Use of a Bit-true Data Flow Analysis for Processor-Specific Source Code Optimization

Heiko Falk
Computer Science 12
University of Dortmund
D-44221 Dortmund
Heiko.Falk@udo.edu

Jens Wagner
Embedded Systems Group
Informatik Centrum Dortmund e. V.
D-44227 Dortmund
wagner@icd.de

André Schaefer
Product Development
ACCESS Systems Europe GmbH
D-46047 Oberhausen
schaefer@access-sys-eu.com

Abstract

Nowadays, key characteristics of a processor's instruction set are only exploited in high-level languages by using inline assembly or compiler intrinsics. Inserting intrinsics into the source code is up to the programmer, since only few automatic approaches exist. Additionally, these approaches base on simple code pattern matching strategies.

This paper presents techniques for processor-specific code analysis and optimization at the source-level. It is shown how a bit-true data flow analysis is made applicable for source code analysis for the TI C6x DSPs for the very first time. Based on this bit-true analysis, fully automated optimizations superior to conventional pattern matching techniques are presented which optimize saturated arithmetic, reduce bitwidths of variables and exploit SIMD data processing within source codes. The application of our implemented algorithms to complex real-life codes leads to speed-ups between 33% – 48% for the optimization of saturated arithmetic, and up to 16% after SIMD optimization.

1. Introduction

Due to the lack of compilers generating efficient code, assembly-level software development for embedded processors was common in the past. However, both embedded applications and embedded processors are getting more and more complex, and the evolution of optimizing compilers allowed the replacement of assembly-level programming.

However, modern compilers often fail to generate highly efficient machine code even though they include many standard optimizations like e. g. dead code elimination or loop-invariant code motion. This is due to instruction set extensions of modern processors like e. g. digital signal processors *(DSPs)* that can not be exploited directly in programming languages like ANSI-C [4]. Examples for such extensions are saturated arithmetic or multimedia SIMD instructions (*Singe Instruction, Multiple Data*), where no analog constructs exist in programming languages.

To exploit such instruction sets within programming languages, inline-assembly was used in the past: small assembly snippets written and manually optimized by the programmer were embedded in the high-level source codes. The use of inline-assembly is disadvantageous because maintenance and portability of such source codes are poor.

```
sum = a + b;
if (!((a∧b) & 0x80000000))
  if ((sum∧a) & 0x80000000)
    sum = a<0 ? 0x80000000:
                0x7FFFFFFF;
```

```
sum = _sadd(a, b);
```

Figure 1. TI C6x _sadd Intrinsic

Nowadays, almost each compiler for DSPs offers *compiler known functions* or *intrinsics*. Using intrinsics, particular features of a processor can be exploited by the programmer. The compiler maps a call to an intrinsic not to a regular function call, but to a fixed sequence of machine instructions. For example, the complex code shown on the left hand side of Figure 1 implementing a saturating addition in ANSI-C can be replaced by an intrinsic using the TI C6x CCS compiler [11].

Using intrinsics, the resulting optimized source code is highly efficient since the compiler replaces the intrinsic by an extremely fast sequence of assembly instructions. But since intrinsics are non-standardized programming language extensions, source codes using intrinsics are no longer portable at all. Currently, only poor tool support exists to aid the programmer in replacing suitable source code fragments by efficient intrinsics.

This paper combines a bit-true data flow analysis (*DFA*) with source code optimizations exploiting intrinsics for the TI C6x DSP [10]. Using this DFA, we analyze ANSI-C programs and find complex code structures feasible for optimization. Our optimizations replace these complex code fragments by intrinsics for the TI C6x CCS compiler leading to highly optimized non-ANSI-C programs afterwards. In this paper, we present three different optimizations: First, saturated arithmetic common to many audio / video codec standards is detected and optimized. Second, SIMD instructions for parallel packed arithmetic on e. g. halfwords are exploited. Since SIMD instructions rely on data types with small bitwidths like e. g. `short` or `char`, we present a third transformation detecting unused bits of source-level variables and reducing them to the smallest required data type.

Results show that the proposed optimizations lead to significant speed-ups of up to 48%. In addition, the error-prone task of inserting intrinsics into the source code is fully automated now. The source codes the programmer operates on remain fully portable, because the optimizations introducing machine dependencies are applied as post-processing step.

Related work is given in Section 2, Section 3 presents our bit-true source-level DFA. Section 4 presents the intrinsic-based optimizations, followed by Section 5 containing results. Finally, Section 6 concludes this paper.

2. Related Work

[2, 3, 12] are typical examples for code optimizations using intrinsics. They all quantify the effect of various optimizations on the performance of selected benchmarks running on a given processor. In [3], a G.723.1 speech coder is optimized for a TI C6x processor, using intermixed ANSI-C and assembly code to implement the speech coder. Assembly codes are hand optimized in a not described manner, C fragments are replaced by intrinsics wherever possible. In [2], not only intrinsics for the same TI processor are studied, but also other optimizations like e. g. loop unrolling or software pipelining. Some small DSPstone kernels are used to benchmark the applied optimizations. Finally, [12] replaces floating point by fixed point numbers and applies intrinsics and software pipelining to an image reconstruction algorithm for the Philips TriMedia processor.

In all these papers, all optimizations were done manually, no tool support was used. The effect of the used intrinsics on the reported speed-ups is unclear, because results are often presented for all optimizations together, and not just for single substeps. Where results just for intrinsics are given, speed-ups between 30 % – 50 % were reported.

[8] uses loop distribution, unrolling and pattern matching to exploit SIMD instructions for the TriMedia processor. After that, pattern matching is used to find code fragments in the transformed loops that can be replaced by intrinsics. In contrast to [2, 3, 12], the entire transformation process is automated, but results are only presented for three very small loop kernels. In addition, just simple pattern matching techniques are used to optimize the code. No advanced analyses are presented.

[5] exploits SIMD instructions during code selection of a compiler. It extends traditional tree-pattern matching to cope with the fact that SIMD code patterns form data flow graphs and nut just data flow trees *(DFTs)*. This is done by introducing alternatives for code selection into the tree-pattern matcher. For all DFTs, the best alternatives maximizing the usage of SIMD instructions are selected using integer linear programming *(ILP)*. This approach is cumbersome in that sense that it applies tree-based techniques to a graph-based optimization problem. The use of graph-based analyses as proposed in this paper is better suited.

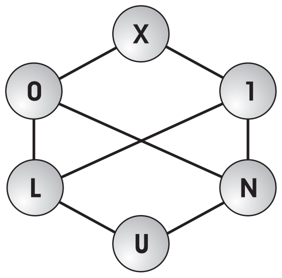

Figure 2. Lattice of Multi Valued Logic

[9] demonstrates that code selection for SIMD instructions can be performed without using tree-based techniques by completely modeling code selection as an ILP.

Traditional DFA [6] is usually tied to a processor's wordlength, i. e. information on definitions and uses *(DEF / USE)* of data only relates to blocks of e. g. 32 bits. Protocol processing applications and network processors operate on bit-packets which typically are not aligned to the wordlength. Bit-packets may start and end at arbitrary positions within a register; they may even cross register boundaries. To generate efficient code for such systems, bit-level DFAs are required computing data flow information for each individual data bit. The bit-level DFA of [1] determines for each bit, if it is unknown but relevant, the constant 0, the constant 1 or a don't care. [7, 13] extend this DFA by the information if the location of a bit in memory is known but its value is not. This extended bit-true DFA is adapted in this paper and is explained in more detail in Section 3.

3. Bit-true Source-Level Data Flow Analysis

This section presents the bit-true DFA used by the processor-specific source code optimizations described in Section 4. Section 3.1 briefly resumes the basic concepts of this DFA [7, 13]. Section 3.2 describes the connection between the DFA and SUIF [14] to analyze ANSI-C source codes.

3.1. Concepts for Bit-true Data Flow Analysis

The bit-level DFA used in this paper is based on a multi valued logic representing the information known about a single bit within the data flow graph *(DFG)*. The multi valued logic can be modeled by the lattice depicted in Figure 2. The lattice's elements encode the following information: **X:** the bit's value is irrelevant for the result of the program (don't care), **0** and **1:** the value is constant, **L:** the bit's value is copied from a given location, but its contents is unknown, **N:** the value is copied from a given location and negated, **U**: the bit is entirely unknown. This lattice forms a partial order according to the edges shown in Figure 2.

The DFA attaches information about the bits on the edges of the DFG which represent operands of an operation. The operations themselves are represented by the DFG's nodes. The DFA operates in two phases.

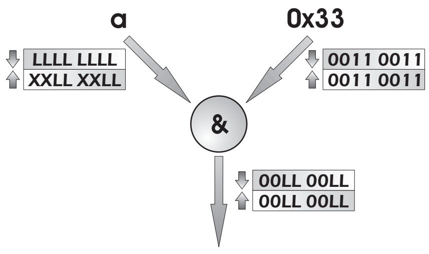

Figure 3. Bit-true DFA of an AND operation

In the beginning, all bits of all edges are set to **U**. First, the DFA traverses the DFG in direction of the edges (top-down analysis), starting from the source nodes. During top-down analysis, the bit-true effect of an operation (DFG node) on the operation's result (outgoing edge) is computed, based on the operation's inputs (incoming edges). The result of the top-down phase is attached to the outgoing edge as so-called *down-value*. This down-value is a string of lattice elements, where each item of the down-value represents the information gathered for the belonging operand's bit.

After that, a bottom-up phase is applied in opposite edge direction, starting from the DFG's sink nodes. Using the pre-computed down-values, bottom-up analysis calculates whether a bit's value dominates the current operation or not. If a bit's value is irrelevant (don't care), its belonging data flow information is set to **X**. The results of the bottom-up phase are attached to the DFG edges as so-called *up-values*.

The course of this DFA is shown using the ANSI-C assignment `d = a & 0x33;` `a` is a non-analyzable function argument, and thus its value is unknown (cf. Figure 3). However, `a` is a known location for the first operand of the AND operation so that all bits of the edge are set to **L** during top-down analysis (marked with $\Downarrow$ in the figure). Since the second operand is a constant, its down-value only consists of **0**s and **1**s. The down-value of the output of the AND operation contains four **0**s due to the zeroes of the down-value of `0x33`. All other bit positions are set to **L** since the bitwise AND of an arbitrary value with **1** results in that value.

During bottom-up analysis, it is detected that four bits of the down-value of operand `a` don't have any influence on the result `d`, because they are overridden by the zero-bits of the constant `0x33`. Accordingly, these bit positions are set to **X** for the up-value of `a`.

This example shows that the top-down phase "simulates" an operation's behavior at the bit level. In this stage, simulation functions are required for all possible operations. Using bitwise truth tables on the lattice shown in Figure 2, down-values of input parameters are connected to compute a resulting down-value. For bottom-up analysis, other simulation functions are needed inserting **X**s in the up-values of an operation's incoming edges. Section 3.2 describes how down and up simulation functions for the operations of the SUIF compiler system and the TI C6x DSP are set up.

3.2. Adaption for Source-Level Analysis

This paper uses SUIF [14] to analyze and optimize ANSI-C source codes. Our optimizations (cf. Section 4) insert intrinsics for the TI C6x DSP into the source codes using SUIF. For these reasons, the bit-true DFA described in Section 3.1 needs to be tailored to this environment. The DFA must be able to do top-down and bottom-up analysis of all operations of SUIF, and all simulation functions need to model the behavior of these operations on the TI C6x platform. For the sake of brevity, the operations of SUIF are not described here, even though they are fully implemented.

Source nodes are either function arguments or constants. Sink nodes are final store operations to either memory locations or variables. Needed data flow information are fully supported by SUIF and used by the DFA.

4. Intrinsic-Based Source Code Optimizations

This section presents the source-level optimizations and transformations using the bit-true DFA for the TI C6x DSP. Section 4.1 describes the insertion of intrinsics for saturated arithmetic. The use of the bit-true DFA to reduce bitwidths of variables is shown in Section 4.2. Section 4.3 presents the exploitation of SIMD intrinsics at the source code level.

4.1. Optimization of Saturated Arithmetic

Saturated arithmetic is a typical DSP feature ensuring that no wrap-around occurs after an arithmetic operation. As already shown in Figure 1, saturated arithmetic is expressed most intuitively in a two-phase approach in ANSI-C: First, the two operands are connected using the operator of choice (here: addition). Then, bit-masking operators are used to check and compare the sign-bits of result and operands. If an over- / underflow is found, the result is assigned the largest / smallest value of its data type.

In order to find ANSI-C code fragments which are candidates for being replaced by an intrinsic, the DFG generated from the C source code is analyzed using the bit-true DFA presented in the previous section. After that, particular subgraphs of the analyzed and annotated DFG are searched representing feasible C code fragments.

In the case of a saturated addition, all occurrences of an `add` operation in the DFG are checked first. For each occurrence, it is verified if the two operands of the addition are XORed and are ANDed with the minimum constant value of the actual data type, and if a comparison with 0 is done. If this is the case, it is checked if the addition's result is XORed with the first operand of the addition, and the outcome needs to be ANDed with the minimum constant again. Finally, an assignment of either the minimum or the maximum constant to the addition's result needs to be present in the DFG. If a subgraph of the DFG is found fulfilling all these criteria, it can be completely removed and can be replaced by a call to the `_sadd` intrinsic.

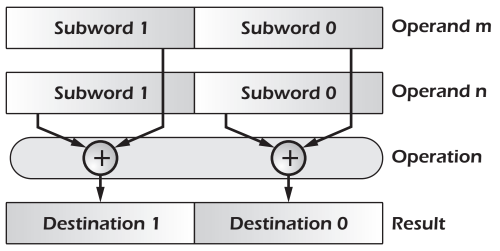

Figure 4. SIMD Halfword Addition

This approach of data flow graph search is much more flexible compared to pure pattern matching techniques [8]. In contrast to a pure source code pattern matching, our DFG search allows that the individual parts of such a feasible source code fragment are scattered – they need not occur line by line in the source code in the exact shape as shown in Figure 1. This is due to the fact that we only consider DFG nodes, edges and their relationships plus the data flow information gathered by the bit-true DFA in order to check the occurrence of the constants required for saturation. The DFG does not make any assumption of where and how the belonging operations reside in the source code.

In addition to the exploitation of saturated additions sketched above, we are able to identify source code fragments representing saturated subtractions, multiplications and left-shifts. For such code fragments, the TI C6x intrinsics `_ssub`, `_smpy` and `_sshl` are inserted, respectively.

4.2. Bitwidth Reduction

The bitwidth reduction transformation assigns data types of smaller bitwidths to variables wherever possible in the source code. This bitwidth reduction by itself is not an optimization that should have a significant impact on the runtime of an application. Instead, it is meant to be a preparatory step potentially enabling the exploitation of SIMD instructions presented in Section 4.3, that relies on the presence of variables of small data types.

In order to reduce bitwidths of variables, the up-values of the DFG edges representing the variable need to be checked. A reduction of a larger data type to a smaller one can only be performed if enough of the most significant bits (*MSBs*) of the variables are redundant (**X**'s in the up-values). Whenever all of the MSBs of the up-value of a variable are set to **X**, this variable is a potential candidate for bitwidth reduction. For the TI C6x DSP, the data type `int` is 32 bits wide, `short` 16 bits, and `char` 8 bits. For a variable of type `int` or `unsigned int`, a reduction to e. g. `char` or `unsigned char` can be done if at least 24 leading **X**'s are detected.

A variable can also be reduced to a smaller data type if it contains enough leading **0**'s. However, care must be taken if the original variable is signed. Then, a reduction is only legal if the MSB of the reduced variable is also **0**.

```
short o1, o2,  | int SIMDr;
      r3, r4;  | short r1, r2;
short r1, r2;  | SIMDr = _add2(_pack2(o1, o3),
               |               _pack2(o2, o4));
r1 = o1 + o2;  | r1 = (short) (SIMDr >> 16);
r2 = o3 + o4;  | r2 = (short) SIMDr;
```

Figure 5. TI C6x _add2 and _pack2 Intrinsics

4.3. Exploitation of SIMD Instructions

SIMD instructions partition a 32-bit word into several identical objects which can then be fetched, stored and operated on in parallel (cf. Figure 4). These instructions allow the full exploitation of the processor's wordlength in signal and data processing applications.

To exploit SIMD instructions, two sequential arithmetic operations (in the example shown in Figure 5: `short` additions) on variables of smaller data type need to be found in the DFG. Under certain conditions, these operations can be replaced by a SIMD intrinsic. In general, SIMD processing consists of three steps (cf. Figure 4): packing, the SIMD operation itself and unpacking of the result. These steps are supported by intrinsics. Corresponding C statements will be automatically replaced by intrinsics.

The above insertion of SIMD intrinsics is only applied if some rules are satisfied: First, the operations to be parallelized have to be data flow independent, i. e. there must not be any flow dependence or anti dependence. Second, the execution of the first sequential arithmetic operation must always be followed by exactly one execution of the second operation. This prevents two operations from being optimized that lie in different levels of the control flow hierarchy.

Finally, care must be taken if the operands or the variables holding the results of the original sequential arithmetic operations are redefined between the two operations to be optimized. There exist several constellations of redefinitions of these variables preventing the SIMD optimization. Using the DFG of the source code to be optimized and the DEF / USE relationships between the sequential operations, a SIMD optimization only takes place if all DEFs of the operands or result variables can be moved before the very first sequential operation legally. Since this is exactly the data flow obtained when using SIMD intrinsics, this restriction guarantees the correctness of the optimization.

Obviously, our DFA-based analysis and optimization approach is superior to pattern matching [8]. Using pattern matching, it is hard or even impossible to check complex data and control flow dependencies between operations to be parallelized. Additionally, our bit-level DFA approach potentially outperforms other techniques since using the bit-level DFA, we are able to generate variables of data types suitable for SIMD processing. Currently, only the SIMD intrinsics `_add2` and `_sub2` for halfword arithmetic are gen-

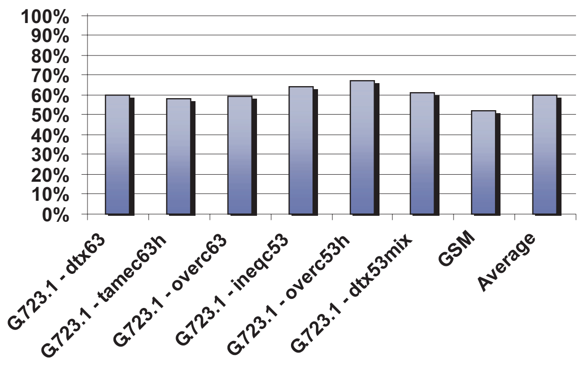

Figure 6. Rel. Runtimes for Saturation Intrinsics

erated by our tool. Even though other TI processors (C64x) support bytewise SIMD instructions, we consider only intrinsics common to all TI C6x processors in this work.

5. Results

This section presents the impact of the optimizations presented in Section 4 on the runtimes of benchmarks for the TI C6x DSP running at 133 MHz. To compute runtimes, the source codes of an application were compiled with the native TI CCS compiler at the highest optimization level. The resulting binary was simulated with the cycle true instruction set simulator of TI, resulting in the runtime of the original unoptimized application. After that, the source codes are fed into our optimization tools generating optimized source codes. These codes were compiled and simulated the same way as was done for the unoptimized source codes, yielding the runtimes of the optimized applications.

Two complex applications served as benchmarks. First, a G.723.1 codec of more than 13,000 lines of C code was used. For this codec, six input files for coding and decoding were available. We benchmarked G.723.1 for all these input frames and present the gathered results. Second, a full implementation of the ANT GSM Codec (4,500 lines of code) and one input sample was considered. Section 5.1 presents the results after optimization of saturated arithmetic. Section 5.2 gives an overview about the amount of redundant bits detected and reduced by our bit-level DFA. Finally, Section 5.3 shows the results for SIMD optimization.

5.1. Optimization of Saturated Arithmetic

Figure 6 shows the effect of the inserted intrinsics for saturated arithmetic on the runtime of the benchmarks for the indicated input frames. The figure shows the corresponding values for the optimized benchmarks as a percentage of the runtimes of the unoptimized versions denoted as 100%.

As can be seen from this figure, the automated insertion of the `_sadd`, `_ssub`, `_smpy` and `_sshl` intrinsics reduces the runtimes significantly. For the G.723.1 application, the runtimes of the optimized code vary between 67% and down to 58% of the runtime of the unoptimized application. Hence, our optimization is able to achieve savings between 33% up to 42%. For the GSM benchmark, the improvements are even higher. Here, the runtimes drop to a level of 52% of the original GSM runtime, implying an improvement of 48%. On average over all simulations depicted in Figure 6, total improvements of 40% were achieved by our optimization.

It is interesting to note that even though the relative improvements for the G.723.1 codec are close to each other for all input frames (approx. 40%), the absolute runtimes are very heterogeneous. For example, input frame `overc53h` requires an absolute amount 1.8 billions of cycles, whereas `dtx63` consumes 82 billions. Since the input frames are designed to cover different coding/decoding parameters and styles of the G.723.1 application, the measured improvements are essentially independent of the input patterns. Hence, the presented results are stable and clearly demonstrate the overall performance of our optimization on real-life benchmarks and input data.

Category	G.723.1	GSM
Totally Unused Bits	24.2%	79.5%
Unused Bits in leading bytes	4.4%	12.6%
Reduced Bits	2.2%	9.2%

Table 1. Detection and Classification of Redundant Bits

5.2. Bitwidth Reduction

Table 1 gives an overview of the percentages of bits classified as redundant and optimized by our bitwidth reduction transformation. It can be observed that the benchmarks contain a significant total amount of redundant bits not having any influence on the benchmark's behavior. For G.723.1, 24.2% of the bits of all variables are marked as don't care by our bit-level DFA. For the GSM application, an even higher rate of 79.5% of bits was detected. However, a bitwidth reduction can only take place if enough redundant bits are found at the most significant positions of a variable. For the G.723.1 application, a total of 4.4% of all bits was found to be redundant and occupying the contiguous most significant complete bytes of their belonging variables. For GSM, the amount of such bits being candidates for bitwidth reduction is around 12.6%. However, a bitwidth reduction finally led to total savings of 2.2% of all bits for G.723.1 and of 9.2% for GSM. As can be seen, the amount of effectively saved bits is much smaller than the number of unused bits in most significant bytes of variables. In both benchmarks, there exist several variables with redundant uppermost bytes that can not be reduced in bitwidth, because these variables serve as function arguments or return values. Since we currently do not apply inter-procedural analysis and optimization, such variables can not be optimized.

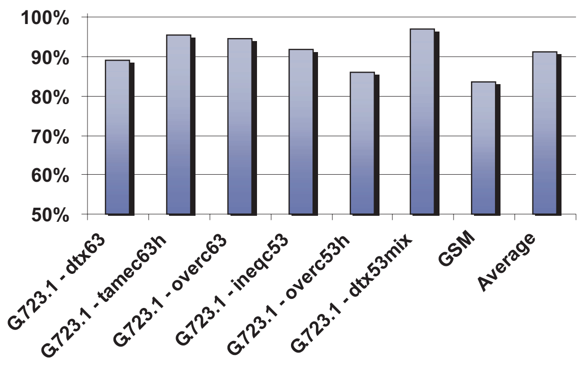

Figure 7. Rel. Runtimes for SIMD Intrinsics

5.3. Exploitation of SIMD Instructions

Figure 7 shows the achieved speed-ups of the benchmarks after insertion of SIMD intrinsics. Again, the figure shows all simulated runtimes after SIMD optimization as percentage of the runtimes before the optimization. For the G.723.1 application, the relative runtimes after SIMD optimization vary between 97% (`dtx53mix`) and 86% (`overc53h`) of that before, leading to savings from 3% up to 14% achieved by our optimization. For GSM, a relative runtime of 84% was measured when using SIMD intrinsics, so that we achieved 16% improvement in this case.

On average over all experiments depicted in Figure 7, savings of 8.9% were achieved. These savings are not as high as one would possibly expect which is due to two reasons. First, our bit-true DFA and our optimizations exclusively consider scalar variables in a source code. In contrast, SIMD instructions are intended to serve for number-crunching when processing large arrays of data. As a consequence, our techniques do not insert SIMD intrinsics for arithmetic on array elements since we currently are unable to verify that the array accesses are data flow independent of each other. Second, we do not apply loop transformations like e. g. loop unrolling or loop distribution prior to our optimizations to make parallelism present in loops explicit. Concluding, it can be stated that the current use of SIMD instructions by our optimization does not yet fully exploit the entire optimization potential present in the benchmarks. However, the circumstance that we still achieve significant runtime savings of up to 16% justifies our concepts.

6. Summary and Future Work

In this paper, novel approaches for processor-specific source code optimization were presented. The key contributions of this work are firstly that we adapted a bit-true data flow analysis for source-level optimization for a TI C6x signal processor. Secondly, three optimizations making use of this bit-true DFA are presented. They base on the automatic insertion of compiler intrinsics into the source code in order to exploit specific features of the TI instruction set. In detail, we successfully exploit instructions for saturated arithmetic and for SIMD processing at the source-level. In addition, techniques for detecting and removing unused bits present in source codes are presented.

The results presented in this paper underline the effectiveness of our approaches. The insertion of intrinsics for saturated arithmetic leads to reductions of runtimes between 33% and up to 48%. Furthermore, our techniques are able to detect that between 24.2% and 79.5% of all bits present in the benchmarks were redundant. Bitwidth reduction was able to eliminate up to 9.2% of all bits. Finally, the exploitation of SIMD instructions at the source code level yields speed-ups by up to 16%.

In the future, we intend to exploit more of the optimization potential present in real-life benchmarks than we currently do. In detail, we plan to make our analyses and optimizations inter-procedural in order to perform global optimizations. Additionally, the bit-true DFA will have to be capable of analyzing array accesses in order to use SIMD instructions for array processing. Finally, pre-processing transformations like e. g. loop unrolling or function inlining can be added to our optimization framework in order to make SIMD parallelism more explicit.

References

[1] M. Budiu and S. C. Goldstein. Bit Value Interference: Detecting and Exploiting Narrow Bitwidth Computations. In *Proceedings of 6th International Euro-Par Conference*, Aug. 2000.
[2] M. Coors, O. Wahlen, H. Keding, O. Lüthje and H. Meyr. TI C62x Performance Code Optimization. In *DSP Deutschland '99 - Grundlagen, Architekturen, Tools, Applikationen*, Sept. 1999.
[3] T. J. Dillon Jr. Texas Instruments Application Report: G.723.1 Dual-Rate Speech Coder: Multichannel TMS320C62x Implementation. *Electronic Engineering Times*, Feb. 2000.
[4] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.
[5] R. Leupers. Code Selection for Media Processors with SIMD Instructions. In *Proceedings of DATE*, Paris, Mar. 2000.
[6] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.
[7] X. Nie and J. Wagner. High Performance Network Protocol Processor - Architecture and Tools. In *Proceedings of "Euro DesignCon"*, Munich, 2004.
[8] G. Pokam, S. Bihan, J. Simonnet and F. Bodin. SWARP: A Retargetable Preprocessor for Multimedia Instructions. In *Proceedings of CPC*, Edinburgh, 2001.
[9] H. Tanaka, S. Kobayashi, Y. Takeuchi, S. Keishi and M. Imai. A Code Selection Method for SIMD Processors with PACK Instructions. In *Proceedings of SCOPES*, Vienna, Sept. 2003.
[10] *TMS320C6000 CPU and Instruction Set Reference Guide*. Texas Instruments Inc., Literature Number SPRU189D, Mar. 1999.
[11] *TMS320C6000 Optimizing Compiler User's Guide*. Texas Instruments Inc., Literature Number SPRU187G, Mar. 2000.
[12] T. Vander Aa, R. Lauwereins and G. Deconinck. Optimizing a 3D image reconstruction algorithm: Analyzing the capabilities of a modern compiler. In *Proceedings of SIPS*, San Diego, Oct. 2002.
[13] J. Wagner. Retargetable Optimizations for Program Codes in Intermediate Representations based on a Bit-True Bidirectional Data Flow Analysis (in german). Patent No. 10361972, German Patent Office, Nov. Munich, 2004.
[14] R. Wilson, R. French, C. Wilson et al. An Overview of the SUIF Compiler System. *http://suif.stanford.edu/suif/suif1*, 1995.