Chapter 8
SIMD Optimization

As concluded in the previous chapter, retargetable compilers, as used in ASIP design environments, are still hampered by their limited code quality as compared to hand-written compilers or assembly code. Consequently, generated compilers must be *manually* refined to a highly optimizing compiler after successful architecture exploration. One way of overcoming this dilemma is to design *retargetable optimizations* for those architectural features that characterize a class of target processors.

This chapter focuses on target processors equipped with *SIMD instructions*. The term *SIMD* dates back to the year 1972 when Flynn [160] classified computers according to the number of data streams they operate on, and the number of instructions they execute (Table 8.1). The acronym *SIMD* stands for *single-instruction multiple data* and the class of computers referred to in the 1970s were vector computers that were able to execute the same operation on multiple vector elements at the same time.

Table 8.1 Flynn's classification

	Single instruction	Multiple instructions
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Today the meaning of the term has slightly changed. It usually denotes a special class of instructions found in many workstation and embedded processors that operate on short vectors of small data. As illustrated in Fig. 8.1, an 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. In future, 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 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

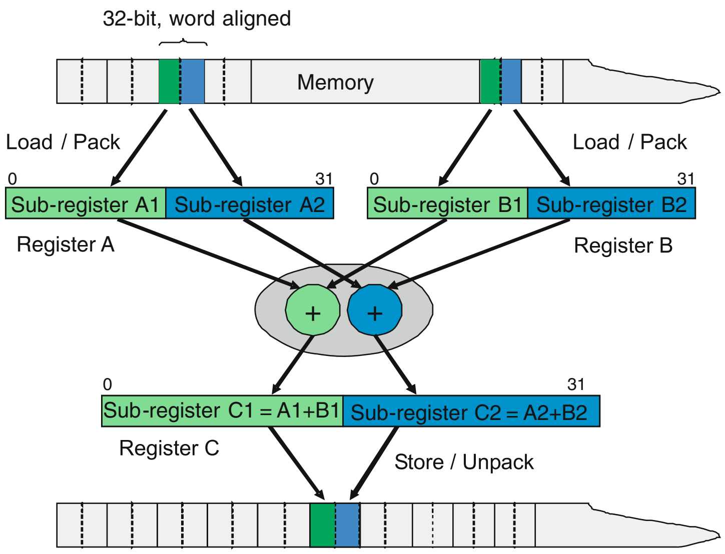

Fig. 8.1 Sample arithmetic SIMD instruction: two parallel ADDs on 16-bit subregisters of 32-bit data registers A, B, and C; the data is loaded/stored at once from/to an alignment boundary

improvements for computation-intensive multimedia workloads [145]. Therefore, many embedded processors for the next generation of high-end video and multimedia devices today feature SIMD instructions.

The SIMD concept is commonly found in general-purpose architectures such as Intel MMX/SSE1–5 [30], IBM/Motorola VMX/AltiVec [183], and AMD 3DNow. Later on, it was introduced in domain-specific processors (e.g., TI C6x, NXP TriMedia) and in recent custom ASIP designs (e.g., Tensilica Xtensa). Even some versions of the popular ARM- and MIPS-based architectures feature SIMD instructions. While several target-specific C compilers already exploit SIMD instructions, there is almost no support in ASIP compilers. Consequently, there is an increasing interest in retargetable compilers with SIMD support. For use in this domain, *retargetable SIMD optimizations* are required. This chapter presents a novel concept for retargetable code optimization for ASIPs with SIMD instructions, and this concept is proven by an implementation within the CoSy compiler that can be retargeted via the *Compiler Designer* GUI and an experimental evaluation for two real-life embedded processors.

The rest of this chapter is organized as follows. In Section 8.1 related work is discussed. The core of the SIMD framework is presented in Section 8.2 before the retargeting procedure is described in Section 8.3. Afterward, Section 8.4 provides the experiments for different embedded processors with SIMD support. Finally, Section 8.5 summarizes the contribution of this approach and points to some future avenues of work.

8.1 Related Work

Traditional code selection typically relies on tree parsing. As mentioned in Section 3.3.2, tree parsing is not suited to exploit SIMD instructions because they exceed the scope of a single DFT. Consequently, compilers require advanced techniques to exploit SIMD instructions.

Most of the current SIMD optimization techniques are based on the traditional loop-based vectorization [24, 95, 212, 213]. Others make use of instruction-packing techniques in conjunction with loop-unrolling to exploit data parallelism within a basic block [240] or a combination of traditional code selection [51] and integer linear programming [26, 221]. As investigated in [101], it is often difficult to apply SIMD optimization techniques since these architectures are largely nonuniform, featuring specialized functionalities, constrained memory accesses, and a limited set of data types. Moreover, complicated loop transformation techniques are needed [213] 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 that is neither adequately aligned nor contiguous. Besides, operations on disjoint vector elements are usually not supported. The detection of misaligned pointer references is presented in [117]. Certain misalignments can be solved either by loop transformations [95, 241] or by data permutation instructions. The efficient representation and generation of such instructions is investigated in [7, 72, 212] and the optimization thereof in [26, 102]. Consequently, only a successful interaction of several optimization modules will be able to leverage SIMD optimization for retargetable compilers.

So far, only advanced compilers (e.g., the Intel compiler [122], IBM XL compiler [7]) are capable of automatically utilizing SIMD instructions. Apart from being inherently nonretargetable, these compilers are mostly restricted to certain C language constructs. Other compilers use dedicated input languages for source-to-source transformations that are restricted to a certain application domain [83, 188]. The vast majority of the compilers, though, still provide only semi-automatic SIMD support via *compiler-known functions* (CKFs). Understandably, this assembly-like programming style is tedious and error prone. Moreover, this comes along with poor maintainability and portability of the code.

Among the ASIP design platforms mentioned in Chapter 4, so far only Tensilica's compiler includes SIMD support. However, its architectural scope is limited to the configurable Xtensa processor [215]. Considering retargetable compilers, recent versions of the *gcc* support SIMD for certain loop constructs [86]. The supported vectorization [71] features alignment and reduction; however, information regarding the concrete retargeting effort and the interaction of loop transformations are not available yet. Furthermore, *gcc* is mainly designed for general purpose processors. As a result, it does not adapt efficiently to specialized, irregular hardware architectures that are quite common in the embedded domain.

A retargetable preprocessor for multimedia instructions is presented in [100]. 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 that 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 how the loop transformations are adapted to a given SIMD architecture.

Summarized, several SIMD utilization concepts with different levels of complexity are available. However, they are mostly implemented in target-specific compilers. Consequently, adapting a SIMD optimization concept to a new target processor becomes a time-consuming and error-prone manual process. Therefore, this book presents an approach for the *efficient utilization of SIMD instructions* while achieving *compiler retargetability* at the same time. The presented SIMD optimization comprises a *loop-vectorizer* and an *unroll-and-pack*-based technique [166], 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 enables 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 ADL descriptions such as LISA. 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.

8.2 SIMD Framework

As mentioned above, a successful SIMD optimization is tightly coupled with several loop transformations in order to exhibit the necessary amount of parallelism and to convert loops into a proper form. Hence, the presented approach consists of several steps as depicted in Fig. 8.2.

First of all, a *loop-carried dependency* [178] and *alignment analysis* (Section 8.2.3) are performed. They provide the necessary annotation needed by the SIMD optimization framework. Afterward, a *SIMD analysis* (Section 8.2.4) searches for loops where SIMD optimization could be applied. For these loops, it determines the parameters for the different *loop transformations* (Sections 8.2.5, 8.2.6, 8.2.7, and 8.2.8). Finally, the SIMD optimization is performed, comprising a loop *vectorizer* (Section 8.2.7) or an unroll-and-pack-based *SIMDfyer* (Section 8.2.9) if vectorization fails. All modules are driven by the same, *retargetable* SIMD specification described in Section 8.3.

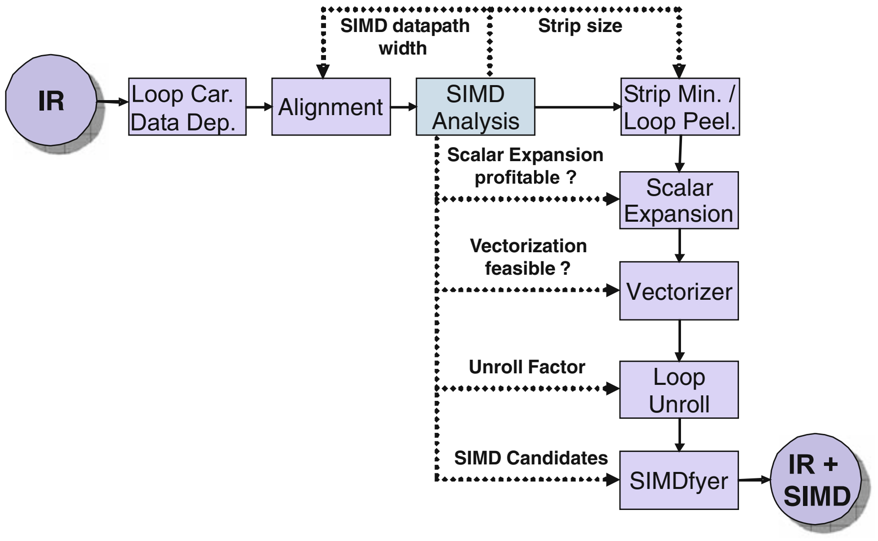

Fig. 8.2 SIMD code generation flow

8.2.1 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 such as addition, subtraction, multiplication, and so on. However, there are usually no dedicated IR elements for SIMD operations such as "two parallel additions." Extending the underlying IR format is not a practicable solution. All already existing compiler engines would have to be manually adapted in order to handle the new IR elements. Otherwise compiler engines might not exploit the full optimization potential or may even fail in the worst case. In either case, poor code quality would be the result. Therefore, generated SIMD instructions are internally represented in the form of CKFs. CKFs are transparent for other compiler modules and are later automatically replaced with assembly instructions in the backend. They are not visible to the compiler user at all. Furthermore, CKFs simplifies code generation to a certain extent, since it abstracts from low-level problems such as register allocation for SIMD subregisters in the backend. Moreover, all existing code generation and optimization engines of the underlying compiler framework can simply be reused. This includes the existing debug facilities of the compiler platform. In this way, the current IR state can be dumped into a human-readable, valid C-code file at any time during the SIMD generation process.

8.2.2 Terminology

Here, the terminology that facilitates the description of the optimization modules in the next sections is briefly introduced. As exemplified in Fig. 8.1, an 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, an 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 two subregisters. A single, primitive operation within the SIMD instruction (e.g., the 16-bit addition) is denoted as an *SIMD candidate*. It is basically a mapping rule covering this primitive operation. From these mapping rules, an *SIMD-candidate matcher* (Section 8.3.1) is generated (i.e., a regular tree pattern matcher) that is used for the identification of such SIMD candidates.

A set of SIMD candidates that can be combined into a SIMD instruction is denoted as an *SIMD-set*. For this purpose, a generated *SIMD-set constructor* is employed (Section 8.3.2). 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 $N = (n_1, \ldots, n_k)$ of SIMD candidates, where k denotes the number of subregisters, and 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 direct or indirect *dependencies* that would prevent parallelism. While this can be analyzed relatively simple for scalar variables, it becomes quite difficult in the case of array and pointer accesses.
3. Fulfill *alignment* constraints of the given target architecture. The data elements in memory must be packed in a single register in advance before the SIMD instruction can be executed. This involves wide load instructions, and hence possibly memory alignment constraints as well as reordering of subregister within a register using special *pack* and *permute* instructions. The same holds for storing the SIMD result again in memory.

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.

8.2.3 Alignment Analysis

One of the constraints when using SIMD instructions is the correct *alignment* of data in memory. In opposition to the original vector machines, which usually were equipped with superscalar memory units, the SIMD enabled general-purpose and embedded processors usually have a scalar memory unit. Parallel data loading is nevertheless possible as long as the data stream is stored contiguously in memory. 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 (Fig. 8.3).

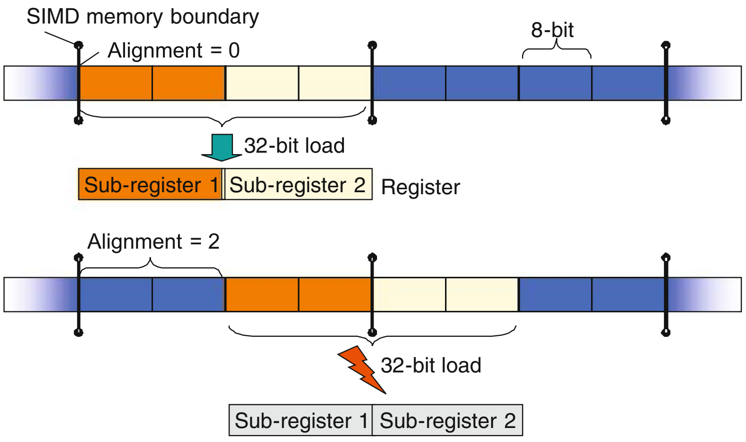

Fig. 8.3 SIMD alignment constraint

This is the optimal case, since the data is already available in the desired format. If however the data is locally disjoint, the required values have to be explicitly *packed* to the register before they are susceptible to SIMD optimizations. The two half-words would have to be loaded into two distinct registers, by doing two separate word-wide loads. In a second step, they can then be combined into a third register. Instead of doing a single load using a single register, at least two registers are used, and two separate loads, as well as an operation to merge the two half-words back into one register, have to be carried out. Even though many architectures offer support instructions such as permutations, multi-register shift operations, subword selection and general pack and unpack operations, the necessity of using them usually incurs a performance hit.

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 *run-time* [35, 117]. This procedure, also known as loop versioning, creates an optimized version of the code along with the original version. At runtime, a check as seen in Listing 8.1 is executed that selects the right version depending on the initial alignment.

```
if( (a is aligned) && (b is aligned) && (c is aligned) )
{
  for(i=0;i<N;i+=4) /* SIMD version */
  {
    c[i:i+4] = a[i:i+4] * b[i:i+4];
  }
} else {
  for(i=0;i<N;i++) /* Standard version */
  {
    c[i] = a[i] * b[i];
  }
}
```

Listing 8.1 Dynamic alignment check = 0

This version generates code that is always correct but obviously has the following two major drawbacks:

1. It increases the code size by more than a factor 2 for the loop.
2. It incurs the runtime overhead of the alignment check, which noticeably hurts performance for small iteration counts.

The strip-mining transformation (Section 8.2.5) needs to take the alignment into account, too. Therefore, an *interprocedural pointer-alignment analysis* [82] has been implemented for precise alignment information. It analyzes every memory access performed 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

$$alignment = p \bmod N \tag{8.1}$$

In order to account for the possibility that a pointer might have, during program execution, values with different alignments, the information is stored as a set E of possible values *modulo* N. If $M = \{0, \ldots, N-1\}$ is the set of all possible values of modulo N and $P = P(M)$ its power set, then $E \in P$.

In order to correctly annotate pointers in the whole program, it is necessary to track the value of pointer variables during their whole lifetime. A pointer generally is:

1. First initialized, usually by means of a memory management function such as `malloc` or by taking the address of an variable object.
2. Used, either directly or in address calculations such as `*(p+i)`, to access values in memory.
3. Manipulated or used in address calculations that are then stored to another pointer variable, which leads to a new initialization (e.g., `p = p + i`).

The analysis therefore needs the ability to determine the initial alignment of pointers. To do so, it needs specific knowledge about the possible initial sources of addresses. In the case of direct initialization by taking a memory address, this is possible using information about the variable object. In case of functions that take pointers as arguments, the initial values of the pointer parameters are not available inside the function. It is therefore sensible to use an interprocedural algorithm, which propagates the information across function boundaries. Next, this value must be tracked from its first *definition* to all its *uses*. This is a classic data-flow problem that can be solved using standard techniques as described in Chapter 3.

The third prerequisite to successfully uncover the alignment information in pointers is the ability to determine the offset for accesses that involve address calculations. In order to evaluate pointer arithmetic such as `*(p+i)`, a transfer function

$$f_g : P^n \longmapsto P \tag{8.2}$$

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 $f_{\text{binary}} : M \times M \mapsto M$. This leads to the following equations:

$$\begin{aligned} f_{\text{Add}}(a,b) &= (a+b) \bmod N = [(a \bmod N) + (b \bmod N)] \bmod N \\ f_{\text{Mul}}(a,b) &= (a \cdot b) \bmod N = [(a \bmod N) \cdot (b \bmod N)] \bmod N \end{aligned} \tag{8.3}$$

They are valid regardless of the value of N. If, however, $N = 2^m$ is a power of two, further functions can be deduced. This is due to the fact that a division by 2^m can be implemented by right shifting the binary representation of an integer value m times. The remainder of the division is then exactly formed by the m bits shifted out of the word. Therefore, it is in the last m bits of the original value. Using this knowledge, the operations AND, OR, XOR, and NOT can be handled without knowledge about the actual value as well.

8.2.4 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 pays off, if the SIMD optimization is later on enabled. Therefore, it is important to apply them only to the most promising loops for SIMD optimization. Hence, an *SIMD analysis* engine is implemented that runs in advance to identify those loops that contain SIMD candidates. For this purpose, the *SIMD-candidate matcher* is employed. Consequently, if the loop body does not contain any SIMD candidate, then it does not make sense to consider it further. Otherwise it 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.

8.2.5 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 an SIMD-set s for the vector operation. *Strip mining* is a loop transformation that divides the loop into strips, where each strip is no longer than the SIMD data path width [178]. Essentially, the loop is decomposed into two nested loops (Listing 8.2):

1. An outer loop (the *strip loop*) that steps between strips.
2. An inner loop (the *element loop*) that steps between single iterations within a loop.

```
// original loop
for (i = 0; i < 100; i++)
{
  A[i] = B[i] * C[i];
}
//outer strip loop
//strip_size = max. #sub-registers
for (is = 0; is<100; is += strip_size)
{ //inner element loop
  for (i=is; i<is+strip_size; i++)
  {
    A[i] =  B[i] * C[i];
  }
}
```

Listing 8.2 Strip mining example

The *SIMD analysis* calculates the iteration count of the element loop, called the *strip size*, based upon 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, the maximum strip size for the transformation is selected:

$$strip\_size = \max\left(\bigcup_{s\in S} k_s\right) \tag{8.4}$$

However, due to possible alignment constrains of the SIMD architecture, strip mining must ensure that each strip starts at an *alignment boundary*. Assuming that arrays are word aligned in memory, then the alignment boundaries are given by

$$alignment\_boundaries = \{i \mid i \bmod strip\_size = 0\} \tag{8.5}$$

where i is the loop counter. However, strip mining is performed in the iteration space. Thus, for array references like $[i+c]$ with c being a constant and $c \neq 0$ (Listing 8.3), the alignment boundary for each strip can differ from the real alignment in memory.

```
for (i = 0;i < 100;i++)
{
  A[i+1] = B[i+1] * C[i+1];
}
```

Listing 8.3 Offset = 1

Therefore, an *offset* can be set, if it remains constant within the loop, to readjust the alignment boundaries defined in the iteration space so that they correspond with the real alignment in memory. Consequently, the offset is always within the range $(-strip\_size, strip\_size)$. The alignment boundary is then given by

$$alignment\_boundaries = \{i \mid i + offset \bmod strip\_size = 0\} \tag{8.6}$$

The boundary information can be easily computed using the information from the alignment analysis. In case the loop does not directly start at an alignment boundary, *loop peeling* is applied to ensure the correct alignment of the data accesses. That means, those iterations causing the misalignment are "peeled off" the original loop and build a separate prolog loop. If the remaining iterations are not divisible by the strip size without remainder, then an extra epilog loop is created as well. Assuming an up-counting loop using a less-than condition, the loop boundaries for the prolog, strip loop, and epilog are defined as follows:

$$bFrom = iFrom + (-(iFrom + offset) \bmod strip\_size) \tag{8.7}$$

$$bTo = iTo - ((iTo + offset) \bmod strip\_size) \tag{8.8}$$

Listing 8.4 shows a generalized example. The initial and final value of the loop counter are given by *iFrom* and *iTo*, respectively, where *bFrom* defines the initial value of the strip loop and the upper bound of the prolog, and *bTo* the upper bound of the strip loop and the initial value of the epilog. Note that the modulo operation must produce a value in the range $[0, strip\_size)$. Furthermore, it must take care of overflows that might occur during the computation of the loop boundaries. Similar equations exist for different conditions and down-counting loops.

```
// peeled iterations (prologue)
for (i = iFrom; i < bFrom; i++)
{
  A[i+c] =  B[i+c] * C[i+c];
}
//strip mined loop
for (is = bFrom
     is < bTo;
     is += strip_size)
{
  for (i = is; i < is+strip_size; i+=1)
  {
    A[i+c] = B[i+c] * C[i+c];
  }
}
//epilogue loop
for (i = bTo; i < iTo; i++)
{
  A[i+c] =  B[i+c] * C[i+c];
}
```

Listing 8.4 Strip mining with offset != 0

8.2.6 Scalar Expansion

When scalars are assigned and later used in the loop, the dependency graph will include flow-dependence relations from the assignment to each use-and-loop-carried anti-dependencies from each use back to the assignment. These anti-dependence relations often cause problems in other transformations and could prevent parallelization of the loop (Listing 8.5). However, the anti-dependence relation can be broken by *scalar expansion* [178]. The basic idea is to allocate an array with one element for each iteration and replace each scalar reference in the loop with a reference to the array. This eliminates the anti-dependence relations. The computed value should be assigned to the original scalar after the loop (Listing 8.6). Scalars that are assigned conditionally can also be expanded given that

1. the scalar is assigned on every path through the loop body and
2. the scalar is not used before any assignment to the same scalar.

If a scalar is found that satisfies these constraints, it is replaced by an array access.

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. For instance, the memory usage can be reduced by strip mining the loop and only expanding the inner element loop.

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

Listing 8.5 Scalar causes anti-dependence

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

Listing 8.6 Replaced scalar with array access

8.2.7 The Vectorizer

A classical *vectorizer* parallelizes the whole loop at once provided that suitable SIMD instructions are available for *all* statements in the loop body 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 Fig. 8.4. 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 an 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).

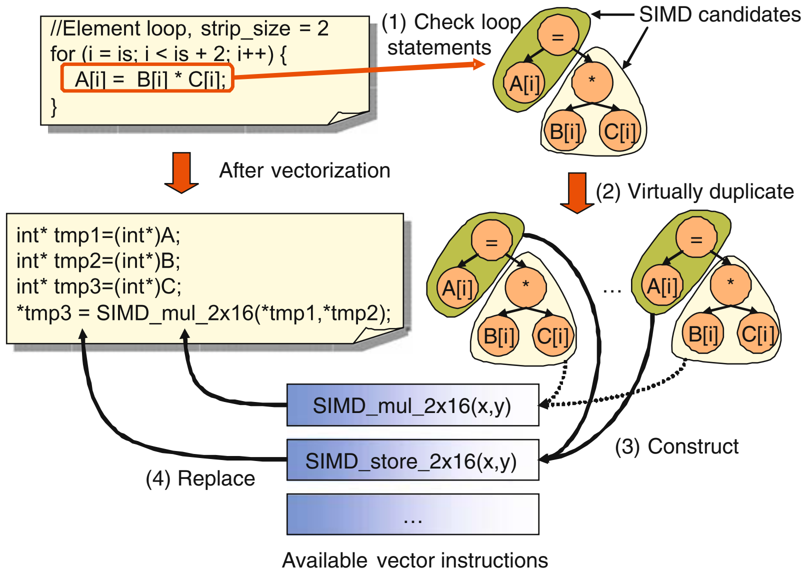

Fig. 8.4 Vectorization example

Of course, it might happen that not all loop statements can be directly parallelized, e.g., due to data dependencies. But still they may contain a certain degree of parallelism. Therefore, loops that could not be vectorized are further processed by the more powerful unroll-and-pack-based *SIMDfyer*.

8.2.8 Loop Unrolling

The *SIMDfyer* implements a technique similar to [240]. 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 an SIMD-set. Thus, since it is desired to fill all possible SIMD-sets S, the best unroll factor can be calculated as

$$unroll\_factor = \max\left(\bigcup_{s \in S} k_s\right) \tag{8.9}$$

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 unroller* to prepare them for the *SIMDfyer*.

8.2.9 The Unroll-and-Pack-Based SIMDfyer

For a given IR of an input C program, an iterative algorithm is used that combines SIMD candidates into SIMD-sets and replaces such sets by CKFs in the IR [55]. 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 where 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* [125] 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 Fig. 8.5.

State (1) shows the initial IR structure for a sample loop body (unrolled twice) that performs a multiplication of two vectors B and 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. First, the algorithm combines the left

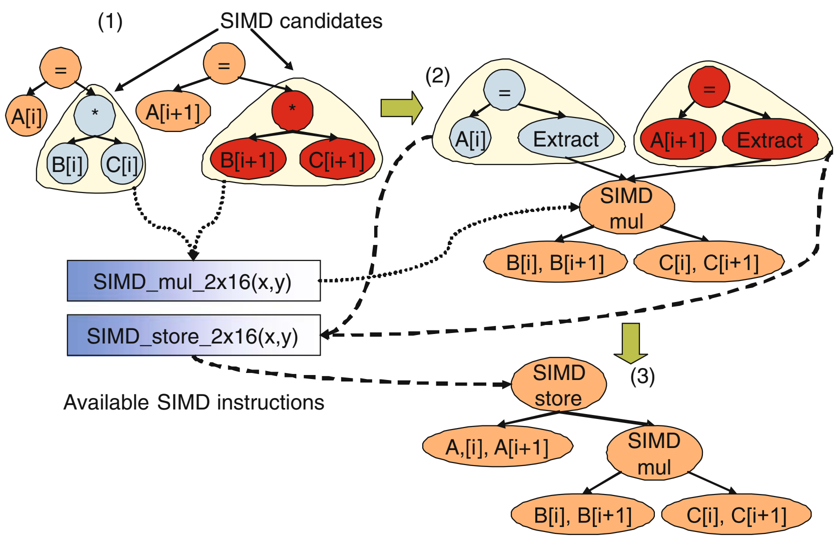

Fig. 8.5 IR states in different iterations

and the right operands (16-bit load operations) of the two "*" to 32-bit SIMD load operations. Afterward, the "*" operations themselves are combined to an 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 an SIMD-set. Note, all superfluous extracts are removed by *dead code elimination* in a later compilation phase. In the following iteration, the two 16-bit "=" operations form an SIMD-set on their own. Finally, the IR state (3) is reached and the algorithm terminates.

The presented approach employs an iterative, step-by-step approach in order to compose an SIMD instruction from a set of SIMD candidates. In this way, an exhaustive search within the given loop body is avoided. Therefore, it requires only low-degree polynomial complexity ($O(n^3)$), a worst case for n variable accesses in the IR. Practical experience shows that this relatively simple heuristic consumes only a few CPU seconds of compilation time while utilizing SIMD instructions very well for speeding up common DSP code benchmarks. Due to the possible necessity of inserting extra code for dynamic pointer-alignment checks before loop entry points and the corresponding code duplication, insertion of SIMD instructions may lead to an increase in code size.

8.2.10 Code Example

This section provides a more detailed example to illustrate the representation of SIMD instructions in the IR. Listing 8.7 shows the initial C source code after preprocessing (strip mining, scalar expansion, and loop unrolling). Assuming the availability of SIMD instructions for *addition* and *multiplication* operating on two 16-bit values, the *SIMD analysis* determines a strip size and an unroll factor of 2 for the loop transformations. Here, scalar expansion is performed on the element loop, which is then fully unrolled afterwards. It is further assumed that the target machine requires SIMD load operations to be *word aligned*.

```
void dotproduct(short *pa, short *pb, short *pc)
{
  short sum;
  short S[2];
  sum = 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];
}
```

Listing 8.7 Initial code

In the first iteration, the two multiplications are detected as SIMD candidates 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 can be easily met by directly using the temporaries the input operands are extracted from. Obviously, this makes the extract operations from the previous iteration superfluous. The resulting code is depicted in Listing 8.9 while Listing 8.10 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.

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

Listing 8.8 First iteration

```
void dotproduct(short *pa, short *pb, short *pc)
{
  short sum;
  short S[2];
  int tmp1,tmp2,tmp3;
  short res0,res1,res2,res3,res4,res5;
  sum = 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);
    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];
}
```

Listing 8.9 Second iteration

```
void dotproduct(short *pa, short *pb, short *pc)
{
  short sum;
  short S[2];
  sum =  S[0] = S[1] = 0;

  if( ((pa|pb|pc) & 3) == 0 )
  {
    for (int 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(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];
}
```

Listing 8.10 Final code

8.3 Retargeting the SIMD Framework

To retarget the SIMD framework, basically two pieces of information are required: first, a description of IR tree patterns that represent a SIMD candidate. This is used to generate the *SIMD-candidate matcher*. Second, the *SIMD-set construction*, the specification of how SIMD candidates can be composed to a valid SIMD-set.

8.3.1 SIMD-Candidate Matcher

The identification of SIMD candidates can be implemented using the tree-covering-based code selection [244]. SIMD candidates can be easily described by regular *mapping 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 such as Burg [52]. In this approach, the regular CoSy tree-pattern-matcher generator is utilized to create a dedicated SIMD-candidate matcher from *SIMD-candidate rules*, which are part of the regular code selector description.[1] 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 (Fig. 8.6).

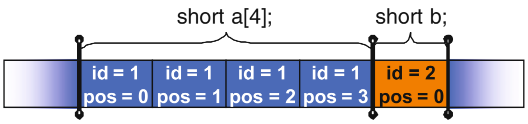

Fig. 8.6 Pos/id for array/scalar variable

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 an SIMD operation is directly consumed by another one. The initial values for these fields are already determined by the prior data-flow/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 Listings 8.11 and 8.12.

```
\\Syntax is name:type
RULE [load] o:mirContent(src:reg_nt)
            -> dst:simd_nt;
CONDITION {
     IS_INT16(o)
}
EMIT {
     dst.pos = get_pos(o);
     dst.id  = get_id(o);
}
```

Listing 8.11 SIMD-candidate rule `load`

The 16-bit `load` rule initializes the SIMD nonterminal's `pos` and `id` fields with the values determined by data-flow/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.

[1] This is not a contradiction to the limitations of tree pattern matching mentioned in Section 8.1. The matcher is only employed to identify those IR operations that might be composed to a full SIMD operation, the complete SIMD match cannot be found directly.

```
RULE   [add] o:mirPlus(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);
}
```

Listing 8.12 SIMD-candidate rule `add`

Additionally, rules to extract a subregister from a full register must be created as well. Those are used to match the extract operations (see Section 8.2.10) inserted in previous iterations of the algorithm. In this way, they become SIMD candidates in the current iteration. All extract rules produce an SIMD nonterminal that 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 (Fig. 8.7).

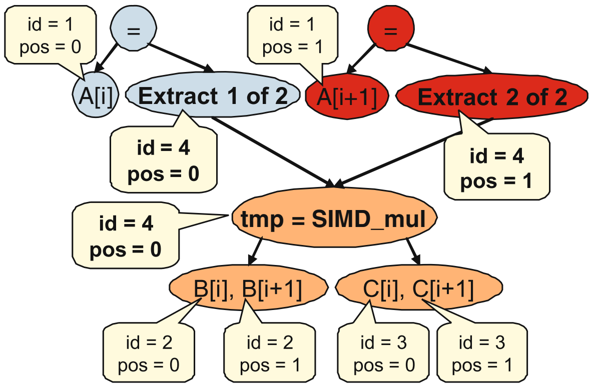

Fig. 8.7 Pos/id for extract operation

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.

8.3.2 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, i.e., a node n_i corresponds to an SIMD-candidate rule name. The examples in Listings 8.13 and 8.14 specify a twofold 16-bit load and add SIMD instruction, using the SIMD-candidate rules from Listings 8.11 and 8.12.

```
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));
}
```

Listing 8.13 SIMD rule twofold 16-bit load

```
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));
}
```

Listing 8.14 SIMD rule dual 16-bit add

Given the set of all identified SIMD candidates $C = \{c_1, c_2, \dots\}$, the set of all 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 Pos(c) denote the `pos` value of the result SIMD nonterminal produced by SIMD-candidate rule c and Id(c) the `id`, respectively. Then the set of valid SIMD-sets S is given by:

$$S = \{(c_1, \dots, c_k) \mid (c_1, \dots, c_k) \in R \wedge \mathrm{Id}(c_i) = \mathrm{Id}(c_j) \wedge \mathrm{Pos}(c_{l+1}) = \mathrm{Pos}(c_l) + 1, \\ \forall i, j \in (1, \dots, k), l \in (1, \dots, k-1)\} \tag{8.10}$$

In other words, the SIMD candidates of a valid SIMD-set must have the same `id` as well as an increasing `pos` value assigned.

Consider the example shown in Listing 8.15. In the first iteration, the `load` rule covers the array accesses, initializes the `id` with an unique number and the `pos` field with the position relative to SIMD load memory boundary. Note that accesses to the same array get always the same `id` assigned. Only the `pos` field varies. It is assumed 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 also create new temporaries which get the same `id` as the temporary the sub-register is extracted from assigned and the `pos` field is set to the extracted subregister number, respectively.

```
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>
// ...
```

Listing 8.15 `pos/id` in the first iteration

Thus, in the next iteration (Listing 8.16), the first and second operands of the first two additions share the same `id`s. Consequently, the same `id` is generated for both results of the additions. Now they can be combined to an `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, the conditions on the `pos` fields can be used to model unaligned SIMD memory operations.

```
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>)
// ...
```

Listing 8.16 `pos/id` in the second iteration

In order to complete the retargetable compilation flow, the CKF calls in the resulting intermediate code must be replaced by valid assembly instructions for the target processor. In this framework, the `COMPOSITION` for an SIMD rule specifies the CKF call that is internally generated for an identified SIMD-set. It consists of an 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 nonterminals that contain the first and second operands 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 (Listing 8.17) and becomes part of the regular backend code selector.

```
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));
}
```

Listing 8.17 Internally generated CKF rule for SIMD_add_2x16

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 this approach.

As mentioned in Chapter 6, the *Compiler Designer* tool comprises techniques to generate mapping rules automatically from the LISA model. Since the SIMD configuration is quite similar to a regular code selector description, the *Compiler Designer* has been extended in order to specify and generate rules for SIMD instructions, too. More specifically, the user creates the SIMD candidate rules using the mapping dialog. In the next step, the user can select those SIMD candidates which build an SIMD-set and assign a proper assembly instruction. From this specification, an SIMD-enabled code selector description for the CoSy compiler platform is finally generated.

8.4 Experimental Results

For the evaluation, two different aspects have to be taken into account. First of all, a precise alignment analysis is a prerequisite for the SIMD optimizations to achieve good results. Therefore, this chapter first evaluates the efficiency of the alignment analysis before the benchmark results for the SIMD optimization itself are presented.

8.4.1 Alignment Analysis

The alignment is classified in one of the three classes:

Unknown: The annotation is $E_i = \emptyset$, the empty set. No information about the alignment could be gathered during the analysis.

Known: The set contains a single value. Thus, the alignment is exactly known.

Ambiguous: The set contains several values. With regard to the annotation precision, this is equivalent to a known value. It means that the alignment will actually change during the runtime of the program.

The metrics used to measure the accuracy is the ratio of annotated to total nodes:

$$r = \frac{\text{number of known nodes} + \text{number of ambiguous nodes}}{\text{number of total nodes}} \tag{8.11}$$

The nominator expression is the sum of both, the exactly known pointers and the ambiguous pointers. This is reasonable since an expression that contains several entries in its set can definitely take on several modulo values, depending on the program's input data. The applications chosen to benchmark the results are taken from the domain of typical DSP and embedded algorithms. They present different degrees of complexity to the compiler, which are as follows.

ADPCM: This is a floating-point implementation of an adaptive differential pulse-code modulation encoder. It is a self-contained program with a `main()` procedure calling a few worker procedures. Data accesses are performed through pointers that are initialized to the addresses of global objects and then manipulated by address arithmetics throughout the program. All the functions were contained in a single compilation unit.

FFT: The FFT works on a 16-bit fixed-point representation but is otherwise similar to the ADPCM described above. Several functions are combined in a single compilation unit. In contrast to the ADPCM, however, the data are passed by means of pointer arguments to function calls.

libmad: This is an open source 32-bit fixed-point implementation [265] of the MPEG-1 audio Layer 1–3 standards [185]. The primary goal of the project is to provide a high-performance mp3 library written in a portable C style. It consists of several modules that are compiled separately and exchange data by means of pointer arguments.

gsm: The implementation used is freely available on the Internet [129]. It is a floating-point implementation of the standard and similar in structure to `libmad`.

AAC: This is the AAC audio codec's reference implementation of the 3GPP consortium. It is written in ANSI-C, spread across a large number of modules, and makes heavy use of complex language elements such as arrays of pointers or nested `struct`s.

H.264: This is another complex library in the same style as the AAC decoder.

The benchmarks above have been chosen to measure the *annotation rate*. As shown in the next section, typical SIMD benchmarks for embedded processors supports only a very basic set of SIMD operations, which must

- be completely regular;
- work on short data types of 8- or 16-bit size;
- work on fixed-point data types.

The test cases here do not comply with these requirements. They operate on floating-point or 32-bit fixed-point representations. Creating fixed-point versions of complex algorithms, however, requires a high engineering effort. For that reason, such versions are usually not publicly available. Nevertheless, the set of test cases chosen does contain a typical set of pointer accesses to floating-point data types and can therefore be used to evaluate how efficiently the analysis can propagate values around the program. The detailed results are given in Table 8.2. In addition to the name and the rate, the number of compilation units (CUs) the program consists of,

Table 8.2 Annotation rate

Name	CUs	Lines	Rate%	Total	Known	Ambiguous	Unknown
adpcm	1	493	100	39	39	0	0
FFT	1	457	93	31	27	2	2
libmad	12	11791	58	3362	1738	211	1413
GSM	14	4014	55	1620	869	28	723
AAC	38	6767	20	5100	811	236	4053
H.264	30	31099	19	13188	2428	90	10670

the total number of lines in the source code, the total number of pointers in the program, and the numbers for known, ambiguous, and unknown annotations are given. It is obvious that the programs tested can be divided into three classes with respect to their predisposition for alignment analysis. The straightforward implementations of the FFT and the ADPCM coder give very good results. These are complete programs, which are available in a single compilation unit, with a single entry point, the `main()` function. The code is written using direct pointers to the data involved. Those pointers are then modified by address arithmetics during the program's execution.

The *GSM* implementation and *libmad* are similar in coding style to the previous class. They make moderate use of structs and usually pass pointers to the memory operated upon. The main difference to the first class is that they are formed by several compilation units. For modules that are largely self-contained and that have a well-defined interface to the outside world, the annotation rate is usually better than for the modules that handle file access. The core-encoder routine for the GSM codec achieved an annotation rate of 70 and 82% of the pointers in the Layer III decoding module of libmad could successfully be annotated. This is due to the fact that the developers of these libraries made liberal use of the `static` storage classifier for functions that enabled the creation of a call graph with less edges. However, a noticeable uncertainty with regards to the interprocedural flow remains, which clearly shows in the average annotation rate of about 55% in these cases.

The programs in the third class, which is hardly analyzable, are reference implementations of recent audio and video codecs. They have been written for readability by humans and correct, yet not necessarily fast execution. This leads to skimpy use of the `static` classifier, nested structures to emulate class hierarchies, and multi-dimensional arrays of structures. An excerpt from the core-decoding module of the aac decoder is shown in Listing 8.18. This coding style makes it very

```
AACDECODER CAacDecOpen(...)
{
  struct AAC_DECODER_INSTANCE *self;
  ...
  AacDecInstance.pAacDecStaticChannelInfo[ch]->pLongWindow[0] =
  OnlyLongWindowSine;
  self->pAacDecChannelInfo[ch]->pCodeBook =
  pAacDecDynamicDataInit[ch]->aCodeBook;
        ...
}
```

Listing 8.18 Source excerpt from the core aac-decoding module

difficult to do the data-flow analysis, upon which the alignment analysis is built. In order to successfully annotate programs like these, not only the values assigned to objects, but also values in memory have to be tracked.

8.4.2 SIMD Optimizations

For experimental evaluation, SIMD-enabled C compilers have been created for the NXP TriMedia processor [190] and the ARM11 [41]. The TriMedia compiler has been designed using the *Compiler Designer* tool whereas the ARM11 compiler is a hand-crafted CoSy compiler. In contrast to, e.g., 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, mostly benchmarks from the DSPStone benchmark suite [269] have been selected and several additional kernels have been implemented, similar to those used in [72, 86, 117]. Furthermore, additional results for the following more complex DSP algorithms are provided:

quantize matrix quantization with rounding
compress discrete cosine transformation to compress a 128 $\times$ 128 pixel image by a factor of 4:1, block size of 8 $\times$ 8
idct 8 $\times$ 8 IEEE-1180 compliant inverse discrete cosine transformation
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 one-by-M tap filter

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.

Regarding the SIMD architecture, the TriMedia is a five-slot VLIW DSP with 128 general-purpose registers and 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 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 speedup 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 leads to deviations from the theoretical speedup factor k in case of k subregisters. The memory is organized in 32-bit words, hence word alignment is required for SIMD memory accesses.

In contrast, the ARM architecture is built around a central, scalar RISC core. It has a register file that 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 bits 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 dot-product support operation, which 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 dot-product support operation cannot be optimized.

Loop unrolling alone already has a large impact on the overall performance. Hence, the speedup is measured by using the following equation:

$$Speedup = \frac{cycles_{Unroll}}{cycles_{Vector+SIMDfyer}} \tag{8.12}$$

$Cycles_{Unroll}$ denotes the number of cycles the test kernel needed when compiled with unrolling turned on, but the SIMD engines (i.e., *Vectorizer* and *SIMD-fyer*) 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 speedup is only due to the SIMD instructions. All other compiler parameters have always been identical.

The results are quantified first for one simple, particular benchmark, that is, a *dot product*, 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];
```

Listing 8.19 Dot product

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, the impact of the alignment analysis and the overhead introduced by scalar expansion is investigated. Figure 8.8 shows the speedup over the number of loop iterations I with and without 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 offs. Beyond that, the speedup is largely independent of I. For high iteration counts, the speedup is asymptotically 2, which corresponds to the theoretical speedup in this case. Obviously, 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 speedup 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 precision afterward. The ARM compiler, however, 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, though, these steps are not necessary since the results of the operations are already 16-bit values.

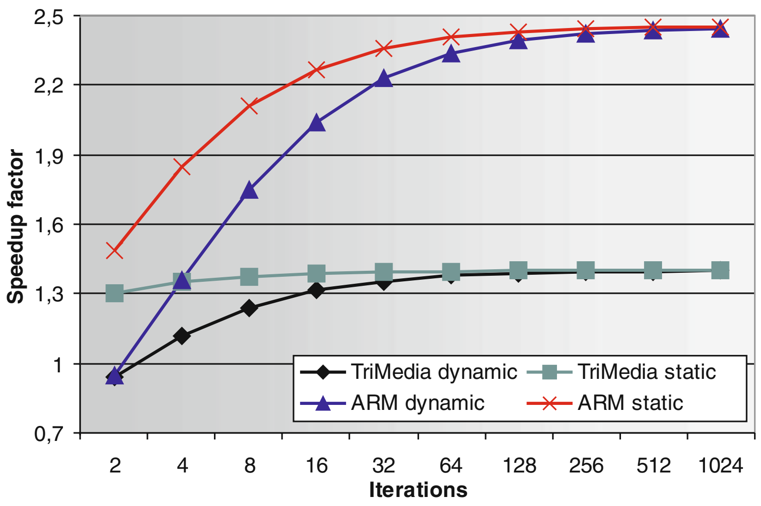

Fig. 8.8 Speedup factor over loop iterations for dot product

The former two cases have demonstrated the dependence of the speedup on the iteration count. Another interesting figure is the development with dependence on rising unroll factors (after SIMD optimization). The example given in Fig. 8.9 shows the progression for the dot product. The number of iterations for this graph has been chosen to $N = 128$. As apparent from Fig. 8.8, this is a number where the speedup is already very close to its peak value.

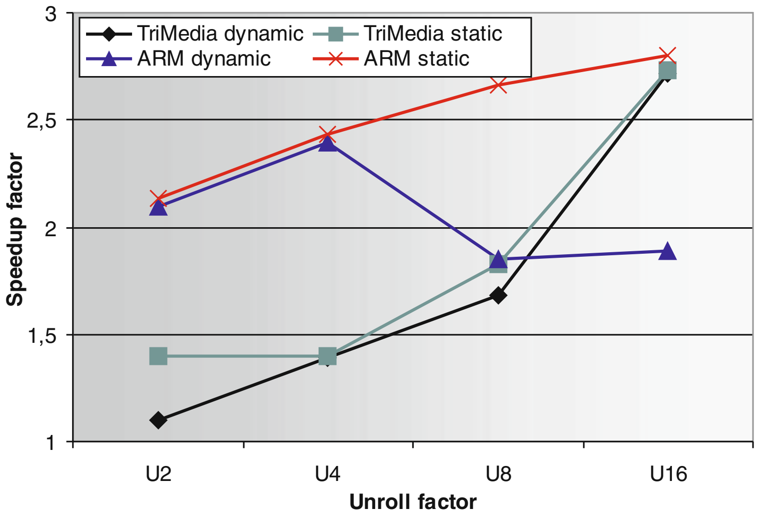

Fig. 8.9 Speedup factor over unroll factor for dot product

In the values for the TriMedia, little difference is seen between the versions with or without dynamic checks. The strong rise in speedup 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. Obviously, 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 speedups come at the expense of an increased code size. Figure 8.10 illustrates the code-size increase for the dot-product 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. First, SIMD directly reduces the number of instructions inside the loop.

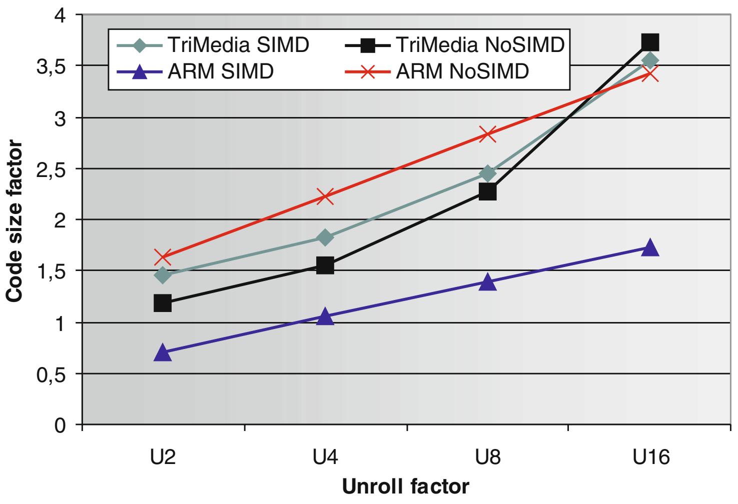

Fig. 8.10 Code size over unroll factor for dot product

Second, the special dot-product-style SIMD instruction almost eliminates the overhead by scalar expansion. This kind of instruction is not available in the TriMedia. Additionally, SIMD reduces the 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, Fig. 8.11 summarizes the speedup 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.

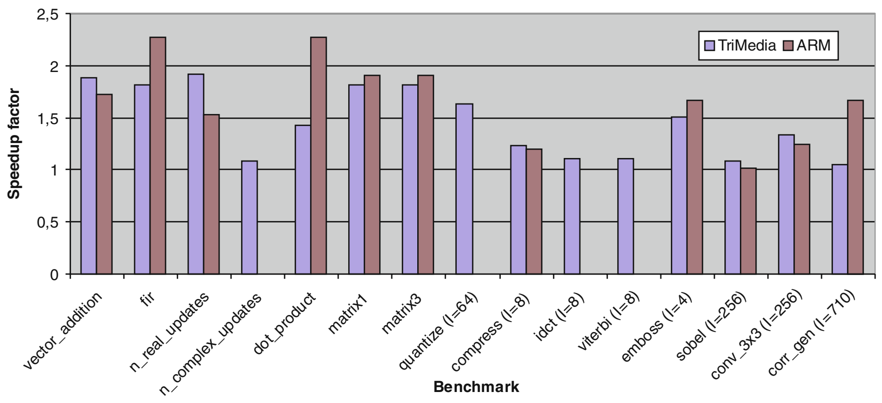

Fig. 8.11 Benchmark results

In the 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. In certain cases, a super-linear speedup for the ARM can be achieved (e.g., 2.2 for *fir*). This is related to the special multiply instructions of the ARM that helps to reduce the overhead introduced by scalar expansion. On the other hand, for three benchmarks, no speedup could be obtained for the ARM due to the lack of a multiplication without accumulation.

Regarding the code size, for the DSPStone kernels, an average code-size factor of 0.9 for the ARM and 1.1 for the TriMedia can be observed, 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.

8.5 Conclusions

Almost all previous approaches to SIMD optimization are tailored to a specific target architecture. This book presents a *retargetable* optimization framework for the class of processors with SIMD support. The underlying concepts are proven by integrating the SIMD framework into the CoSy platform that can be retargeted via the *Compiler Designer* GUI. In this way, SIMD-enabled compiler for two realistic embedded processors were generated. The required retargeting effort is quite limited for both compilers.

This results in a seamless and retargetable path from a single LISA model to a SIMD-enabled C compiler. While previous backend-oriented SIMD optimization techniques potentially led to higher code quality, significant speedup results for standard benchmarks were generally obtained with this framework. Hence, the presented approach provides a good and practical compromise between *code efficiency* and *compiler flexibility*.

The 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 [7, 72, 212], 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 *support for permutation* seems to be a promising extension for the future.