# Retargetable Code Optimization with SIMD Instructions

Manuel Hohenauer, Christoph Schumacher, Rainer Leupers,
Gerd Ascheid, and Heinrich Meyr
Institute for Integrated Signal Processing Systems
RWTH Aachen University, Germany

Hans van Someren
Associated Compiler Experts bv
Amsterdam, The Netherlands

## ABSTRACT

Retargetable C compilers are nowadays widely used to quickly obtain compiler support for new embedded processors and to perform early processor architecture exploration. One frequent concern about retargetable compilers, though, is their lack of machine-specific code optimization techniques in order to achieve highest code quality. While this problem is partially inherent to the retargetable compilation approach, it can be circumvented by designing flexible, configurable code optimization techniques that apply to a certain range of target architectures. This paper focuses on target machines with SIMD instruction support which is widespread in embedded processors for multimedia applications. We present an efficient and quickly retargetable SIMD code optimization technique that is integrated into an industrial retargetable C compiler. Experimental results for the Philips Trimedia processor demonstrate that the proposed technique applies to real-life target machines and that it produces code quality improvements close to the theoretical limit.

## Categories and Subject Descriptors

C.1.1 [**Single Data Stream Architectures**]: RISC/CISC, VLIW architectures; C.1.2 [**Multiple Data Stream Architectures**]: Single-instruction-stream, multiple-data-stream processors (SIMD); D.3.4 [**Processors**]: Retargetable compilers, optimization

## General Terms

Performance, Algorithms

## Keywords

SIMD, vectorization, subword parallelism, retargetable compilers

## 1. INTRODUCTION

With the increasing acceptance of application specific instruction set processors (ASIPs) [1, 2, 3] as efficient and flexible implementation vehicles in embedded system-on-chip SoC design, more and more commercial platforms (e.g. LISATek or Tensilica) are available for ASIP architecture exploration and design. These platforms comprise *retargetable software development tools*, including C compiler, instruction set simulator, debugger, and (dis)assembler, enabling the designer to quickly explore ASIP architectural alternatives for a given range of embedded applications. A key component of many of these platforms is the *retargetable C compiler*, which can, automatically or semi-automatically, be adapted to generate code for different target architectures. While retargetable compilers have found significant use in ASIP design in the past years, they are still hampered by their limited code quality as compared to hand-written compilers or assembly code. This is no surprise, since higher compiler flexibility comes at the expense of a lower amount of target-specific code optimizations. Therefore, it is common practice to manually enhance a generated compiler with target-specific optimizations, once the ASIP architecture exploration phase has converged and an initial working compiler is available.

A promising approach to further reduce ASIP compiler design effort is to identify *target processor classes* which, due to their architectural features, demand for specific code optimization techniques, and to implement these specific techniques such that retargetability within the given processor class is achieved. An example is the *retargetable software pipelining* support recently introduced for the CoSy compiler platform [4]. While being less useful for scalar architectures, software pipelining is a necessity for the class of VLIW processors, and for this class it can be designed in a retargetable (or configurable) fashion.

This paper focuses on another class of target processors, namely those equipped with *SIMD instructions*. Wikipedia provides the following definition:

> *In computing, SIMD (Single Instruction, Multiple Data) is a set of operations for efficiently handling large quantities of data in parallel, as in a vector processor or array processor. First popularized in large-scale supercomputers (. . . ), smaller-scale SIMD operations have now become widespread in personal computer hardware. Today the term is associated almost entirely with these smaller units.*

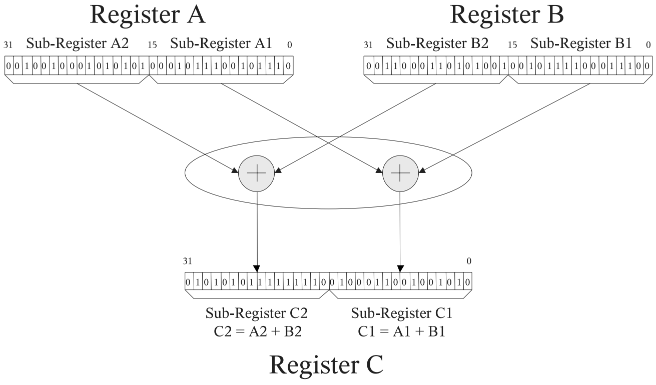

**Figure 1:** ***Sample arithmetic SIMD instruction: two parallel ADDs on 16-bit sub-registers of 32-bit data registers A, B, C***

In fact, many embedded processors today feature SIMD instructions in their instruction sets, in order to speed up execution of multimedia computational kernels. As exemplified in fig. 1, a SIMD

*CODES+ISSS'06,* October 22–25, 2006, Seoul, Korea.

instruction performs several primitive operations in parallel, using operands from several *sub-registers* of the processor’s data registers at a time. Other typical SIMD instructions perform more complex operations (e.g. partial dot products) or serve for sub-register packing and permutation. By using SIMD instructions, computations on ”short” data types (e.g. in audio or image processing) can be significantly accelerated. If instructions operate on data registers divided into $s$ sub-registers, then a linear peak speedup by a factor of $s$ can theoretically be expected (in section 6 it will be shown that deviations from this theoretical speedup occur due to different reasons). In turn, a poor utilization of SIMD implies a large loss in code quality.

While the SIMD concept has been introduced first for standard architectures (e.g. Intel MMX, Motorola AltiVec, AMD 3DNow!), it was quickly adopted in ASIPs for DSP applications (e.g. TI C6x, Philips Trimedia), and is being used in today´s custom ASIP designs, too (e.g. Tensilica Xtensa). Therefore, support for SIMD instructions in retargetable compilers is of high interest.

Several target specific C compilers already do include SIMD utilization, but it is still very weakly supported in ASIP compilers. For the use in this domain, *retargetable SIMD optimizations* are required. Therefore, in this paper, we propose a novel concept for retargetable code optimization for ASIPs with SIMD instructions, and we prove this concept by an implementation within an existing retargetable compiler framework and an experimental evaluation for a real-life, complex embedded processor.

The rest of this paper is organized as follows. In section 2 some related work is discussed. Based on the system overview in section 3, sections 4 and 5 describe the core of our approach, i.e. an algorithm for exploiting SIMD instructions in the compiler backend as well as the retargeting procedure for this algorithm. Section 6 summarizes our experiments for a sample embedded processor with SIMD support. Finally, sections 7 and 8 mention current limitations and conclusions.

# 2. RELATED WORK

A key problem in compiler utilization of SIMD instructions is that traditional code generation techniques, such as tree covering with dynamic programming [11], fail in case of SIMD. Hence, compilers without dedicated techniques are unlikely to exploit SIMD instructions at all.

Many C compilers, though, provide semi-automatic SIMD support via *compiler known functions* (CKFs) or *intrinsics*. CKFs make assembly instructions accessible at the C programming level, where the compiler expands a CKF call like a macro. However, due to the low-level programming style and poor portability of code with CKFs, this cannot be considered a satisfactory solution. Some advanced compilers (e.g. for Intel MMX and SSE) provide automatic generation of SIMD instructions, yet restricted to certain C language constructs. Moreover, these compilers are inherently non-retargetable.

ASIP design platforms comprising retargetable C compilers include LISATek [5], Expression [6], Mescal [1], and ASIPMeister [7]. However, no SIMD support has been reported for those tools yet. Tensilica’s [8] compiler for the configurable Xtensa processor supports SIMD, but it is restricted to a narrow range of architectures.

In the domain of ”general purpose” retargetable compilers, recent versions (4.x) of the gcc support SIMD [17] for certain loop constructs, but gcc is generally known as being difficult to adapt efficiently to embedded processor architectures.

In research on embedded processor code optimization, a number of techniques for SIMD utilization have been proposed recently. In [9] a combination of traditional code selection [10] and Integer Linear Programming based optimization is presented. This approach achieves high code quality but suffers from high complexity for large programs. The work in [13] presents an efficient approach for packing operations step by step into SIMD instructions, and it presents results for the AltiVec ISA. Further works in this domain deal with memory alignment optimization, length conversion [15] and interleaved data for SIMD [16], pointer alignment analysis [14], and flow graph permutations for more effective SIMD instruction packing [18].

In summary, a number of techniques for SIMD utilization in compilers with different levels of complexity are available, most of which are adapted for a certain target machine. Porting of a SIMD optimization technique to a new target machine is still a tedious manual process. Therefore our approach emphasizes *efficient utilization of SIMD instructions* and *compiler retargetability* at the same time. As outlined in the next section, this is implemented by integrating a relatively simple, yet effective, SIMD optimization pass into a well-tried retargetable C compiler framework. The amount of required target specific information is limited, so that most of it can be extracted automatically from high-level processor models.

# 3. SYSTEM OVERVIEW

We use the C compiler generation technique described in [12] that bridges the gap between the LISATek tools for ASIP design [5] and the CoSy compiler generation platform [4]. From a single target processor model, given in the LISA 2.0 architecture description language (ADL), a complete software development tool chain (C compiler, ISS, assembler, etc.) can be generated. The Compiler Designer tool [12] extracts a compiler-oriented view from a given LISA 2.0 ADL model and transforms it into a custom specification format. On this specification, CoSy is invoked as a ”backend” to generate the compiler executable.

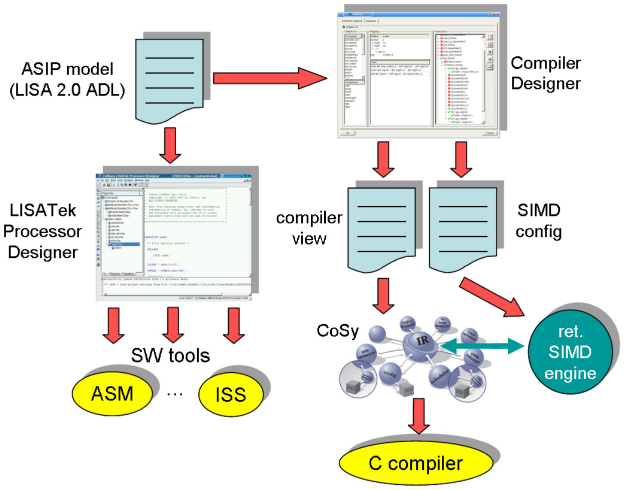

**Figure 2:** ***Tool flow for retargetable compilation with SIMD support***

CoSy compilers are designed in a highly modular fashion, which enables to add new optimization ”engines” mostly in a plug-and-play fashion. Hence, we added a retargetable *SIMD engine* (plus several auxiliary engines) into the framework that implements the techniques described in detail in section 4. Due to the coupling to LISATek, the SIMD properties of the target processor can be described within the same ”golden” ADL ISA model that drives the entire ASIP design process, and they can be largely automatically translated into the CoSy compiler description format (see section 5). This tool flow, illustrated in fig. 2, enables a complete and retargetable path from the target machine model to a SIMD-enabled C compiler.

# 4. SIMD ENGINE

## 4.1 Basic design decisions

Many existing approaches generate SIMD instructions only at a late stage of the compilation process, i.e. in the backend. The advantage is that more information is available, both about the input program’s precise data flow and about the target machine instruction set, potentially leading to higher code quality. However, such late formation of SIMD instructions is not well suitable for *retargetable* compilers.

We therefore chose a *high level approach to SIMD optimization*, where combination of SIMD instructions takes place as one of the first compilation phases, almost directly after the program intermediate representation (IR) has been generated from the C source code. This approach is easily retargetable, since it requires only very basic knowledge of the target machine (as described in section 5). Furthermore, it simplifies code generation, since it abstracts from low-level problems like register allocation for SIMD sub-words in the backend. In addition, our approach allows for reuse of all existing ”downstream” standard code generation and optimization engines of the underlying compiler framework.

A second design decision concerns the representation of generated SIMD instructions in the compiler’s IR. All IR formats comprise elements for representing primitive operations like ”+”, ”-”, ”*”, etc. However, there is usually no notion of SIMD operations such as ”two parallel ADDs”. Hence, an extension of the underlying IR format would be required. From a practical viewpoint, such extensions have a dramatic impact on most ”downstream” compiler engines. They demand either for expensive manual adaptations in the engines or lead to poor performance of optimization engines, which in turn implies poor code quality. Therefore, we decided to represent generated SIMD instructions internally in the form of *compiler-known functions* (CKFs). CKFs are transparent for other compiler engines and therefore cause no problems[1]. A further advantage of using CKFs is that during the SIMD generation process, which in turn involves multiple sequential stages, the current IR can be dumped into a human-readable valid C code file anytime for debugging purposes.

## 4.2 Preprocessing

Two preprocessing engines have been developed to exhibit sufficient sub-word level parallelism for the subsequent SIMD instruction combination. The first one performs *loop unrolling*, a well known standard code transformation [11] that duplicates loop bodies by a given *unroll factor*. While our underlying compiler platform CoSy already contains an unrolling engine, we added a custom engine that gives precise control about which loops are unrolled by what factor.

The second preprocessing engine performs *scalar expansion*. This transformation serves to split accumulators within unrolled loop bodies into multiple variables that are added only after loop execution. Scalar expansion (exemplified in fig. 3) removes artificial inter-statement dependencies, e.g. by splitting accumulator variables thereby exposing more parallelism. It is used as a subroutine in the main algorithm described below.

```
s0 = s1 = s2 = s3 = 0;
for(i=0; i<64; i+=4)
{
    s0 = s0 + a[i+0] * b[i+0];
    s1 = s1 + a[i+1] * b[i+1];
    s2 = s2 + a[i+2] * b[i+2];
    s3 = s3 + a[i+3] * b[i+3];
}
sum = sum + s0 + s1 + s2 + s3;
```

**Figure 3:** ***Sample FOR-loop unrolled by factor 4 and after scalar expansion***

## 4.3 SIMD instruction combination

For a given IR of an input C program, we use an iterative algorithm (fig. 4) that combines IR operations into SIMD instructions and replaces such instructions by CKFs in the IR. It focuses on the innermost loops with a single basic block, since these tend to be the hot spots of the input program. Certain multiple basic block constructs may have been merged into a single basic block by an if-conversion pass prior to SIMD. The algorithm forms SIMD instructions step by step, starting at IR nodes that represent operand LOADs from memory (”collect memory accesses”), and updating memory access information after each step, in order to finalize combination by generating SIMD STORE operations.

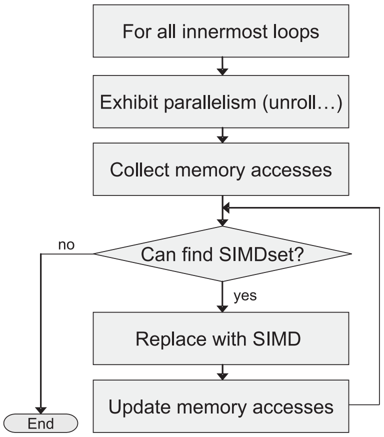

**Figure 4:** ***Iterative SIMD instruction combination flowchart***

We denote a set of IR operations that can be combined into a SIMD instruction as a *SIMD-set*. The algorithm for SIMD-set formation first constructs a *data flow graph* (DFG) for the loop body to exhibit interdependencies. Next, it checks a number of constraints for tuples $N = (n_1, \ldots, n_k)$ of DFG nodes, where $k$ denotes the number of sub-registers as specified in the machine description (frequently $k = 2$ or $k = 4$). Amongst others[2], nodes $n_i$ of a potential SIMD-set must

1. represent *isomorphic operations* that can be combined to a SIMD instruction according to the target machine description
2. show no *interdependencies* that would prevent parallelism
3. satisfy *memory alignment* constraints if demanded by the target machine

For the latter, an inter-procedural pointer alignment analysis similar to [14] has been implemented to improve the static (i.e. compile-time) alignment checking and to reduce the overhead of dynamic (i.e. run-time) alignment checks, which may still be required depending on the given input program characteristics.

After successful construction of each SIMD-set, a corresponding CKF is put into the IR, and the iteration from fig. 4 goes on. The basic idea of the iteration is illustrated in fig. 5. Part (1) shows an initial IR for a sample loop body (unrolled twice) that computes the dot product of two vectors $a$ and $b$ and stores the result in vector $y$. The left and right fragments of the computations are isomorphic and meet the memory alignment constraints. Therefore, after three iterations, the left and the right arguments (16-bit LOAD operations) of the two ”+” operations have been combined to 32-bit SIMD LOAD operations, and finally the ”+” operations themselves are combined to a SIMD instruction. Now the IR has the intermediate state shown in fig. 5 (2). Note that a valid IR is retained in each step. For this purpose, explicit ”extract” operations have been inserted that select 16-bit sub-words out of the 32-bit result of the dual add operation. The next iteration finds that the two 16-bit STORE operations form a SIMD-set, too, which in turn enables to later eliminate the extract operations in a cleanup phase. Finally, the IR state in fig. 5 (3) is reached, and the algorithm terminates.

[1] It is important to note here that this does not imply the disadvantages of CKFs mentioned in section 2. In our approach, CKFs are only used as a special IR element. They are later automatically replaced with assembly instructions in the backend. The compiler *user* is not bothered with CKFs at all, while for the processor *designer* it is a one-time effort to specify the CKF semantics for the SIMD instructions of a given target machine.

[2] The detailed description is omitted here for sake of brevity, since the constraints resemble those already described in detail in previous work, see section 2.

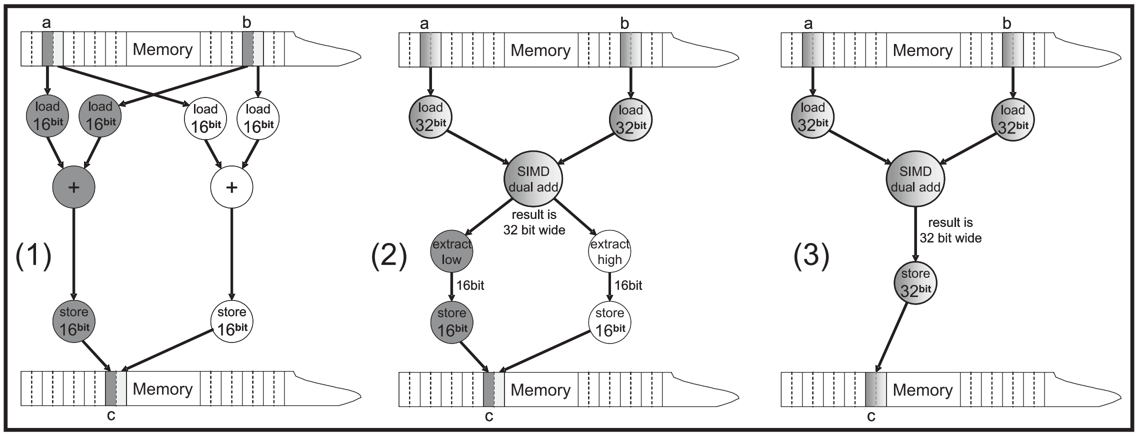

**Figure 5:** ***IR states in different iterations***

Since the algorithm avoids an exhaustive search within the given loop body in favor of an iterative, step-by-step approach to SIMD instruction formation, it requires only low-degree polynomial complexity ($O(n^3)$ worst case for $n$ variable accesses in the IR). In practice we found that this relatively simple heuristic consumes only a few CPU seconds of compilation time and utilizes SIMD instructions very well for speeding up common DSP code benchmarks. Insertion of SIMD instructions may lead to an increase in code size, though, due to the possible necessity of inserting extra code for dynamic pointer alignment checks before loop entry points and keeping both code versions (SIMD and non-SIMD loop).

## 4.4 Code example

We provide a more detailed example to illustrate the representation of SIMD instructions in the IR. Fig. 6 shows the initial sample C source code after preprocessing. We assume availability of SIMD instructions for two sub-words and a target machine that requires memory aligned SIMD operands.

```
void dotproduct(short int *pa; short int *pb; short int *pc)
{
  short int sum;
  short int s0, s1, s2, s3;
  int i;

  sum = 0;
  s0 = s1 = s2 = s3 = 0;
  for(i=0; i<64; i+=4)
  {
    s0 = s0 + (*pa * *pb) * *pc;
    pa++; pb++; pc++;
    s1 = s1 + (*pa * *pb) * *pc;
    pa++; pb++; pc++;
    s2 = s2 + (*pa * *pb) * *pc;
    pa++; pb++; pc++;
    s3 = s3 + (*pa * *pb) * *pc;
    pa++; pb++; pc++;
  }
  sum = sum + (s0 + s1) + (s2 + s3);
}
```

**Figure 6:** ***Initial code after loop unrolling and scalar expansion***

In the first iterations of the algorithm from fig. 4, the SIMD engine identifies that two pairs of multiplications can be combined and be replaced by CKFs (`SIMD_mul_2x16`). Furthermore, necessary subword extract functions (`EXTRACT_short_x_of_2`) are inserted, and temporary variables for intermediate results are allocated.

Fig. 7 shows the resulting code after several further steps (as generated by the IR-to-C code dump facility of our compiler platform). The SIMD-set computation has been finalized by detecting that the multiply results can be processed further by SIMD additions. Extract operations are only required after the "SIMDfied" loop body in order to recover the correct partial accumulator values (`s0, ..., s3`) resulting from scalar expansion. Here it is assumed that the alignment analysis cannot resolve the alignment of the pointers, thus a dynamic alignment check has been inserted (`if(((pa|pb|pc) & 3) == 0)`) to rule out misaligned pointers. If the check fails, a non-SIMD version of the loop is executed in the else-branch. Finally, standard optimizations, such as dead code elimination, have been invoked to remove superfluous operations (e.g. extracts) from previous phases. The resulting code is passed to the compiler backend for assembly code generation.

```
void dotproduct(short int *pa, *pb, *pc)
{
  short int sum;
  short int s0, s1, s2, s3;
  int i;
  int SEaccucommon01, SEaccucommon23;

  sum = 0;
  s0 = s1 = s2 = s3 = 0;
  SEaccucommon01 = SEaccucommon23 = 0;

  if( ((pa|pb|pc) & 3) == 0 )
  {
    for(i=0; i<64; i+=4)
    {
      SIMD_add_2x16(SEaccucommon01,
          SIMD_mul_2x16(SIMD_mul_2x16((int*)pa,
           (int*)pb), (int*)pc));
      SIMD_add_2x16(SEaccucommon23,
          SIMD_mul_2x16(SIMD_mul_2x16((int*)pa,
           (int*)pb), (int*)pc));
      pa+=4; pb+=4; pc+=4;
    }
    s0 = EXTRACT_short_1_of_2(SEaccucommon01);
    s1 = EXTRACT_short_2_of_2(SEaccucommon01);
    s2 = EXTRACT_short_1_of_2(SEaccucommon23);
    s3 = EXTRACT_short_2_of_2(SEaccucommon23);
  }
  else
  {
    for(i=0; i<64; i+=4)
    {
      s0 = s0 + (*pa * *pb) * *pc;
      s1 = s1 + (*(pa+1) * *(pb+1) * *(pc+1);
      s2 = s2 + (*(pa+2) * *(pb+2) * *(pc+2);
      s3 = s3 + (*(pa+3) * *(pb+3) * *(pc+3);
      pa+=4; pb+=4; pc+=4;
    }
  }
  sum = sum + (s0 + s1) + (s2 + s3);

}
```

**Figure 7:** ***Final code after finding all SIMD sets, dynamic alignment check insertion, standard optimizations, and IR cleanup***

# 5. RETARGETING THE SIMD ENGINE

As mentioned in section 3, we developed an interface between LISA ADL processor models and the CoSy compiler generation platform. From the LISA model, a *SIMD configuration* can be generated that stores SIMD-relevant compiler information about the target machine, and that can be embedded into a compiler specification for CoSy. The SIMD configuration first of all captures global parameters such as memory word lengths and alignment requirements of C data types. These parameters control how many operations can be packed together and if memory alignment checking needs to be carried out.

```
typedef struct
{
  char* ruleName;             /* clear-text description */
  char* ckfName;              /* CoSy-internal name of CKF */
  int ckfNum;                 /* CoSy-internal number of CKF */
  opNamesInfo opName;         /* instruction semantics */
  signednessInfo signedness;  /* signedness of the operands */
  int p_in;                   /* fan-in */
  int op_size;                /* operand size in bits */
} simdSignature;
```

**Figure 8:** ***SIMD instruction signature format***

The SIMD instruction combination algorithm from section 4 operates on a *signature* of each available SIMD instruction as shown in fig. 8. The first three components serve for unique identification of each instruction and their association with compiler known functions (CKFs). Component opName stores semantic information, i.e. whether the instruction performs an arithmetic operation, LOAD/STORE etc. This information can be extracted from the ADL processor model. Further signature components store information about signedness, number of operands, and bit width. Fig. 9 shows an example signature for a 4 × 8-bit multiply SIMD instruction. Specialized signature formats are available for more complex arithmetic SIMD instructions and ”extract” operations.

The SIMD configuration enables the execution of the SIMD instruction combination procedure from section 4. However, in order to complete the retargetable compilation flow, the CKF calls in the resulting intermediate code (cf. fig. 7) must be replaced by valid assembly instructions for the target processor. In our framework, this means that the relation between CKFs and assembly must be propagated to the CoSy system.

CoSy uses code selection rules or *mapping rules* for IR to assembly code mapping. Basically, each rule describes how a certain IR operation is mapped to target assembly code. Our existing Compiler Designer tool [12] (cf. fig. 2) comprises techniques to generate mapping rules automatically from an ADL processor model. Hence, in principle this technique could be extended to generate rules for SIMD instructions, too. However, due to practical considerations mentioned in section 4.1, our code selector has been designed to work with CKFs. CoSy compilers know about declared CKFs and handle them essentially within a single mapping rule, where a switch/case construct is used to emit the correct assembly code for each CKF.

The example in fig. 10 shows such a CKF mapping rule in pseudocode. The rule operates on C function calls and checks whether the given function is a CKF. The cost metric is assigned depending on the code optimization objective (speed or size). Assuming there is a CKF usr_ckf_sadd_2_16, the matching target assembly code is finally emitted; in this example a ”dspdualadd” instruction (cf. fig. 1) for the Trimedia VLIW architecture, as well as some ”nop” instructions that may be eliminated later via instruction scheduling. The required entries for the CKF mapping rules can be semi-automatically generated by analyzing the ADL processor model for SIMD instructions and looking up the corresponding assembly instruction syntax.

# 6. EXPERIMENTAL RESULTS

For experimental evaluation we created a SIMD-enabled C compiler with the design flow from fig. 2 for the Philips Trimedia 32 processor [19] and compiled multimedia application kernels, mostly taken from the DSPStone benchmarks [20] and similar to those used in [14] [17] [16]. Furthermore we provide results for more complex DSP algorithms described in table 1.

For the given Trimedia LISA ADL model, the required retargeting effort for SIMD support was several days. A similar workload can be expected for other processors, depending on architecture features. Note we intentionally did not perform a comparison to the native Trimedia C/C++ compiler that comes with sophisticated target-specific optimizations, which would lead to biased results. Instead, we focused on studying the net speedup (measured with a cycle-true instruction set simulator) by integrating the SIMD engine into Compiler Designer while using only retargetable optimizations.

```
  sig = (simdSignature *) malloc(sizeof(simdSignature));
  sig->ruleName   = "unsigned mul 4*8bit";
  sig->ckfName    = "umul_4_8";
  sig->ckfNum     = 100202;
  sig->opName     = MULTIPLICATION;
  sig->signedness = UNSIGNED;
  sig->p_in       = 4;
  sig->op_size    = 8;
```

**Figure 9:** ***Sample SIMD instruction signature instance***

```
RULE f:IR_Function_Call(...)
CONDITION {
  ... /* verify this function is a registered SIMD CKF */
  }
COST 2; /* cost metric for optimized code selection */
EMIT {  /* code for assembly instruction emission */
  switch(function_id)
  {
  ...
  case usr_ckf_sadd_2_16: /* unique CKF id */
    fprintf(OUTFILE,"\tIF r1 nop ,\n\t   IF r1 nop ,
     \n\t  dspidualadd  %s   %s -> %s ,
     \n\t   IF r1 nop ,\n\t   IF r1 nop  ;\n",
      REGNAME(source_reg_1), REGNAME(source_reg_2),
      REGNAME(destination_reg));
    break;
  ...
  }
}
```

**Figure 10:** ***Excerpt of CoSy CKF mapping rule for SIMD***

| benchmark | description |
|---|---|
| quantize | matrix quantization with rounding |
| compress | dct to compress a 128 x 128 pixel image by a factor of 4:1, block size of 8 x 8 |
| idct_8x8 | IEEE-1180 compliant idct, |
| viterbigsm | GSM full rate convolutional decoder |

**Table 1: Benchmark description**

The Trimedia is a 5-slot VLIW DSP with a number of SIMD instructions. Due to its VLIW architecture, using SIMD instructions does not lead to a speedup in all cases. For instance, one can issue 5 parallel ADD instructions simultaneously, while only 2 dual-ADD SIMD instructions can be issued at a time. Furthermore, SIMD instructions may have a higher latency than regular instructions (e.g. 1 cycle for an ADD vs. 2 cycles for a dual-ADD). So, unless the instruction scheduler is not able to find suitable instructions for filling the VLIW slots saved by SIMD, no speedup can be expected. If there is resource pressure, though, SIMD instructions help to reduce the memory bottleneck (at most 2 parallel LOADs/STOREs), provided that a minimum loop unroll factor is applied.

There are also further effects, due to the C coding style or register allocation effects in the compiler backend, that lead to deviations from the theoretical speedup factor $s$ in case of $s$ sub-registers. We quantify our results first for one simple, particular benchmark, i.e. a *dotproduct*, where vector elements are accessed by means of array accesses in the C code:

```
for(i=0;i<N;i++) sum += a[i] * b[i];
```

Fig. 11 gives the speedup achieved by our SIMD engine in relation to the unroll factor $U$ and the number of loop iterations $I$ (i.e. the vector size). Loop unrolling was enabled for both the SIMD and non SIMD version to distinguish the SIMD speedup from the performance effects of loop unrolling. A minimum value of $I$ is needed to compensate setup overhead (e.g. for scalar expansion). Beyond that, the speedup is largely independent of $I$. The speedup does depend on $U$, though, since resource pressure increases with $U$, and so does the effectiveness of SIMD. For $U = 32$, the speedup is asymptotically 2, which corresponds to the theoretical speedup in this case.

Finally, table 2 summarizes the speedup results for the DSPStone benchmarks, described both in array and pointer oriented style, and for a fixed iteration count ($I = 1024$) with loop unrolling enabled for the SIMD and non SIMD version. In addition, the last four lines in table 2 give results for the four more complex DSP routines. In

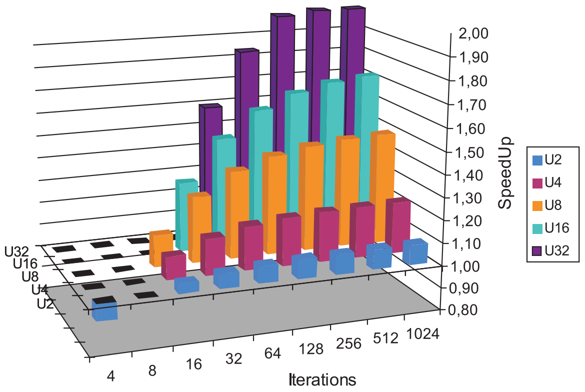

**Figure 11:** ***Speedup for dotproduct***

presence of dynamic alignment checks the SIMD loop version including the alignment check overhead has been measured. A significant speedup was obtained in most cases. The speedup for the complex DSP routines is generally lower, since a smaller fraction of the benchmark code can be mapped to SIMD instructions than in the case of the DSPStone kernels. Still, a speedup of 7% up to 66% was observed. However, there are also counterexamples, such as *n_complex_updates*, where a slowdown has been measured. A code analysis revealed that this is due to subtle scheduling problems in the compiler backend, which could be eliminated by a more sophisticated instruction scheduler. On the other hand, backend effects sometimes also lead to a super-linear speedup (e.g. 2.4 for *matrix1/array/U32*).

As program speedup is the primary objective in utilization of SIMD instructions, we omit detailed results and analysis of code size effects here. Analogous to the speedup, we observed a code size decrease by a factor of 0.6 on average, as compared to unrolled benchmarks without use of the SIMD engine, and a code size increase by a factor of 1.5 for *matrix1x3* which required a dynamic alignment check with a SIMD and non-SIMD loop version in the code.

| **benchmark** | **style** | **U2** | **U4** | **U8** | **U16** | **U32** |
|---|---|---|---|---|---|---|
| vector addition | array | 1.55 | 1.71 | 1.83 | 1.91 | 1.96 |
| fir | array | 1.09 | 1.23 | 1.49 | 1.72 | 2.11 |
| n_real_updates | array | 1.64 | 1.78 | 1.88 | 1.94 | 1.96 |
| n_complex_updates | array | 0.89 | 0.87 | 0.88 | 0.87 | 0.88 |
| dot product | array | 1.09 | 1.23 | 1.49 | 1.72 | 2.11 |
| matrix1 | array | 1.07 | 1.36 | 1.58 | 1.82 | 2.40 |
| mat1x3 | array | 1.08 | 1.28 | 1.49 | 1.74 | 1.85 |
| vector addition | ptr | 1.66 | 1.79 | 1.88 | 1.93 | 1.96 |
| fir | ptr | 1.11 | 1.39 | 1.81 | 2.31 | 2.35 |
| n_real_updates | ptr | 1.75 | 1.85 | 1.92 | 1.96 | 1.96 |
| n_complex_updates | ptr | 1.06 | 1.08 | 1.08 | 1.08 | 1.08 |
| dot product | ptr | 1.11 | 1.39 | 1.81 | 2.31 | 2.35 |
| matrix1 | ptr | 1.11 | 1.39 | 1.81 | 2.30 | 2.58 |
| mat1x3 | ptr | 1.11 | 1.39 | 1.81 | 2.31 | 2.47 |
| quantize (I=64) | array | 1.53 | 1.59 | 1.63 | 1.65 | 1.66 |
| compress (I=8) | array | 1.07 | 1.08 | 1.23 | n/a | n/a |
| idct_8x8 (I=8) | array | 1.08 | 1.09 | 1.11 | n/a | n/a |
| viterbigsm (I=8) | array | 1.09 | 1.10 | 1.11 | n/a | n/a |

**Table 2: Speedup results**

## 7. LIMITATIONS

Our current implementation shows several limitations, whose elimination would probably lead to higher code quality and would allow to handle a wider range of loop constructs. Currently, the preprocessing pass is being improved to take the trade-off between code size increase by loop unrolling and speedup into account to determine the optimal unrolling factor. A further improvement is to use the pointer alignment analysis information to avoid misaligned pointers by loop peeling, to insert data reorganization operations enforcing proper alignment and to exploit unaligned memory operations of certain architectures. Further effort can also be invested in *array index analysis*. We used custom symbolic manipulation functions to handle simple linear index expressions in the compiler IR, but recognizing more complex expressions will be beneficial. Finally, there are limitations imposed by the underlying CoSy platform in its current version concerning the *precision of data dependency* and *alias analysis*, influencing the exposed parallelism, and *instruction scheduling*, responsible for utilization of VLIW slots. Future extensions like *points-to* and *loop-carried dependencies* analysis are required to handle more complex access patterns and to steer loop transformations for better SIMD recognition.

## 8. CONCLUSIONS

In contrast to previous, largely target specific, code optimizations for SIMD instructions, we propose a *retargetable* approach in order to enable SIMD utilization for a wide range of processor architectures at limited manual effort. This is achieved by using a novel SIMD engine that works at a *high level in the compilation flow*, and by using an *ADL based retargeting* technique. This concept has been proven by integrating the SIMD engine within the C compiler generator of an existing industrial ASIP design framework and generating a SIMD-enabled compiler for a realistic DSP processor. While previous backend-oriented SIMD optimization techniques potentially lead to higher code quality, significant speedup results for standard benchmarks were generally obtained with our engine. Hence, we believe that the proposed approach provides a good and practical compromise between code efficiency and compiler flexibility. Future work will concentrate on application to further SIMD target architectures and improvements in code quality by removing the current limitations identified above.

## 9. REFERENCES

[1] M. Gries, K. Keutzer, H. Meyr, et al.: *Building ASIPs: The Mescal Methodology* Springer, 2005

[2] J.A. Fisher: *Customized Instruction Sets for Embedded Processors*, Design Automation Conference (DAC), 1999

[3] A. Oraioglu, A. Veidenbaum: *Application Specific Microprocessors (Guest Editors' Introduction)*, IEEE Design & Test Magazine, Jan/Feb 2003

[4] Associated Compiler Experts: CoSy compiler platform, www.ace.nl

[5] CoWare Inc.: LISATek tools, www.coware.com

[6] P. Mishra, N. Dutt, A. Nicolau: *Functional abstraction driven design space exploration of heterogenous programmable architectures*, Int. Symp. on System Synthesis (ISSS), 2001

[7] A.Kitajima, M. Itoh, J. Sato, A. Shiomi, Y. Takeuchi, M. Imai: *Effectiveness of the ASIP Design System PEAS-III in Design of Pipelined processors*, Asian and South Pacific Design Automation Conference (ASP-DAC), pages 649–654, 2001

[8] Tensilica Inc.: Xtensa C compiler, www.tensilica.com

[9] R. Leupers: *Code Selection for Media Processors with SIMD Instructions*, Design Automation & Test in Europe (DATE), 2000

[10] 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

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

[12] M. Hohenauer, O. Wahlen, K. Karuri, et al.: *A Methodology and Tool Suite for C Compiler Generation from ADL Processor Models*, Design Automation & Test in Europe (DATE), 2004

[13] S. Larsen, S. Amarasinghe: *Exploiting superword level parallelism with multimedia instruction sets*, Proc. Programming Language Design and Implementation (PLDI), 2000

[14] I. Pryanishnikov, A. Krall, et al.: *Pointer Alignment Analysis for Processors with SIMD Instructions*, Proc. 5th Workshop on Media and Streaming Processors, 2003

[15] P. Wu, A. Eichenberger, K. O'Brien: *Efficient SIMD Code Generation for Runtime Alignment and Length Conversion*, Proc. of the International Symposium on Code Generation and Optimization (CGO), 2005

[16] D. Nuzman, I. Rosen, A. Zaks: *Auto-Vectorization of Interleaved Data for SIMD*, IBM Research Report No. H-0235, 2005

[17] Auto-vectorization in GCC, gnu.j1b.org/software/gcc/projects/tree-ssa/vectorization.html

[18] A. Kudriavtsev, P. Kogge: *Generation of Permutations for SIMD Processors*, Proc. Languages, Compilers and Tools for Embedded Systems (LCTES), 2005

[19] Philips: Trimedia architecture. www.semiconductors.philips.com/news/content/file_233.html

[20] 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