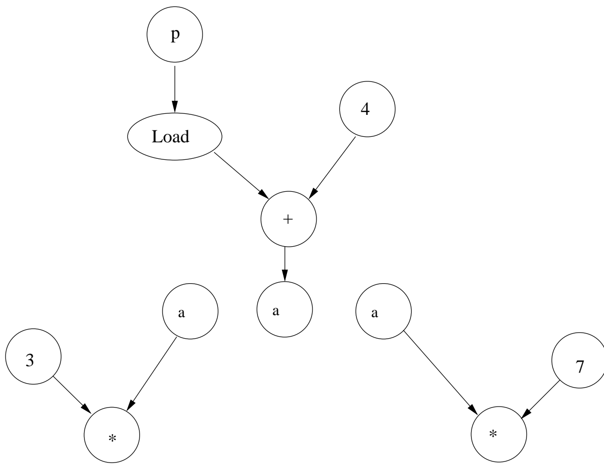

Figure 10. *Decomposed forest of data-flow trees*

In this paper we use the *iburg* tree parser generator [27, 28]. The iburg-toolset produces from an extended Backus-Naur grammar specification a tree-parser C source code as output. Figure 11 shows a short form example of an iburg grammar.

The tree-pattern matching algorithm is based on two phases: A bottom-up phase, where for each node all possible match patterns are calculated. For each node iburg is calculating the covering at minimum cost. The second phase is top-down, where the machine code is emitted. Every node of the tree is visited twice by the algorithm. Therefore the complexity is linear to the number of nodes and a constant factor given by the grammar. In this paper we will use tree-pattern matching to post-optimize the assembly output of our compiler.

2.3 Ertl's tree-pattern matching grammar-composition for free arithmetical simplification

Ertl shows in [29] that optimizations can be for free if a given tree-pattern matching grammar is extended by additional non-terminals and the according rules. The artifice behind this is that an operation is not emitted in place, but remembered by an additional non-terminal. Because other operations on the way up the tree may make the operation obsolete or can do the same computation with less costs. The example in figure 12 shows an Ertl composition. Rules 4 to 7 are the necessary rules to match Negation, the And and the Or expressions (˜, &, |). Rules 8 to 13 are an Ertel composition. Where rules 8 and 13 remember the bitwise negation of a register for later application in the additional non-terminal *negreg*. Rules 8 and 9 proceed the negation by transforming a *negreg* into a *reg* and vice-versa. At this point they are not effected by our grammar extension because rules 8 and 11 are exactly equivalent to rule 5 and also rules 9 and 10 are equivalent to 5. But rules 12 and 13 develop *De Morgan's Theorem* (Equation 1) as an optimization to the grammar. Equation 2 is an example for an application of rule 12. And equation 3 applies rule 13. Another application of rule 13 is equation 4.

Figure 13 outlines a data-flow tree which is covered by usage of rule 13. On the left hand side a standard grammar needs the cost of twice the rule 5 and once the rule 6, which gives overall cost of 3. On the right hand side rules 8, 9, 11 and 12 are used which gives costs of 2, because rules 8 and 9 are for free. Figure 14 displays the resulting asm-code with and without usage of the Ertl composition in Figure 12 which saves one of three machine operations. Ertl [29] also shows examples for advanced constant folding, flag optimization and optimization of unary operators. In some cases Ertl gives some impressive results of 50% code-size reduction for small trees.

$$\overline{a \vee b} = \bar{a} \wedge \bar{b}$$
$$\overline{a \wedge b} = \bar{a} \vee \bar{b} \quad (1)$$
$$\bar{a} \vee \bar{b} = \overline{\overline{\bar{a} \vee \bar{b}}} = \overline{\bar{\bar{a}} \wedge \bar{\bar{b}}} = \overline{a \wedge b} \quad (2)$$
$$\bar{a} \wedge \bar{b} = \overline{\overline{\bar{a} \wedge \bar{b}}} = \overline{\bar{\bar{a}} \vee \bar{\bar{b}}} = \overline{a \vee b} \quad (3)$$
$$\overline{\bar{a} \wedge b} = \bar{\bar{a}} \vee \bar{b} = a \vee \bar{b} \quad (4)$$

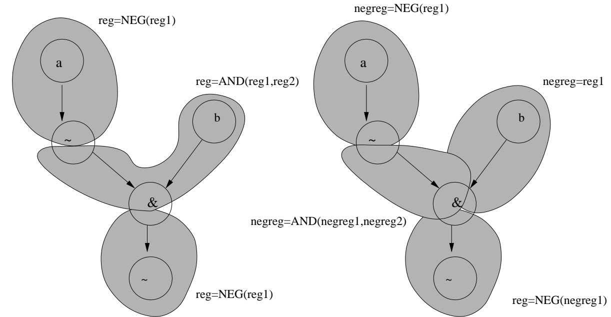

Figure 13. *Tree pattern matching using standard grammar (left) and standard grammar extended by Ertl's composition (right)*

2.4 Dynamic cost functions

To force a selection between a number of rules which match to the same pattern in run-time. iburg allows *dynamic cost functions*. Till now we assumed the cost of a pattern to be constant. But iburg allows the costs of a pattern to be calculated. Hence the costs of a pattern can be dependent on it's parameters.

```
#No non−terminal: Production                #Cost# Assembly
 1     reg        : Load(reg1)              #    1# LD R−dest, ∗R−source
 2     reg        : Add(reg1, const)        #    1# Add R−dest,R−source,const
 3     reg        : Mul(reg1, const)        #    1# Mul R−dest,R−source,const
```

Figure 11. *Example of a short-form tree-parsing grammar*

```
#No non−terminal: Production       #Cost# Assembly
 4     reg  : REG                  #   0 #
 5     reg  : NEG(reg1)            #   1 # NEG(reg, reg1)
 6     reg  : AND(reg1, reg2)      #   1 # AND(reg, reg1, reg2)
 7     reg  : OR(reg1, reg2)       #   1 # OR(reg, reg1, reg2)
#Ertl composition
 8 negreg  : NEG(reg1)             #   0 #
 9     reg  : NEG(notreg1)         #   0 #
10 negreg  : reg1                  #   1 # NEG(negreg, reg1)
11     reg  : notreg1              #   1 # NEG(reg, negreg1)
12 negreg  : AND(notreg1,notreg2)  #   1 # OR(negreg,negreg1, negreg2)
13 negreg  : OR(notreg1,notreg2)   #   1 # AND(negreg,negreg1, negreg2)
```

Figure 12. *Example of a short-form tree-parsing grammar*

```
#No non−terminal: Production#Cost                        # Assembly
 14 reg  : OR(reg1, reg2)     #(reg1==0||reg2==0)?0:∞; #
 15 reg  : OR(reg1, reg2)     #                       1 # OR(reg1, reg2)
```

Figure 15. *Example of a dynamic cost function*

```
NEG(R1,a)              NEG(R1,b)
AND(R2,R1,b)           OR(R2,a,R1)
NEG(R3,R2)
```

Figure 14. *Resulting assembly code of the DFT (figure 13) using standard grammar (left) and grammar extended by Ertl's composition (right)*

Figure 15 shows a constant folding example. If one parameter of the Or-operation is equal to zero the pattern 14 with no code emission is selected, but if both parameters are unequal to zero the pattern 15 is chosen. Of course, this is only a simple example for dynamic cost-functions. In section 2.5 we will use much more advanced functions.

2.5 Bit-packet detection and integer size instruction substitution

In this section we outline how extended bit-level data-flow analysis, Ertl's composition on tree-pattern matching and dynamic cost-functions work together in an algorithm to find a replacement for a given sequence of statements by a special bit-packet operation.

We assume that all parameters and the result (all edges) of a pattern are labeled with a lattice-string of a previous data-flow analysis. We introduce the following non-terminals to distinguish between three different types of parameters:

bf: A Bit-Field, the parameter contains a number of consecutive bits of a source (a register). All other bits of the parameter are constant or *don't care* ($\langle X\rangle, \langle 0\rangle, \langle 1\rangle$).

conBF: The parameter can be assumed as a concatenation of two bit-fields or the insertion of a bit-field into an integer. The result is a register which contains a bit-field and the remaining part of the integer.

reg: A Register can be assumed as an complete integer with any value including the other two cases.

To detect these non-terminals we define two functions which will be used in dynamic cost-functions: *isBF(reg)* which returns **true** if the register is a bit-field and *isConBF(reg)* which returns **true** if the register is a concatenation of two bit-fields as it is defined above.

Figure 16 illustrates a part of a tree-pattern matching grammar which detects a bit-field insertion into an integer. Rules 16, 17 and 18 match the isolation of the bit-field and remembers the bit-field in the *bf* non-terminal. Rule 19 gives an example for the insertion of a bit-field into an integer. To enhance code quality "no code" is emitted in this

```
t1 = (a & 1)<<3;
t2 = t1 | ((unsigned)b >> 5);
t3 = t2 << 3;
t4 = ((unsigned)b >> 2) & 7;
r  = t3 | t4;
```

Figure 17. *C-example for bit-packet insertion*

```
AND (t1, a, 1)
SHL (t1, t1, 3)       ;t1 = (a & 1)<<3

SHR (t2, b, 5)
AND (t2, t2, 0x7)     ;((unsigned)b >> 5)

OR  (t2, t1, t2)      ;t2=t1|((unsigned)b>>5)
SHL (t3, t2, 3)       ;t3=t2 << 3

SHR (t2, b, 2)
AND (t2, t2, 0x7)     ;((unsigned)b >> 2) & 7

OR  (r, t3, t4)       ;r  = t3 | t4
```

Figure 18. *Assembly-code example for bit-packet insertion*

rule. If the result is needed somewhere, as a register, rule 20 would be activated where a bit-field move instruction is inserted into the assembly code.

Figure 17 shows a typical bit-packet insertion in ANSI-C. The resulting assembly-code is outlined in figure 18. If we use a bit-true data-flow analysis we detect that the result r is built from 3 bit-packets: the constant 0 at position 7, the bit-packet a_0 at position 6 and the bit-packet $b_7 \dots b_2$ at position $5 \dots 0$. Therfore only 3 operations (instant of 9) are needed: r needs to be set to the constant, the insertion of a_0 and the insertion of $b_7 \dots b_2$.

Figure 19 outlines the data-flow analysis.

3 Implementation

To test our optimization we extent the already existing *Infineon NP C compiler* described in [18] with our optimization. Like almost all compilers our compiler is subdivided into a frontend and a backend. As a frontend, we use the LANCE compiler system developed at the University of Dortmund. This front-end is widely device independent and is used in many compiler projects [30]. Our backend, build on a widely retargetable tool-suite, is described in [31]. Because our backend is based on an intermediate representation, where tools like code-selection, optimizations and register-allocation are used as plug-ins, we are easily

```
#No non−terminal: Production #                    Cost# Assembly
16  bp      : reg               #                     0 #
17  bp      : AND(bp, const) #     (isBF(bp))?0: ∞; #
18  bp      : SHL(bp, const) #                     0 #
19  conBP   : OR(bp, bp)      # (isConBF(conBP))?0: ∞; #
20  reg     : conBP             #                     1 # movBP(reg,src1,off1,src2,off2,width)
```

Figure 16. *Part of a tree-pattern matching grammar to detect bit-packet operations*

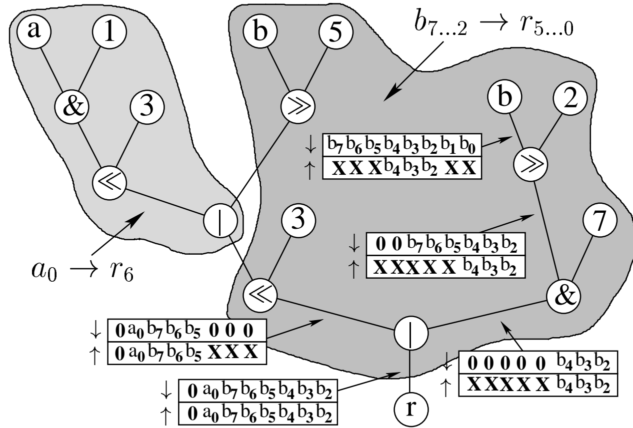

Figure 19. *DFT representation and two coverings*

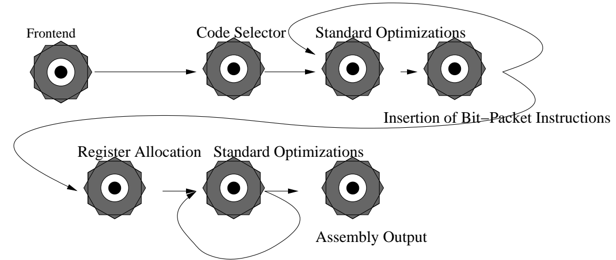

Figure 20. *Location of our optimization in the Infineon NP C compiler*

able to integrate a new optimization. Our optimization uses the intermediate representation and the processor description provided by the tool-set as interface. Therefore we believe that it is retargetable with low expenses to any architecture which supports bit-packet operations. We placed the optimization into the loop of pre-pass optimizations before the register allocation because it lowers the register-pressure (Figure 20). The advantage of this location is that the code for the address-generation is visible, because it is located after the code-generation phase. Further more there is no need for a real replace of obsolete code. The bit-packed operations are inserted into the assembly code and afterwards an already implemented dead-code elimination removes the obsolete code. Our optimization also profits from the previous running backend-optimizations which simplify the data-flow of a given assembly program.

The implementation of our optimization is only experimental. We noticed that especially the grammar for tree-pattern matching needs to be extended to detect a high percentage of bit-packet transfers in larger benchmarks.

4 Results

For benchmarking of our optimization we used an existing compiler where we plugged-in an additional optimization. Therefore we saved time for implementation. On the other hand we had the problem that the interface to the intermediate representation is not optimized to our needs. For example the insertion of bit-packets is the only tool in the compiler where a bit-oriented data-flow analysis is used. This may cause new problems in the compiler backend. For instance, the insertion of bit-oriented instructions may knock-out existing, integer based optimizations. In our test-cases the insertion of bit-packet operations enhances the code quality notedly, but we notice that the effort is lowered by the impeding of existing tools.

To create results we compared three cases: Compilation with the existing compiler, Compilation with additional use of Bit-Packet Operation Insertion and Hand-optimizing of compiler generated code. We decide to generate no code completely by hand. Even hand-optimizing small code fragments is extremely expensive. Therefore, at this early stage, we only present five small code examples. The benchmarks are created from fragments of the GSM-kernel source code [19]:

ins1 Encoding: insertion of bit-packets smaller than an integer into an existing stream.

ins2 Encoding: insertion of bit-packets overlapping an integer

extr1 Decoding: Extraction of bit-packets from a stream into an integer

extr2 Decoding: Extraction from overlapping integers

mem Bit-packet memory to memory data-transfers

The following table shows the benefit of Bit-Packet Operation Insertion and Hand-optimized code as different to the compiler with-out Bit-Packet Operation Insertion in percentage of cycles.

Name	Bit-Packet Insertion	Hand-optimized Code
ins1	-12%	-15%
ins2	-8%	-19%
extr1	-15%	-27%
extr2	-8%	-22%
mem	-14%	-45%

Figure 21. *Benefit in percentage of cycles*

5 Conclusion and further work

In previous work we showed that Bit-Packet operations can be supported by a compiler using *Compiler Known Functions* or *Compiler Intrinsics*. In this paper we showed that an automatic approach without user interaction. Adopting and extending known techniques we found a fast solution to detect Bit-Packet operations in a program data-flow.

First experimental results give an impression of the possibility to enhance the code quality of a compiler if the complete instruction set is supported. In opposite to [18] the algorithm outlined in this paper is usable for retargetable code with no machine specific compiler intrinsics. However our results also show that the code quality of the compiler can still be improved and that hand-optimized code saves 50 % more execution time than our optimization.

In further work we will enhance our tree-pattern matching grammar to detect a higher rate of bit-packed operations. We will also support the full instruction set of the processor which will enhance the code quality. Furthermore we will analyze the impact of the inserted bit-packed operations to other existing optimizations like pipe-hole optimization in order to close the gap to hand-optimized code.

Acknowledgments

This work was supported by the Informatik Centrum Dortmund (www.icd.de), by Infineon Technologies and Agilent Technologies. The authors would also like to thank Jörg Eckart, Robert Pyka and Luis Gomez, who significantly contributed to the tool implementation and test.

References

[1] The GNU CC Project, Free Software Foundation Inc. (2001). [Online]. Available: http://www.gnu.org/software/gcc/gcc.html

[2] R. M. Stallman: *Using and Porting GNU CC for Version 2.95*, Free Software Foundation Inc., Cambridge: 1999

[3] "Instruction Set Reference", in *Intel Architecture Developer's Manual*, Vol. 2, Order No. 243191, Intel Corporation, 1999, pp. 3-158, 3-355

[4] "Short Information MAS 3509 Compressed-Audio Decoder", Order No. 6251-505-2SI, Micronas 1999

[5] B. Wess: "Automatic Instruction Code Generation based on Trellis Diagrams", in *Proc. IEEE Int. Symp. on Circuits and Systems (ISCAS)*, 1992, pp. 645-648

[6] G. Araujo and S. Malik: "Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures", in *Proc. 8th Int. Symp. on System Synthesis (ISSS)*, 1995, pp. 36-41

[7] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang: "Code Optimization Techniques for Embedded DSP Microprocessors", in *Proc. 32nd Design Automation Conference (DAC)*, 1995, pp. 593-598

[8] A. Timmer, M. Strik, J. van Meerbergen, and J. Jess: "Conflict Modeling and Instruction Scheduling in Code Generation for In-House DSP Cores", in *Proc. 32nd Design Automation Conference (DAC)*, 1995, pp. 593-598

[9] S. Bashford and R. Leupers: "Constraint Driven Code Selection for Fixed-Point DSPs", in *Proc. 36th Design Automation Conference (DAC)*, 1999, pp. 817-822

[10] D.H. Bartley: "Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes", in *Software – Practice and Experience*, vol. 22(2), 1992, pp. 101-110

[11] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang: "Storage Assignment to Decrease Code Size", in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1995, pp. 186-195

[12] R. Leupers and F. David: "A Uniform Optimization Technique for Offset Assignment Problems", in *Proc. 11th Int. System Synthesis Symposium (ISSS)*, 1998, pp. 3-8

[13] E. Eckstein, and A. Krall: "Minimizing Cost of Local Variables Access for DSP Processors", in *Proc. ACM Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 1999, pp. 20-27

[14] R.J. Fisher, and H.G. Dietz: ”Compiling for SIMD Within a Register”, in *Proc. 11th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC98)*, 1998, pp. 290-304

[15] R. Leupers: ”Code Selection for Media Processors with SIMD Instructions”, in *Design Automation & Test in Europe (DATE)*, 2000, pp. 4-8

[16] S. Larsen and S. Amarasinghe: ”Exploiting Superword Level Parallelism with Multimedia Instruction Sets”, in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2000, pp. 145-156

[17] P. Paulin: ”Network Processors: A Perspective on Market Requirements, Processors Architectures, and Embedded S/W Tools, in *Design Automation & Test in Europe (DATE)*, 2001, pp. 420-427

[18] J. Wagner and R. Leupers: ”C Compiler Design for a Network Processor”, in *IEEE Trans. on CAD of Integrated Circuits and Systems (TCAD)*, Vol. 20, No. 11, 2001, pp. 1302-1308

[19] J. Degener and C. Bormann: ”GSM: Implementation of the 13 kbps GSM speech coding standard”, (2001). [Online]. Available: file://tub.cs.tu-berlin.de/pub/tubmik/gsm-1.0.tar.Z

[20] X. Nie, L. Gazsi, F. Engel, and G. Fettweis: ”A New Network Processor Architecture for High-Speed Communications”, in *IEEE Workshop on Signal Processing Systems (SiPS)*, 1999, pp. 548-557

[21] A. W. Appel: *Modern Compiler Implementation in C*, Cambridge, New York: Cambridge University Press, 1998

[22] H. Yasuura, H. Tomiyama, A. Inoue, and F. N. Eko: ”Embedded System Design Using Soft-Core Processor and Valen-C”, in *Proc. of Asia Pacific Conf. on Hardware Description Languages* ,1997 , pages 121–130

[23] R. Leupers: *Retargetable Code Generation for Digital Signal Processors*, Norwel: Kluwer 1997

[24] D. Brooks, M. Martonosi: ”Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance”, in *Proc. of HPCA-5*, 1999

[25] M. Budiu, S. C. Goldstein: ”Bit Value Inference: Detecting and Exploiting Narrow Bitwidth Computations”, in *Proc. of 6th International Euro-Par Conference*, 2000

[26] A. V. Aho, S. C. Johnson, J. D. Ullman: ”Code Generation for Expressions with Common Subexpressions”, in *Journal of the ACM*, vol. 24, No. 1, 1977

[27] C. W. Fraser, D. R. Hanson, T. A. Probesting: ”Engineering a Simple, Efficient Code Generator Generator”, in *ACM Letters on Programming Languages and Systems*, vol. 1, no. 3, 1992, pp. 213-226

[28] C. W. Fraser, D. R. Hanson, T. A. Probesting: ”Burg – Fast Optimal Instruction Selection and Tree Parsing”, in *ACM SIGPLAN Notices 27 (4)*, 1993, pp. 68-76

[29] M. A. Ertl: ”Optimization During Tree-Parsing Code Selection”, Talk given at *The Dagstuhl Seminar on Code Optimization*, 2000

[30] R. Leupers: *LANCE: A C Compiler Platform for Embedded Processors*, Embedded Systems/Embedded Intelligence, (2001). [Online]. Available: http://LS12-www.cs.uni-dortmund.de/∼leupers

[31] J. Wagner and R. Leupers: ”A Fast Simulator and Debugger for a Network Processor”, in *Proc. of Embedded Intelligence Conference*, 2002