A SIMD Optimization Framework for Retargetable Compilers

MANUEL HOHENAUER, FELIX ENGEL, RAINER LEUPERS,
GERD ASCHEID, and HEINRICH MEYR
RWTH Aachen University

Retargetable C compilers are currently widely used to quickly obtain compiler support for new embedded processors and to perform early processor architecture exploration. A partially inherent problem of the retargetable compilation approach, though, is the limited code quality as compared to hand-written compilers or assembly code due to the lack of dedicated optimizations techniques. This problem can be circumvented by designing flexible, retargetable code optimization techniques that apply to a certain range of target architectures. This article focuses on target machines with SIMD instruction support, a common feature in embedded processors for multimedia applications. However, SIMD optimization is known to be a difficult task since SIMD architectures are largely nonuniform, support only a limited set of data types and impose several memory alignment constraints. Additionally, such techniques require complicated loop transformations, which are tailored to the SIMD architecture in order to exhibit the necessary amount of parallelism in the code. Thus, integrating the SIMD optimization *and* the required loop transformations together in a single retargeting formalism is an ambitious challenge. In this article, we present an efficient and quickly retargetable SIMD code optimization framework that is integrated into an industrial retargetable C compiler. Experimental results for different processors 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

Additional Key Words and Phrases: SIMD, vectorization, subword parallelism, retargetable compilers, ASIP

ACM Reference Format:

Hohenauer, M., Engel, F., Leupers, R., Ascheid, G., and Meyr, H. 2009. A SIMD optimization framework for retargetable compilers. *ACM Trans. Architect. Code Optim.* 6, 1, Article 2 (March 2009), 27 pages. DOI = 10.1145/1509864.1509866 <http://doi.acm.org/10.1145/1509864.1509866>

Authors' addresses: Manuel Hohenauer Felix Engel, Rainer Leupers, Gerd Ascheid and Heinrich meyr, Institute for Integrated Signal Processing Systems, RWTH Aachen University, Germany; email: Manuel.hohenauer@iss.rwth-aachen.de.

Permission to make digital or hard copies part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 1544-3566/2009/03-ART2 \$5.00
DOI 10.1145/1509864.1509866 <http://doi.acm.org/10.1145/1509864.1509866>

ACM Transactions on Architecture and Code Optimization, Vol. 6, No. 1, Article 2, Publication date: March 2009.

1. INTRODUCTION

With the increasing acceptance of application specific instruction set processors (ASIPs) [Gries and Keutzer 2005; Fisher 1999; Oraioğlu and Veidenbaum 2003] as efficient and flexible implementation vehicles in embedded system-on-chip (SoC) design, more and more commercial platforms (e.g., CoWare Processor Designer or Tensilica Xtensa) 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. A retargetable compiler has to be as target-independent as possible in order to be applicable for a wide variety of processor types. As a result, such compilers can only make few assumptions about the target machine (i.e., less target-specific hardware features can be exploited to produce efficient code) [Leupers and Marwedel 2001; Leupers 2000a]. 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 the ASIP compiler design effort is to classify target processors according to their architectural features and thus, their demands 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 [Associated Computer Experts by (ACE)]. 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 fashion.

This article focuses on another class of target processors, namely those equipped with SIMD instructions. As illustrated in Figure 1, a SIMD instruction performs several primitive operations in parallel, using operands from several subregisters of the processor's data registers at a time. The operands are typically 8-, 16-, or even 32-bit wide. However, the SIMD data paths might even grow larger with the advances in semiconductor technology. Other typical SIMD instructions perform more complex operations (e.g., partial dot products) or serve for subregister packing and permutation.

From a hardware perspective, SIMD instructions are basically easy to control and have a simple structure (the existing data path is basically just split) without extra register file ports. This makes them inherently simple and thus keeps the hardware cost low. Meanwhile, they can provide significant performance improvements. Therefore, many embedded processors for the next generation of high-end video and multimedia devices today feature SIMD instructions.

Fig. 1. Sample arithmetic SIMD instruction: two parallel ADDs on 16-bit subregisters of 32-bit data registers A, B, C; the data is loaded/stored at once from/to a alignment boundary.

While the SIMD concept was introduced firstly for standard architectures (e.g., Intel MMX/SSE/SSE2/SSE3, IBM/Motorola VMX/Altivec, AMD 3DNow!), it was quickly adopted in ASIPs for multimedia applications (e.g., TI C6x, NXP TriMedia, MIPS), and is being used in today's custom ASIP designs (e.g., Tensilica Xtensa). Therefore, support for SIMD instructions in retargetable compilers is of high interest. Several target-specific C compilers already include SIMD utilization, but it is still very weakly supported in ASIP compilers. For use in this domain, retargetable SIMD optimizations are required.

Therefore, in this article we propose a novel concept for retargetable code optimization for ASIPs with SIMD instructions, and we demonstrate this concept by an implementation within an existing, well-tested retargetable compiler framework and an experimental evaluation for different real-life embedded processors.

The rest of this article is organized as follows. In Section 2, related work is discussed. After the system overview in Sections 3, we describe the core of our approach in Sections 4, 5 (i.e., a framework for exploiting SIMD instructions in the compiler backend as well as the retargeting procedure for this framework). Section 6 summarizes our experiments for two embedded processors with SIMD support. Afterward, Section 7 mentions current limitations of our approach. Finally, Section 8 summarizes the contribution of our work and points to some future avenues of work.

2. BACKGROUND AND RELATED WORK

A key problem in compiler utilization of SIMD instructions is that traditional code generation techniques, such as tree covering with dynamic programming [Muchnick 1997], fail in case of SIMD. Hence, compilers without dedicated techniques are unlikely to exploit SIMD instructions at all. Most of the current SIMD optimization techniques are based on the traditional loop-based vectorization [Allen and Kennedy 1987; Cheong and Lam 1997; Krall and Lelait 2000;

Wu et al. 2005]. Others make use of instruction packing techniques in conjunction with loop-unrolling to exploit data parallelism within a basic block [Larsen and Amarasinghe 2000] or a combination of traditional code selection [Fraser et al. 1992] and Integer Linear Programming [Leupers 2000b; Kudriavtsev and Kogge 2005]. As investigated in Ren et al. [2003], it is often difficult to apply SIMD optimization techniques since SIMD architectures are largely nonuniform, featuring specialized functionalities, constrained memory accesses and a limited set of data types. Moreover, complicated loop transformation techniques are needed [Allen and Kennedy 1987] to exhibit the necessary architecture dependent amount of parallelism in the code. Another hurdle to applying SIMD techniques is packing of data elements into registers and the limitations of the SIMD memory unit: Typically, SIMD memory units provide access only to contiguous memory elements, often with additional alignment constraints. Computations, however, may access the memory in an order which is neither adequately aligned nor contiguous. Besides, operations on disjoint vector elements are not supported directly. The detection of misaligned pointer references is presented in Pryanishnikov et al. [2003]. Certain misalignments can be solved either by loop transformations [Cheong and Lam 1997; Larsen et al. 2002] or by data permutation instructions. The efficient representation and generation of such instructions is investigated in Eichenberger et al. [2004], Wu et al. [2005], and Nuzman et al. [2006] and the optimization thereof in Kudriavtsev and Kogge [2005] and Ren et al. [2006]. Consequently, only a successful interaction of several optimization modules will be able to leverage SIMD optimization for retargetable compilers. Therefore, developing a retargeting formalism for SIMD architectures that integrates the SIMD optimization *and* the required loop transformations is an ambitious challenge.

2.1 Compiler Support for SIMD

Only advanced compilers (e.g., Intel compilers [Intel Corporation], IBM XL compiler [Eichenberger et al. 2004]) provide automatic generation of SIMD instructions. However, they are restricted to certain C language constructs. Moreover, these compilers are inherently nonretargetable. Other compilers use dedicated input languages for source-to-source transformation, which are restricted to a certain application domain [Franchetti et al. 2005; Rizzolo and Padua 2005]. Most C compilers, though, still provide only semiautomatic SIMD support via compiler known functions (CKFs). 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.

ASIP design platforms comprising retargetable C compilers include CoWare Processor Designer [Coware Inc.], Expression [Mishra et al. 2001], Mescal [Gries and Keutzer 2005] and ASIPMeister [Kitajima et al. 2001]. However, no SIMD support has been reported for those tools yet. Tensilica's [Tensilica Inc.] compiler for the configurable Xtensa processor supports SIMD, but it is restricted to a narrow range of architectures. Considering retargetable compilers, recent versions of the *gcc* [GNU Compiler Collection] support SIMD for

certain loop constructs. The supported vectorization [Nuzman and Henderson 2006] features alignment and reduction; however, information regarding the concrete retargeting effort and the interaction of loop transformations are not yet available. Furthermore, *gcc* is mainly designed for general purpose processors. As a result, it does not adapt efficiently to specialized, irregular hardware architectures, which are quite common in the embedded domain.

A retargetable preprocessor for multimedia instructions is presented in Pokam et al. [2004]. The approach mixes loop distribution, unrolling, and pattern matching to exploit SIMD instructions. Contrary to other approaches, it can be extended at user level. The matching is based on a set of target-specific code rewrite rules, which are described using C code patterns. However, the efficiency of this approach strongly depends on the coding style of the input program. Furthermore, no information is available on how the loop transformations are adapted to a given SIMD architecture.

2.2 Our Contribution

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 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. Our SIMD optimization comprises a loop-vectorizer and a unroll-and-pack based technique (an earlier version of the latter is described in Hohenauer et al. [2006]), which are both driven by the same SIMD specification. The retargeting formalism is fully integrated into the compiler backend specification. The advantage is that many generators for the standard backend components (e.g., the code selector) can be reused for the SIMD optimization to a great extent. This reduces the retargeting effort and, moreover, enables a greater flexibility to specify the SIMD architecture. The amount of required target-specific information is limited so that most of it can be extracted automatically from high-level processor models. Moreover, the retargeting information is also used to steer the loop transformations, such as unrolling and strip-mining, required to exhibit the necessary (i.e., SIMD architecture dependent) amount of parallelism and to deal with memory alignment issues. In sum, this provides a flexible and efficient SIMD optimization framework for a wide variety of SIMD architectures. Our approach is integrated into an industrial retargetable C compiler framework and the improvements are shown for different target processors and relevant benchmarks.

3. SYSTEM OVERVIEW

In this article, we employ CoWare's Processor Designer [Coware Inc.] as ASIP design platform. An earlier version of this environment has been described in detail in Hoffmann et al. [2002]. It builds around the Language for Instruction set architectures (LISA) architecture description language (ADL) that describes the behavior, the structure, and the I/O interfaces of a processor architecture. The environment has been used to describe a wide variety of architectures,

including ARM7, C62x, C54x, MIPS32 4K, and to develop ASIPs, such as ICORE [Glökler et al. 2000]. An integrated design environment (IDE) is provided to support the manual creation and configuration of the LISA model. From the IDE, the so called LISA processor compiler is invoked. It parses the description and generates several software development tools like instruction set simulator (suitable for integration in cosimulation environments [Hoffmann et al. 2001]), debugger, profiler, assembler, and linker, and it provides capabilities for VHDL and SystemC generation for hardware synthesis. The retargetable C compiler is seamlessly integrated into this tool chain and uses the same single “golden reference” LISA model to drive retargeting.

The processor designer relies on the CoSy system from ACE for compiler generation. CoSy is a modular C/C++ compiler generation system that offers numerous configuration possibilities both at the level of the intermediate representation (IR) and the backend for machine code generation. The Backend Generator (BEG) is the most important component of the CoSy system. It takes so called code generator description (CGD) files as input and generates most of the backend source code. A CGD model consists mainly of three components:

- (1) a specification of available target processor resources like registers or functional units
- (2) a description of mapping rules, specifying how C/C++ language constructs map to (potentially blocks of) assembly instructions
- (3) a scheduler table that captures instruction latencies as well as instruction resource occupation on a cycle-by-cycle basis

Apart from that, CoSy requires some more information like function calling conventions or the C data type sizes and memory alignment. BEG automatically generates a C/C++ compiler from this information and the CGD model.

The so called compiler designer [Hohenauer et al. 2004] tool extracts compiler-relevant information from a given LISA processor model and translates it to a corresponding CGD description. All relevant machine features are presented to the user who can then refine or modify the generated information. Afterward, CoSy can be invoked as a “back-end” to generate the compiler executable.

CoSy compilers are designed in a highly modular fashion. Each module, called “engine,” works on the compiler’s IR of the input program. The execution order of all engines is freely configurable. New optimization engines can be added mostly in a plug-and-play fashion. Hence, we added a retargetable SIMD framework, consisting of several separate engines, into the CoSy platform that implements the techniques described in detail in Section 4. Due to the coupling to processor designer, the SIMD properties of the target processor can be described within the same golden LISA model that drives the entire ASIP design process, and they can be automatically translated into the CGD format (see Section 5). This tool flow, illustrated in Figure 2, enables a seamless and retargetable path from the target machine model to a SIMD-enabled C compiler.

Fig. 2. Tool flow for retargetable compilation with SIMD support.

4. SIMD FRAMEWORK

It is known that a successful SIMD optimization is tightly coupled with several loop transformations [Allen and Kennedy 1987] in order to create the necessary amount of parallelism and to convert loops into a proper form. Hence, our approach consists of several steps as depicted in Figure 2. First of all, loop carried dependency [Wolfe 1995] and alignment analysis (Section 4.3) are performed. They provide the necessary annotation needed by our SIMD optimization framework. Afterward, a SIMD analysis (Section 4.4) searches for loops where SIMD optimization could be applied. For these loops it determines the parameters for the different *loop transformations* (Sections 4.5 through 4.8). Finally, the SIMD optimization is performed, comprising a loop *vectorizer* (Section 4.7) or an unroll-and-pack based *SIMDfyer* (Section 4.9), if vectorization fails. All modules are driven by the same retargetable SIMD specification described in Section 5.

4.1 Basic Design Decisions

A basic design decision concerns the representation of generated SIMD instructions in the compiler's IR. All IR formats comprise elements for representing primitive operations like addition, subtraction, multiplication, and so on. However, there is usually no notion of SIMD operations, such as “two parallel

additions.” Hence, an extension to the underlying IR format would be required. From a practical viewpoint, such extensions have a dramatic impact on most “downstream” compiler engines. They either demand 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). Typically, IR formats allow to precisely define the side-effects of each CKF. As a result, CKFs are transparent for other compiler engines and therefore cause no problems.¹ Moreover, CKFs simplify code generation, since they abstract from low-level problems like register allocation for SIMD subregisters in the backend. In addition, all existing “downstream” code generation and optimization engines of the underlying compiler framework can be reused. As a side effect, the current IR can be dumped into a human-readable, valid C code file anytime for debugging purposes during the SIMD generation process.

4.2 Terminology

Here, we briefly introduce the terminology that facilitates the description of our modules in the next sections. As exemplified in Figure 1, a SIMD instruction performs independent, usually identical operations on a certain bit range within the input register and also writing the results to a corresponding range in the output register. In other words, a SIMD instruction splits a full register into k subregisters (frequently $k = 2$ or $k = 4$). In the given example, the lower and upper parts of the arguments are added and written to the lower and upper part of the destination register, respectively. Thus, this SIMD instruction operates on 2 subregisters. A single, primitive operation within the SIMD instruction (e.g., the 16-bit addition) is denoted as a SIMD candidate. It is basically a tree pattern covering this primitive operation. From these tree patterns a SIMD candidate matcher (Section 5.1) is generated (i.e., a regular tree pattern matcher) that is used for the identification of such SIMD candidates.

We denote a set of SIMD candidates that can be combined into a SIMD instruction as a SIMD-set. For this purpose we employ a generated SIMD-set constructor (Section 5.2). This is basically a combination function that tries to collect suitable SIMD candidates under given constraints such that a valid SIMD-set can be built.

The algorithm for SIMD-set constructions assumes that the results from the data flow analysis are already available. Next, it checks a number of constraints for tuples $N = (n_1, \dots, n_k)$ of SIMD candidates, where k denotes the number of subregisters. Amongst others², nodes n_i of a potential SIMD-set must

¹It 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.

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

Fig. 3. SIMD alignment constraint.

- (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

A constructed SIMD-set (i.e., the related IR nodes) can then be replaced by a CKF call. The regular code selector description is enriched with CKF mapping rules so that later during the code emission phase the proper assembly code for the SIMD instruction can be emitted.

4.3 Alignment Analysis

The SIMD memory unit usually implies certain constraints on the memory access. For example, a twofold SIMD instruction operating on 16-bit data types typically uses a 32-bit wide, word aligned load operation to pack them at once in a 32-bit register (Figure 3). If the word alignment cannot be assured at compile time, additional code (i.e., a dynamic alignment check) is required to ensure correct alignment during runtime. The strip mining transformation (Section 4.5) needs to take the alignment into account too. Therefore, we implemented an interprocedural pointer alignment analysis similar to Pryanishnikov et al. [2003] for precise alignment information. It analyses every memory access done through pointers with respect to the capabilities of the SIMD memory unit. The offset from the supported SIMD memory boundary, that is, the alignment, is calculated using the modulo operator. If p is a pointer and N the SIMD memory address size, then the alignment of the memory access is given by:

$$alignment = p \bmod N \quad (1)$$

Since each pointer might have different values and associated alignments, the information is stored as a set E of possible values *modulo* N . If $M = \{0, \dots, N - 1\}$ is the set of all possible values modulo N and $P = P(M)$ its

power set, then $E \in P$. In order to evaluate pointer arithmetic, such as $*(p+i)$, a transfer function

$$f_g : P^n \mapsto P \quad (2)$$

is used to compute the impact on E . The transfer function, naturally, depends on the operator of the arithmetic expression. For example, the most common operations in address calculation, the addition and multiplication, are binary operators and thus, the corresponding transfer functions have the form $f_{Binary} : M \times M \mapsto M$. This leads to the following equations:

$$\begin{aligned} f_{Add}(a, b) &= (a + b) \bmod N = [(a \bmod N) + (b \bmod N)] \bmod N \\ f_{Mul}(a, b) &= (a \cdot b) \bmod N = [(a \bmod N) \cdot (b \bmod N)] \bmod N \end{aligned} \quad (3)$$

Similar transfer functions exist for the remaining operators.

4.4 SIMD Analysis

The preparative loop transformations consist of strip mining, scalar expansion and loop unrolling. They must be parameterized according to the underlying SIMD architecture. Incorrect parameters might prevent SIMD optimization or lead to nonoptimal results. The transformations often only pay off if they enable the SIMD optimization afterward. Therefore, it is important to apply them only to the most promising loops for SIMD optimization. Hence, we implemented a SIMD analysis engine that runs in advance to identify those loops, which contain SIMD candidates at all. For this purpose, we employ the SIMD candidate matcher. Consequently, if the loop body does not contain any SIMD candidates then it does not make sense to consider it further. Otherwise, the SIMD analysis determines for each SIMD candidate how many of them would be needed to build a SIMD-set that matches one of the available SIMD instructions using the SIMD-set constructor. From this information, it derives the parameters for the different loop transformations.

4.5 Strip Mining and Loop Peeling

Many vectorizable loops cannot be directly optimized in case the iteration count is larger than the number of SIMD candidates k_s that fit into a SIMD-set s for the vector operation. Strip mining is a loop transformation that divides the loop into stripes, where each strip is no longer than the SIMD data path width [Wolfe 1995]. Essentially, the loop is decomposed into two nested loops: an outer loop (the strip loop), which steps between strips and an inner loop (the element loop), which steps between single iterations within a loop (Figure 4(a)). The SIMD analysis calculates the iteration count of the element loop, called the strip size, based on all SIMD-sets S that can be built with the identified SIMD candidates in the loop. Since it might happen that each SIMD-set has a different number of subregisters k , we select the maximum strip size for the transformation:

$$strip_size = \max \left(\bigcup_{s \in S} k_s \right) \quad (4)$$

<pre>// original loop for (i=iFrom; i< iTo; i++) { A[i+offset] = B[i+offset] * C[i+offset]; } //strip loop //strip_size = max. #sub-registers for (is = iFrom; is < iTo; is += strip_size) { //element loop for (i=is; i<iis+strip_size; i++) { A[i+offset] = B[i+offset] * C[i+offset]; } }</pre>	<pre>// peeled iterations for (i = iFrom; i < iFrom+(mod(-(iFrom+offset),strip_size)); i++) { A[i+offset] = B[i+offset] * C[i+offset]; } //strip mined loop for (is = iFrom + mod(-(iFrom+offset),strip_size); is < iTo - mod(iTo+offset,strip_size); is += strip_size) { for (i=is; i<iis+strip_size; i+=1) { A[i+offset] = B[i+offset] * C[i+offset]; } }</pre>
--	---

(a) Strip Mining with offset = 0

(b) Strip Mining with offset != 0

Fig. 4. Simple strip mining examples.

<pre>for (i=0; i < N; i++) { s = B[i] * C[i]; A[i] = s+1/s; }</pre>	<pre>for (i=0; i<= N; i++) { S[i] = B[i] * C[i]; A[i] = S[i]+1/S[i]; } s = S[N];</pre>
--	---

(a) Scalar s causes antidependence

(b) Replaced scalar with array access

Fig. 5. Scalar expansion examples.

However, due to possible alignment constraints of the SIMD architecture, strip mining must ensure that each strip starts at an alignment boundary. For instance, if the iteration counter i starts at zero and the loop contains array indices of the kind $[i + \text{offset}]$ with $\text{offset} \neq 0$, then the alignment boundaries are given by:

$$\text{alignment_boundaries} = \{i \mid i + \text{offset} \bmod \text{strip_size} = 0\} \quad (5)$$

This information is already provided by the alignment analysis. If the offset remains constant within the loop, it can be eliminated by loop peeling. That means, those iterations causing the misalignment are “peeled off” the original loop and build a loop on their own (Figure 4(b)). Note that the modulo operation must produce a value in the range $[0, \text{strip_size})$. Furthermore, it must take care of overflows that might occur during the computation of the loop boundaries.

4.6 Scalar Expansion

When scalars are assigned and later used in the loop, the dependence graph will include flow dependence relations from the assignment to each use and loop carried antidependences from each use back to the assignment. These antidependence relations often cause problems in other transformations and could prevent parallelization of the loop (Figure 5(a)). However, the antidependence relation can be broken by scalar expansion [Wolfe 1995]. The basic idea is to allocate an array with one element for each iteration and replace each scalar

Fig. 6. Vectorization example.

reference in the loop with a reference to the array. This will eliminate the anti-dependence relations. The computed value should be assigned to the original scalar after the loop (Figure 5(b)). One obvious drawback of scalar expansion, though, is the increased memory consumption of the program. If not carefully managed, this penalty can overcome the benefits gained by SIMD. In our implementation, we reduce the memory usage (if applicable) by strip mining the loop and only expanding the inner element loop.

4.7 The Vectorizer

A classical vectorizer parallelizes the whole loop at once provided that suitable SIMD instructions are available for all statements and no data dependencies limit parallelization. Another prerequisite is that the iteration count must match the number of SIMD candidates needed to build the SIMD-set for the vector operation. Obviously, this is a perfect match for strip mined loops. The vectorization algorithm is exemplified in Figure 6. In the first step (1), it checks all inner loops whether each statement consists only of SIMD candidates using the SIMD candidate matcher. In step (2), it virtually duplicates the SIMD candidates according to the iteration count of the current loop. For these virtual SIMD candidates, it tries then to construct a SIMD-set that matches an available SIMD instruction with the SIMD-set constructor (3). Finally, if valid SIMD-sets can be constructed for each statement then the whole loop will be replaced by the corresponding SIMD instructions (4).

Of course, in many loops not all statements can be directly parallelized (e.g., due to data dependencies). But still they may contain a certain degree

of parallelism. Therefore, loops which could not be vectorized are further processed by the more powerful unroll-and-pack based SIMDfyer.

4.8 Loop Unrolling

The SIMDfyer implements a technique similar to Larsen and Amarasinghe [2000]. This requires loops to be unrolled properly to ensure full utilization of the SIMD data path. The SIMD analysis customizes the unroll factor to the number of SIMD-candidates k_s that fit into a SIMD-set s that can be constructed for the given loop body. This is basically the same as for the strip size calculation. Consequently, strip mined loops will be unrolled completely if they are not vectorized. It may happen that the loop contains several SIMD candidates, which can be combined in different ways to a SIMD-set. Thus, since it is desired to fill all possible SIMD-sets S , the best unroll factor can be calculated as:

$$\text{unroll_factor} = \max \left(\bigcup_{s \in S} k_s \right) \quad (6)$$

The SIMD analysis annotates the unroll factor to each loop that contains SIMD candidates. The value of all loops left after vectorization will be read by the loop unrolling engine to prepare them for the SIMDfyer.

4.9 The Unroll-and-Pack Based SIMDfyer

For a given IR of an input C program, we use an iterative algorithm that combines SIMD candidates into SIMD-sets and replaces such sets by CKFs in the IR. Even though the algorithm could in principle process all basic blocks inside a procedure, it focuses only on the loops, typically the hot spots of the input program. More specifically, only those loop bodies in which the SIMD analysis identified SIMD candidates before. Certain multiple basic block constructs, though, may have been merged into a single basic block by an if-conversion [Allen et al. 1983] pass prior to the SIMD optimization. The algorithm forms SIMD instructions step by step. If a complete SIMD-set could be built, it will be replaced by the corresponding CKF. Since each iteration may generate new SIMD candidates, the list of SIMD candidates is updated after each step. The identification of SIMD candidates is performed by the SIMD candidate matcher. The basic idea of the iteration is illustrated in Figure 7.

State (1) shows the initial IR structure for a sample loop body (unrolled twice) that performs a multiplication of two vectors B , C and stores the result in vector A . The left and right elements of the computations are isomorphic and are assumed to meet the memory alignment constraints. Firstly, the algorithm combines the left and the right operands (16-bit load operations) of the two “*” to 32-bit SIMD load operations. Afterward, the “*” operations themselves are combined to a SIMD instruction. The corresponding IR has the intermediate state (2). In order to preserve the semantic correctness, explicit “extract” operations are inserted that select 16-bit subwords out of the 32-bit result of the SIMD dual multiplication operation. These extracts are also considered as SIMD candidates and hence, can also be used to build a SIMD-set. Note, all superfluous extracts are removed by dead code elimination in a later

Fig. 7. IR states in different iterations.

compilation phase. In the following iteration, the two 16-bit “=” operations form a SIMD-set on their own. Finally, the IR state (3) is reached and the algorithm terminates.

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 the corresponding code duplication.

4.10 Code Example

We provide a more detailed example to illustrate the representation of SIMD instructions in the IR. Figure 8(a) shows the initial C source code after preprocessing (strip mining, scalar expansion and loop unrolling). We assume availability of SIMD instructions for addition and multiplication operating on two 16-bit values. Thus, the SIMD analysis determines a strip size and an unroll factor of two for the loop transformations. Here, scalar expansion is performed on the element loop, which is then fully unrolled afterward. It is further assumed that the target machine requires SIMD load operations to be word aligned.

In the first iteration, the SIMDfyer identifies that two pairs of 16-bit operands can be loaded at once. Moreover, subregister extract functions (EXTRACT_short_x_of_2) are inserted to preserve the semantic correctness of the code, and temporary variables for intermediate results are allocated.


```

void dotproduct(short *pa;
               short *pb;
               short *pc)
{
    short sum;
    short S[2];
    sum = 0;
    S[0] = S[1] = 0;
    for (int is = 0; is < 64; is += 2)
    {
        S[0] = S[0] + (*pa * *pb) * *pc;
        pa++; pb++; pc++;
        S[1] = S[1] + (*pa * *pb) * *pc;
        pa++; pb++; pc++;
    }
    sum = sum + S[0] + S[1];
}

```

(a) Initial code.

```

void dotproduct(short *pa,short *pb,short *pc)
{
    short sum;
    short S[2];
    int tmp1, tmp2;
    short res0, res1, res2, res3;
    sum = 0;
    S[0] = S[1] = 0;

    for (int is = 0; is < 64; is += 2)
    {
        tmp1 = (int*)pa; //SIMD load
        tmp2 = (int*)pb; //SIMD load
        res0 = EXTRACT_short_1_of_2(tmp1);
        res1 = EXTRACT_short_2_of_2(tmp1);
        res2 = EXTRACT_short_1_of_2(tmp2);
        res3 = EXTRACT_short_2_of_2(tmp2);

        S[0] = S[0] + (res0 * res2) * *pc;
        pa++; pb++; pc++;
        S[1] = S[1] + (res1 * res3) * *pc;
        pa++; pb++; pc++;
    }
    sum = sum + S[0] + S[1];
}

```

(b) Intermediate code after the first iteration.

```

void dotproduct(short *pa, short *pb,
               short *pc)
{
    short sum;
    short S[2];
    int is;

    int tmp1,tmp2,tmp3;
    short res0,res1,res2,res3,res4,res5;

    sum = 0;
    S[0] = S[1] = 0;

    for(is=0; is<64; is+=2)
    {
        tmp1 = (int*)pa;
        tmp2 = (int*)pb;
        res0 = EXTRACT_short_1_of_2(tmp1);
        res1 = EXTRACT_short_2_of_2(tmp1);
        res2 = EXTRACT_short_1_of_2(tmp2);
        res3 = EXTRACT_short_2_of_2(tmp2);
        tmp3 = SIMD_mul_2x16(tmp1, tmp2);
        res4 = EXTRACT_short_1_of_2(tmp3);
        res5 = EXTRACT_short_2_of_2(tmp3);

        S[0] = S[0] + res4 * *pc;
        pa++; pb++; pc++;
        S[1] = S[1] + res5 * *pc;
        pa++; pb++; pc++;
    }
    sum = sum + S[0] + S[1];
}

```

(c) Intermediate code after the second iteration.

```

void dotproduct(short *pa,short *pb,short *pc)
{
    short sum;
    short S[2];
    int is;

    sum = 0;
    S[0] = S[1] = 0;

    if( ((pa|pb|pc) & 3) == 0 )
    {
        for (is = 0; is < 64; is += 2)
        {
            (int) S[0] = SIMD_add_2x16((int)S[0],
                                     SIMD_mul_2x16(
                                         SIMD_mul_2x16((int*)pa,
                                                         (int*)pb),
                                                         (int*)pc));
            pa+=2; pb+=2; pc+=2;
        }
    }
    else
    {
        for(is=0; is < 64; is += 2)
        {
            S[0] = S[0] + (*pa * *pb) * *pc;
            pa++; pb++; pc++;
            S[1] = S[1] + (*pa * *pb) * *pc;
            pa++; pb++; pc++;
        }
    }
    sum = sum + S[0] + S[1];
}

```

(d) Final code after finding all SIMD sets, dynamic alignment check insertion, standard optimizations, and IR clean-up.

Fig. 8. Code examples.

Figure 8(b) depicts the code after the first iteration (as generated by the IR-to-C code dump facility of our compiler platform). In the next iteration, the two multiplications are detected as SIMD candidate and are replaced by a CKF (`SIMD_mul_2x16`). The SIMD multiplication implies certain conditions in which subregisters the input operands must be located in. Since the input operands are given by the extract operations from the previous iteration, these conditions

Fig. 9. Pos/id for array/scalar variable.

can be easily met by directly using the temporaries the input operands are extracted from. Obviously, this makes the extract operation from the previous iteration superfluous. The resulting code is depicted in Figure 8(c).

Figure 8(d) shows the final code after several further steps. The SIMD-set computation has been finalized by detecting that the multiply results can be processed further by SIMD additions. No extract operations are required, since the results can be directly written by a wide store to the array created by 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.

5. RETARGETING THE SIMD FRAMEWORK

To retarget the SIMD Framework basically two pieces of information are required: Firstly, a description of IR tree patterns, which represent a SIMD candidate. This is used to generate a so called SIMD candidate matcher. Secondly, the SIMD-set construction, the specification of how SIMD candidates can be composed to a valid SIMD-set.

5.1 SIMD Candidate Matcher

The identification of SIMD candidates is basically the same technique as used for tree covering based code selection [Muchnick 1997]. Therefore, they can easily be described by regular code selector rules. Normally, such a rule describes how a certain IR operation is mapped to target assembly code. Nonterminals, typically the rule operands, are used as “temporaries” to transfer values from one rule to another. From this specification, a tree pattern matcher for code selection can be generated with tools like Burg [Fraser et al. 1992]. In our case, the standard CoSy tree pattern matcher generator is used to create a dedicated SIMD candidate matcher from SIMD candidate rules, which are part of the regular code selector description.³ Such rules use special *SIMD* nonterminals containing two specific attributes: A *pos* field for the subregister number within a full register and an *id* to identify a memory area, for example, allocated by a scalar variable or an array (Figure 9).

³This is no contradiction to the limitations of tree pattern matching mentioned in Section 2. The matcher is only used to identify those IR operations, which might be composed to a full SIMD operation, the complete SIMD match cannot be found directly.

<pre> \\Syntax is name:type RULE [load] o:IR_Load(src:reg_nt) -> dst:simd_nt; CONDITION { IS_INT16(o) } EMIT { dst.pos = get_pos(o); dst.id = get_id(o); } </pre>	<pre> RULE [add] o:IR_add(src1:simd_nt, src2:simd_nt) -> dst:simd_nt; CONDITION { IS_INT16(o) && src1.pos == src2.pos } EMIT { dst.pos = src1.pos; dst.id = newid(src1.id,src2.id); } </pre>
--	---

(a) SIMD candidate rule named load for a 16-bit load. (b) SIMD candidate rule named add for a 16-bit add.

Fig. 10. SIMD candidate rules.

As will be explained later in more detail, the former is needed to check subregister or alignment constraints and the latter becomes important when the packed result of a SIMD operation is directly consumed by another one. The initial values for these fields are already determined by the prior dataflow/alignment analysis and are initialized when a load operation is matched. Furthermore, each rule can be referenced using its unique rule name. Examples for two SIMD candidate rules named load and add are shown in Figure 10.

The 16-bit load rule initializes the SIMD nonterminal's pos and id fields with the values determined by dataflow/alignment analysis. The produced SIMD nonterminal may then be consumed by the add rule. Additional conditions can be used to select only those IR operators for a certain data type or to specify constraints on the subregister of the operands. In this example, the 16-bit add rule matches only if both input operands are located in the same subregister. Additionally, rules to extract a subregister from a full register must be created as well. Those are used to match the inserted extract operations (see Section 4.10) in order to reuse results from the previous iterations of the algorithm. In this way, they become SIMD candidates in the current iteration. All extract rules produce a SIMD nonterminal, which sets id to the id of the temporary the result is extracted from and the pos field to the position of the extracted subregister, respectively.

The SIMD candidate matcher's flexibility is only limited by the capabilities of the underlying tree pattern matcher generator. Since the concepts are already supported by the existing code selector description, only minimum changes to the retargetable compiler platform are required. Since tree covering based code selection is the state of the art in compiler design, this part can also be easily ported to other platforms.

5.2 SIMD-Set Constructor

Special SIMD rules describe valid tuples $N = (n_1, \dots, n_k)$ of SIMD candidates, where k denotes the number of subregisters. In contrast to regular mapping rules, they take the names of SIMD candidate rules instead of nonterminals as input operands. The examples in Figure 11 specify a twofold 16-bit load and add SIMD instruction, using the SIMD candidate rules from Figure 10:

Based upon this specification, a valid SIMD-set N can be derived as follows. Given the set of all identified SIMD candidates $C = \{c_1, c_2, \dots\}$, the set of all

<pre> SIMD RULE simd_load(a:load, b:load); COMPOSITION CKF#1 (src:a.src) -> dst:reg_nt(a.dst, b.dst); EMIT { printf("LOAD32 [%s] -> %s", REGNAME(src), REGNAME(dst)); } </pre>	<pre> SIMD RULE simd_add_2x16 (a:add, b:add); COMPOSITION CKF#2 (arg1:reg_nt(a.src1, b.src1), arg2:reg_nt(a.src2, b.src2)) -> dst:reg_nt (a.dst, b.dst); EMIT { printf ("\tDUALADD16\t%s,%s -> %s", REGNAME(arg1), REGNAME(arg2), REGNAME(dst)); } </pre>
(a) SIMD rule for a twofold 16-bit load	(b) SIMD rule for a dual 16-bit add

Fig. 11. SIMD rules to describe a SIMD-set.

possible SIMD-sets S is given by $S \subseteq \mathcal{P}(C)$, whereas each tuple in S must be in the set of all SIMD rules R as defined in the compiler configuration. Furthermore, it must match certain implicit conditions. Let $\text{Pos}(c)$ denote the pos value of the result SIMD nonterminal produced by SIMD candidate rule c and $\text{Id}(c)$, the id, respectively. Then the set of valid SIMD-sets S is given by:

$$S = \{(c_1, \dots, c_k) \mid (c_1, \dots, c_k) \in R \wedge \text{Id}(c_i) = \text{Id}(c_j) \wedge \text{Pos}(c_{l+1}) = \text{Pos}(c_l) + 1, \\ \forall i, j \in (1, \dots, k), l \in (1, \dots, k-1)\} \quad (7)$$

In other words, the SIMD candidates of a valid SIMD-set must have the same id as well as an increasing pos value assigned. It might happen that there exist several possibilities of merging candidates to a valid SIMD-set. For the time being, the current combination algorithm does not perform a cost analysis. It just selects the first valid SIMD-set that could be built from the SIMD candidates.

Consider the example shown in Figure 12(a). In the first iteration, the load rule covers the array accesses, initializes the id with a unique number, and the pos field with the position relative to a SIMD load memory boundary. Note that accesses to the same array get always the same id assigned while only the pos field varies. It is assumed here that the arrays are aligned to a word boundary. Now, due to the implicit condition of the SIMD_load, the only way to create a complete SIMD-set is to combine two adjacent loads (i.e., increasing pos) from the same id. All other combinations would violate at least one constraint. Both SIMD_loads create a temporary with a new id. Afterward, the operations to extract the subregisters have been inserted as well. As mentioned above, the extracts create also new temporaries, which get the same id assigned as the temporary the subregister is extracted from, and the pos field is set to the extracted subregister number respectively.

In the next iteration (Figure 12(b)), the first and second operand of the first two additions share the same ids. Consequently, the same id is generated for both results of the additions. Now they can be combined to a SIMD_add. The implicit id condition actually enforces that the packed operands of the previous SIMD_load are directly reused, otherwise this might result in an expensive repacking of the operands if for instance, the first addition is combined with the fourth addition. Note that it is also possible to specify an explicit condition for the SIMD rules to overwrite the defaults for pos and id. As an example,

```

for(i=0; is < 64; i += 2)
{
  //      <pos=0,id=1>   <pos=0,id=2>
  a[i]   = b[i]   +   c[i];
  //      <pos=1,id=1>   <pos=1,id=2>
  a[i+1] = b[i+1] +   c[i+1];
  //      <pos=0,id=3>   <pos=0,id=4>
  x[i]   = y[i]   +   z[i];
  //      <pos=1,id=3>   <pos=1,id=4>
  x[i+1] = y[i+1] +   z[i+1];
}
// In the 1st iteration:
// load -> <pos=0,id=1>, ...
// SIMD_load(<pos=0,id=1>,<pos=1,id=1>)
//      -> <pos=0,id=5>
// SIMD_load(<pos=0,id=2>,<pos=1,id=2>)
//      -> <pos=0,id=6>
// EXTRACT_short_1_of_2(<pos=0,id=5>)
//      -> <pos=0,id=5>
// EXTRACT_short_2_of_2(<pos=1,id=5>)
//      -> <pos=1,id=5>
// EXTRACT_short_1_of_2(<pos=0,id=6>)
//      -> <pos=0,id=6>
// EXTRACT_short_2_of_2(<pos=1,id=6>)
//      -> <pos=1,id=6>
// ...

```

(a) SIMD rule conditions in first iteration.

```

for(i=0; is < 64; i += 2)
{
  //<pos=0,id=5>
  tmp1 = (int*)(b+i);
  //<pos=0,id=5>
  res0 = EXTRACT_short_1_of_2(tmp1);
  //<pos=1,id=5>
  res1 = EXTRACT_short_2_of_2(tmp1);
  //<pos=0,id=6>
  tmp2 = (int*)(c+i);
  //<pos=0,id=6>
  res2 = EXTRACT_short_1_of_2(tmp2);
  //<pos=1,id=6>
  res3 = EXTRACT_short_2_of_2(tmp2);
  ...
  //      <pos=0,id=5>   <pos=0,id=6>
  a[i]   = res0   +   res2;
  //      <pos=1,id=5>   <pos=1,id=6>
  a[i+1] = res1   +   res3;
  ...
}
// In the 2nd iteration:
// add(<pos=0,id=5>,<pos=0,id=6>)
//      -> <pos=0,id=56>
// add(<pos=1,id=5>,<pos=1,id=6>)
//      -> <pos=1,id=56>
// SIMD_add(<pos=0,id=56>,<pos=1,id=56>)
// ...

```

(b) SIMD rule conditions in second iteration.

```

RULE [CKF#2] o:IR_FuncCall( arg1:reg_nt,arg2:reg_nt)
                        -> dst:reg_nt;
CONDITION {
  CKF_Number(o) == CKF#2
}
EMIT {
  printf ("\tDUALADD16\t%s,%s -> %s",
          REGNAME(arg1),
          REGNAME(arg2),
          REGNAME(dst));
}

```

(c) Internally generated CKF rule for SIMD_add_2x16.

Fig. 12. Generated CKF rule and code examples for rule conditions.

conditions on the pos fields can be used to model unaligned SIMD memory operations.

In order to complete the retargetable compilation flow, the CKF calls in the resulting intermediate code and must be replaced by valid assembly instructions for the target processor. In our framework, the COMPOSITION for a SIMD rule specifies the CKF call, which is internally generated for an identified SIMD-set. It consists of a unique CKF number, the argument(s) to be passed to the CKF call and the assembly code that is finally emitted. For example, the COMPOSITION for SIMD_add_2x16 describes that the arguments for the CKF call are register non-terminals, which contain the first and second operand of the combined add rules. From this specification, a regular code selector rule matching the CKF with the given number and assembly syntax is automatically generated (Figure 12(c)) and becomes part of the regular backend code selector.

Like for the SIMD candidate matcher, many concepts are already supported by the existing tree pattern matcher generator. Thus, only a few changes are required to the existing generator to support our approach.

Table I. Benchmark Description

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,
viterbi	GSM full rate convolutional decoder
emboss	Converts an image using an emboss filter
sobel	Applies a sobel filter to an image
corr_gen	Generalized correlation with a 1 by M tap filter

6. EXPERIMENTAL RESULTS

For experimental evaluation, we created SIMD-enabled C compilers with the design flow from Figure 2 for the NXP TriMedia 32 processor [NXP Semiconductors] and the ARM11 [Advanced RISC Machines Ltd.]. In contrast to the AltiVec or SSE extension, both architectures support SIMD only for short (i.e., 8-bit and 16-bit) integer data types, which is quite common for embedded processors. Hence, benchmarks employing floating point computations cannot be used. Therefore, we chose benchmarks mostly from the DSPStone benchmark suite [Zivojnovic et al. 1994] and implemented kernels similar to those used in Pryanishnikov et al. [2003], GNU Compiler Collection, Nuzman et al. [2006]. Furthermore, we provide additional results for more complex DSP algorithms listed in Table I.

For the given TriMedia and ARM LISA ADL models, the required retargeting effort for SIMD support is quite limited. The corresponding CGD descriptions for SIMD consist of 393 (TriMedia) and 698 (ARM) lines of code, which accounts for roughly 7% (TriMedia) and 14% (ARM) of the complete CGD description. A similar workload can be expected for other processors, depending on architecture features. Note, we intentionally did not perform a comparison to the native C/C++ compilers for these architectures, which come with sophisticated target-specific optimizations, which would lead to biased results. Instead, we focused on studying the net speed-up (measured with a cycle-true instruction set simulator) by integrating the SIMD engine into the compiler designer while using only retargetable optimizations. Since loop unrolling alone already has a large impact on the overall performance, we measured the speed-up by using the following equation:

$$Speed-up = \frac{cycles_{Unroll}}{cycles_{Vector+SIMDfyer}} \quad (8)$$

$Cycles_{Unroll}$ denotes the number of cycles the test kernel needed when compiled with unrolling turned on, but the SIMD engines (i.e., Vectorizer and SIMDfyer) turned off. $Cycles_{Vectorizer+SIMDfyer}$ denotes the number of cycles the kernel needed when compiled with the same unrolling factor and the SIMD engines activated. Hence, the speed-up is only due to the SIMD instructions. All other compiler parameters have always been identical.

6.1 NXP TriMedia

The TriMedia is a 5-slot VLIW DSP with 128 general purpose registers and a number of SIMD instructions. It features several SIMD instructions to process byte or half-word data values in 32-bit registers. Due to its VLIW architecture, using SIMD instructions does not lead to a speed-up in all cases. For instance, one can issue five parallel ADD instructions simultaneously, while only two dual-ADD SIMD instructions can be issued at a time. Furthermore, SIMD instructions may have a higher latency than regular instructions (e.g., one cycle for an ADD vs. two 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 speed-up can be expected. However, if the memory is the bottleneck (at most two parallel LOADs/STOREs), SIMD instructions still help to reduce the memory pressure. 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 speed-up factor s in case of s subregisters. The memory is organized in 32-bit words, which requires a word alignment for SIMD memory accesses.

6.2 ARM11

The ARM architecture [Advanced RISC Machines Ltd.] is built around a central, scalar RISC core. It has a register file, which consists of 31 general purpose registers (at any one time only 16 register are visible) and six status registers. The memory is also organized in 32-bit words. It requires the same word alignment for all memory accesses as the TriMedia. The ARM11's instruction set supports only a limited set of SIMD instructions, which consists of additions and subtractions of byte or half-word data values in 32-bit registers. Furthermore, the ARM features a complex dotproduct support operation, that multiplies two pairs of half-words in parallel, and adds the two resulting word wide values to an accumulator. Since there is no direct SIMD multiplication operation available, kernels that do not match this dotproduct support operation cannot be optimized.

6.3 Evaluation

We quantify our results first for one simple, particular benchmark, that is, 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];
```

Due to the dependency on `sum`, a scalar expansion has to be applied to the loop before SIMD instructions can be inserted. First of all, we investigate the impact of the alignment analysis and the overhead introduced by scalar expansion. Figure 13 shows the speed-up over the number of loop iterations I with (static) and without (dynamic) alignment analysis using a fixed unroll factor of 4. It can be clearly seen that a certain iteration count is required to compensate the overhead by scalar expansion until SIMD pays off. Beyond that, the speed-up is largely independent of I . For high iteration counts, the speed-up

Fig. 13. Speed-up over loop iterations for dotproduct.

is asymptotically 2, which corresponds to the theoretical speed-up in this case. However, the version without the dynamic alignment check reaches the break-even point considerably faster than the one with the checks. The reason for the extremely high speed-up obtained on the ARM processor is due to type conversions. Since the multiplications in the non-SIMD version produce results of 32 bits size, these have to be converted to 16-bit width afterward. The ARM compiler generates a sequence of a logical left shift by 16 bits, followed by an arithmetic right shift back to achieve this. In the SIMD version, however, these steps are not necessary, since the results of the operations are already 16-bit values.

The former two cases have demonstrated the dependence of the speed-up on the iteration count. Another interesting figure is the development with dependence on rising unroll factors (after SIMD optimization). The example given in Figure 14 shows the progression for the dotproduct. The number of iterations for this graph has been chosen to $I = 128$. As Figure 13 illustrates, this is a number where the speed-up is already very close to its peak value.

In the values for the TriMedia, little difference is seen between the versions with or without dynamic checks. The strong rise in speed-up for the high unroll factors is due to the additional resource pressure created by the large loop body. Since the VLIW architecture is inherently parallel, this pressure is needed to completely saturate the CPU. The ARM's progression, however, shows an unexpected decline in performance for higher unroll factors. After close examination, the cause has been determined to be register shortage resulting in a considerable amount of spill code. Hence, the ARM greatly benefits from the removal of the dynamic check, since registers are freed and thereby more degrees of freedom are left to the register allocator. The TriMedia processor, with its 128 available registers, is not affected by this problem.

Loop unrolling is known to have a large impact on the code size. Hence, larger speed-ups come at the expense of an increased code size. Figure 15 illustrates

Fig. 14. Speed-up over unroll factor for dotproduct.

Fig. 15. Code size over unroll factor for dotproduct.

the code size increase for the dotproduct kernel ($I = 128$) due to unrolling for both the SIMD and non-SIMD version. The not unrolled, non-SIMD version is used as baseline. Due to the RISC architecture of the ARM, the code size increase caused by unrolling alone is more significant than for the TriMedia. However, the SIMD version for the ARM can compensate the code size effect of unrolling to a great extent. Firstly, SIMD directly reduces the number of instructions inside the loop. Secondly, the special dotproduct style SIMD instruction almost eliminates the overhead by scalar expansion. This kind of instruction is not available in the TriMedia. Additionally, SIMD reduces the

Fig. 16. Benchmark results.

number of instructions for the TriMedia as well, but not necessarily the number of VLIW words. Hence, the SIMD version shows a larger code size factor than the non-SIMD version. For high unroll factor, the parallel functional units of the TriMedia become saturated, which leads to a stronger rise of the code size. However, for modest unroll factors (2 or 4) the increase in code size is acceptable for both architectures.

Finally, Figure 16 summarizes the speed-up results for all benchmarks. The number of loop iterations I for the DSPStone kernels is fixed ($I = 128$) and for the more complex DSP routines as specified. For each benchmark, the unroll factor is 4. In the presence of dynamic alignment checks, the SIMD loop version including the alignment check overhead has been measured. A significant speed-up was obtained in most cases. The speed-up 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 speed-up of 7% up to 66% was observed. In certain cases, a super-linear speed-up for the ARM can be achieved (e.g., 2.2 for *fir*). This is related to the special multiply instructions of the ARM, which helps to reduce the overhead introduced by scalar expansion. On the other hand, for three benchmarks no speed-up could be obtained for the ARM due to the lack of a multiplication without accumulation.

As program speed-up is the primary objective in utilization of SIMD instructions, we omit detailed results and analysis of code size effects here. For the DSPStone kernels, we observed an average code size factor of 0.9 for the ARM and 1.1 for the TriMedia, as compared to benchmarks with unrolling enabled but without use of the SIMD optimizations. The code size of the complex kernels essentially remains the same for both architectures, since only a small portion of the code is replaced by SIMD instructions.

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. As pointed out in Eichenberger et al. [2004], Wu et al. [2005], and Nuzman et al. [2006], SIMD optimization is often hindered by limitations of the SIMD memory unit in combination with the memory access patterns in current applications. It is often necessary to reorder the subregisters, using special permute instructions before SIMD instructions can be applied at all. So far, these instructions are rarely supported by embedded processors. However, with the advances in semiconductor technology the SIMD data path width will increase in the future and thus, it becomes more likely that next-generation embedded processors will support those. Therefore, we plan to integrate support for permutation in the near future. Once the benchmarks become more complex, it becomes necessary to implement a cost model for the combination algorithm in order to find the optimal SIMD-set. Besides, there is currently no profitability analysis to decide whether it is worth at all to perform a SIMD optimization for a given loop. However, this could be easily implemented using CoSy's feedback mechanism. Performance estimates for both loop versions can be obtained before the final decision is made. Furthermore, 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. Future extensions like a points-to analysis are required to handle more complex access patterns for better SIMD recognition.

8. CONCLUSION

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 framework and by using an ADL-based retargeting technique. This concept has been demonstrated by integrating the SIMD engine within the C compiler generator of an existing industrial ASIP design framework and generating a SIMD-enabled compiler for two realistic embedded processors. While previous backend-oriented SIMD optimization techniques potentially lead to higher code quality, significant speed-up results for standard benchmarks were generally obtained with our framework. 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.

REFERENCES

- ASSOCIATED COMPUTER EXPERTS (ACE). The COSY compiler development system. <http://www.ace.nl>.
- ADVANCED RISC MACHINES LTD. The ARM11 processor. <http://www.arm.com>.
- ALLEN, J. R., KENNEDY, C. P., AND WARREN, J. 1983. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'83)*. ACM, New York.
- ALLEN, R. AND KENNEDY, K. 1987. Automatic translation of FORTRAN programs to vector form. *ACM Trans. Program. Lang. Syst.* 9, 4, 491–542.

- CHEONG, G. AND LAM, M. S. 1997. An optimizer for multimedia instruction sets. In *Proceedings of the 2nd SUIF Compiler Workshop*. Stanford University, CA.
- COWARE INC. Processor Designer. <http://www.coware.com>.
- EICHENBERGER, A. E., WU, P., AND O'BRIEN, K. 2004. Vectorization for SIMD architectures with alignment constraints. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI'04)*. ACM, New York, 82–93.
- FISHER, J. A. 1999. Customized instruction-sets for embedded processors. In *Proceedings of the Design Automation Conference (DAC'99)*. ACM, New York, 253–257.
- FRANCHETTI, F., KRAL, S., LORENZ, J., AND UEBERHUBER, C. W. 2005. Efficient utilization of SIMD extensions. *Proc. IEEE*. 93, 2, 409–425.
- FRASER, C. W., HANSON, D. R., AND PROEBSTING, T. A. 1992. Engineering a simple, efficient code-generator generator. *ACM Lett. Program. Lang. Syst.* 1, 3, 213–226.
- FRASER, C. W., HENRY, R. R., AND PROEBSTING, T. A. 1992. BURG: Fast optimal instruction selection and tree parsing. *SIGPLAN Not.* 27, 4, 68–76.
- GLÖCKLER, T., BITTERLICH, S., AND MEYR, H. 2000. ICORE: a low-power application specific instruction set processor for DVB-T acquisition and tracking. In *Proceedings of the 13th Annual IEEE International ASIC/SOC Conference*. IEEE, Los Alamitos, CA.
- GNU COMPILER COLLECTION. Auto-vectorization in GCC. <http://gcc.gnu.org/projects/tree-ssa/vectorization.html>.
- GRIES, M. AND KEUTZER, K. 2005. *Building ASIPs: The Mescal Methodology*. Springer-Verlag, Berlin, Germany.
- HOFFMANN, A., KOGEL, T., AND MEYR, H. 2001. A framework for fast hardware-software co-simulation. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE'01)*. IEEE, Los Alamitos, CA.
- HOFFMANN, A., MEYR, H., AND LEUPERS, R. 2002. *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, The Netherlands.
- HOHENAUER, M., SCHARWAECHTER, H., KARURI, K., WAHLEN, O., KOGEL, T., LEUPERS, R., ASCHEID, G., MEYR, H., BRAUN, G., AND VAN SOMEREN, H. 2004. A methodology and tool suite for C compiler generation from ADL processor models. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE'04)*. IEEE, Los Alamitos, CA, 1276.
- HOHENAUER, M., SCHUMACHER, C., LEUPERS, R., ASCHEID, G., MEYR, H., AND V. SOMEREN, H. 2006. Retargetable code optimization with SIMD instructions. In *Proceedings of the International Conference on Hardware/Software Co-design and System Synthesis (CODES+ISSS)*. ACM, New York, 148–153.
- INTEL CORPORATION. Intel C compiler. <http://www.intel.com>.
- KITAJIMA, A., ITOH, M., SATO, J., SHIOMI, A., TAKEUCHI, Y., AND IMAI, M. 2001. Effectiveness of the ASIP design system PEAS-III in design of pipelined processors. In *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC'01)*. ACM, New York, 649–654.
- KRALL, A. AND LELAIT, S. 2000. Compilation techniques for multimedia processors. *Int. J. Parall. Prog.* 28, 4, 347–361.
- KUDRIAVTSEV, A. AND KOGGE, P. 2005. Generation of permutations for SIMD processors. In *Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*. ACM, New York, 147–156.
- LARSEN, S. AND AMARASINGHE, S. 2000. Exploiting superword level parallelism with multimedia instruction-sets. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI'00)*. ACM, New York, 145–156.
- LARSEN, S. W. E. AND AMARASINGHE, S. P. 2002. Increasing and detecting memory address congruence. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*. IEEE, Los Alamitos, CA, 18–29.
- LEUPERS, R. 2000a. *Code Optimization Techniques for Embedded Processors*. Kluwer Academic Publishers, The Netherlands.
- LEUPERS, R. 2000b. Code selection for media processors with SIMD instructions. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'00)*. ACM, New York, 4–8.
- LEUPERS, R. AND MARWEDEL, P. 2001. *Retargetable Compiler Technology for Embedded Systems*. Kluwer Academic Publishers, The Netherlands.

- MISHRA, P., DUTT, N., AND NICOLAU, A. 2001. Functional abstraction driven design space exploration of heterogeneous programmable architectures. In *Proceedings of the International Symposium on System Synthesis (ISSS'01)*. ACM, New York, 256–261.
- MUCHNICK, S. S. 1997. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, San Francisco, CA.
- NUZMAN, D. AND HENDERSON, R. 2006. Multi-platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)*. IEEE, Los Alamitos, CA, 281–294.
- NUZMAN, D., ROSEN, I., AND ZAKS, A. 2006. Auto-vectorization of interleaved data for SIMD. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI'06)*. ACM, New York, 132–143.
- NXP Semiconductors. The TriMedia media processor. <http://www.nxp.com>.
- ORAIUGLU, A. AND VEIDENBAUM, A. 2003. Application specific microprocessors (Guest Editors' Introduction). *IEEE Des. Test Comput.* 20.
- POKAM, G., BIHAN, S., SIMONNET, J., AND BODIN, F. 2004. SWARP: a retargetable preprocessor for multimedia instructions. *Concurr. Comput. Pract. Exper.* 16, 2-3, 303–318.
- PRYANISHNIKOV, I., KRALL, A., AND HORSPOOL, N. 2003. Pointer alignment analysis for processors with SIMD instructions. In *Proceedings of the 5th Workshop on Media and Streaming Processors*. ACM, New York.
- REN, G., WU, P., AND PADUA, D. 2003. A preliminary study on the vectorization of multimedia applications for multimedia extensions. In *Proceedings of the 16th International Workshop of Languages and Compilers for Parallel Computing*. Springer, Berlin, Germany.
- REN, G., WU, P., AND PADUA, D. 2006. Optimizing data permutations for SIMD devices. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI'06)*. ACM, New York, 118–131.
- RIZZOLO, N. AND PADUA, D. 2005. HiLO: high level optimization of FFTs. In *Proceedings of the 18th International Conference on Languages and Compilers for High Performance Computing*. Springer, Berlin, Germany, 238–252.
- TENSILICA, INC. Xtensa C compiler. <http://www.tensilica.com>.
- WOLFE, M. J. 1995. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman, Boston, MA.
- WU, P., EICHENBERGER, A. E., AND WANG, A. 2005. Efficient SIMD code generation for runtime alignment and length conversion. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)*. IEEE, Los Alamitos, CA, 153–164.
- ZIVOJNOVIC, V., VELARDE, J., SCHLÄGER, C., AND MEYER, H. 1994. DSPStone—a DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT)*. IASTED, Calgary, Alberta.

Received January 2008; revised September 2008; accepted November 2008