Pointer Alignment Analysis for Processors with SIMD Instructions

Ivan Pryanishnikov and Andreas Krall
Technische Universität Wien
Institut für Computersprachen
Argentinierstrasse 8
1040 Wien, Austria
{prianich,andi}@complang.tuwien.ac.at

Nigel Horspool
University of Victoria
Department of Computer Science
Victoria, BC
Canada V8W 3P6
nigelh@uvic.ca

ABSTRACT

Embedded processors for media applications usually have SIMD instructions. SIMD instructions provide a form of vectorization where a large machine word is viewed as a vector of subwords and the same operation is performed on all subwords in parallel. Systematic usage of SIMD instructions can significantly improve program performance. Usually each memory access must be aligned with the instruction's data access size. With C becoming the dominant language for programming embedded devices, there is a clear need for C compilers which optimize the use of SIMD instructions. An important problem in designing such compilers is the question of determining whether a C pointer is aligned, i.e., whether it refers to the beginning of a machine word.

In this paper, we describe a method which determines the alignment of pointers at compile time. The alignment information is used to reduce the number of dynamic alignment checks and the overhead incurred by them. Our method uses an interprocedural analysis which analyzes pointer values propagated through function calls. The effectiveness of our method is substantiated by experimental results which show that the alignments of about 50% of the pointers can typically be statically determined.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – compilers; optimization;

General Terms

Algorithms, Measurement, Theory

Keywords

DSP, SIMD, vectorization, alignment analysis, static pointer analysis

1. INTRODUCTION

Digital Signal Processors (DSPs) are highly specialized embedded microprocessors designed for real time processing of digitized analog signals. Such processors have emerged in recent years to handle audio, video, graphics, and communication tasks. These processors are known as *media processors*. They operate on very specific media data, and are designed for a relatively narrow set of applications. Therefore, they are usually equipped with unique instruction sets, and their architectural features are designed to provide high performance.

A typical computing scenario involves executing the same or almost the same sequence of operations on different elements of a large data set, e.g. an array. In this situation, a traditional computing model, where a *single* instruction (such as load, store, or integer addition) operates on a *single* data element, is not very efficient. The Single Instruction Multiple Data (SIMD) model is designed to improve program execution performance by performing the same type of computation on different data items in parallel [10, 7]. For example, a typical SIMD instruction operates on two adjacent bytes simultaneously.

To use a block of data in a load or store instruction, the elements of the block must be adjacent in memory. Typically, a block is a 32 or 64 bit machine word, and the elements are single bytes or short integers. Since a block load or store is carried out as a one word instruction, the SIMD model usually requires that a block of data be naturally aligned, i.e., be mapped to an address which is zero modulo the block size. In other words, a block has to coincide with a natural machine word.

In this paper, we are primarily interested in LOAD/STORE architectures, where LOAD and STORE are the only instructions which interact with the main memory. Arithmetic and logical SIMD instructions cannot reference memory directly; instead they must access registers and constants. This leaves two principal directions for optimizing SIMD code: (i) we may use arithmetic and logical SIMD instructions to parallelize independent computations, and (ii) we may use LOAD and STORE operations to access data vectors in memory efficiently.

As many hardware features have to be taken care of, em-

```
short void f(short a[]) {
    int *b = (int*)a;
    ...
}
```

(a) b is aligned

```
short void f(short a[]) {
    int *b = (int*)(a+1);
    ...
}
```

(b) b is unaligned

Figure 1: Aligned versus unaligned pointer access

bedded software for media processors has traditionally been programmed in assembler. Recently, however, the C programming language, using highly optimizing compilers, has also been used for programming media processors. The C language allows use of pointers to memory locations. We say that a pointer is aligned if its value is zero modulo the access size. In general, C pointer arithmetic may result in unaligned pointers, and thus lead to unaligned data buffers. Unaligned pointers are often the result of casts between different types of pointers, as seen in Figure 1. (Untyped pointers, declared as `void*` are quite common in C code.)

Depending on the computer architecture, the use of an unaligned data operand for a SIMD instruction has unfortunate effects at run-time. The effects may be incorrect results, an error interrupt, or a severe degradation in performance. In all cases, the compiler should avoid generating code that accesses unaligned data with SIMD LOAD and STORE instructions. Therefore the compiler should either statically verify or generate code that dynamically verifies whether the pointers are aligned before they are used as operands of SIMD instructions.

Although refined dynamic methods have been presented [9], dynamic checking of address alignment usually increases program size and requires run-time checks. Compile-time checking of alignments using a static analysis method should always be the preferred approach, with the compiler generating code appropriate for the alignments that have been determined. Dynamic checks should be used only for cases where the static techniques have failed to yield useful information. The goal is to have a powerful static analysis method where dynamic checks are rarely needed.

In this paper, we present an interprocedural static pointer alignment analysis for the SIMD model based on a C compiler prototype. Alignment analysis considerably reduces the number of dynamic checks, and helps shrink both the code size and the number of executed cycles. In addition, our analysis enables further code optimizing transformations which adjust unaligned buffers to fit the alignment requirements.

The paper is organized as follows: The compiler framework and basic SIMD optimization are given in Section 2. Detailed descriptions of our interprocedural alignment analysis is provided in Section 3. Experimental results are presented in Section 4. Section 5 provides an overview of related work. Section 6 concludes.

2. COMPILER OPTIMIZATIONS FOR SIMD INSTRUCTIONS

The C compiler used for the experiments described in this paper is based on the OCE compiler framework from Atair [1]. This framework contains a front end which includes many classical machine-independent compiler optimizations. The code generator of the compiler presented in [6] is designed to handle irregular architecture features, such as restricted register sets, hardware loops, and special address modes.

The compiler has a special optimization pass to create SIMD instructions. The implementation is based on the algorithm presented in [7], and described in detail also in [13]. The SIMD optimization pass operates on individual C functions, using the machine-independent intermediate representation generated by the front end of the compiler. The algorithm combines several structurally equivalent expressions with 8 or 16 bit operands into one expression with 32 bit operands and corresponding SIMD-operators. The same algorithm works for larger word lengths as well.

The most promising code fragments for use of SIMD instructions are loops which process data arrays. When these loops are unrolled a few times, it is often possible to use SIMD instructions. More formally, the innermost loops of a function are unrolled K times, where K depends on the data types. For example, if the array contains 16 bit elements, and the operands are 32 bits, $K = 2$. A loop where the number of iterations is not a multiple of K may be made amenable to SIMD optimization by preceding the unrolled loop with a preloop. Consider for example a program of the form:

```
for (i = 0; i<N; i++)
    Π(i);
```

where Π stands for a piece of code using variable `i` (assuming that `i` is not modified by Π), and `N` is the number of iterations. The preloop for this program is

```
for (i = 0; i<N%K; i++)
    Π(i);
```

while the unrolled loop assumes the form:

```
for (i = N%K+1; i<N; i += K) {
    Π(i);
    Π(i+1);
    ...
    Π(i+K-1);
}
```

Depending on the alignment information about pointers in Π, we may need a postloop instead of a preloop. The con-

```
short *a, *b, *c;
for (i = 0; i<N; i += 2) {
    a[i]   = b[i]+c[i];
    a[i+1] = b[i+1]+c[i+1];
}
```

(a) original program

```
short *a, *b, *c;
if (a&3 || b&3 || c&3) {
    /* at least one pointer is unaligned, so
       execute more general version of loop */
    for (i = 0; i<N; i += 2) {
        a[i]   = b[i]+c[i];
        a[i+1] = b[i+1]+c[i+1];
    }
} else  {
    /* all pointers are aligned, so process
       two array elements at a time */
    for (i = 0; i<N; i += 2) {
        a[i:i+1] = b[i:i+1]+c[i:i+1];
    }
}
```

(b) program with dynamic checks

Figure 2: Dynamic alignment checking of pointers.

struction is completely analogous. These loop transformations for C programs are implemented at the intermediate representation level.

The unrolled loop is inspected to identify candidates for SIMD instructions. The candidates must be executed in each loop iteration. Several candidate operations can be combined into one SIMD instruction only if there are no true dependencies or output dependencies between them. Once this has been verified, SIMD statements are generated, replacing the corresponding candidates.

If a statement in the original code contains a scalar expression, the SIMD optimization computes a suitable K-dimensional non-scalar expression to replace it. The elements of an expanded scalar equal the value of the scalar from which the expression was expanded. For example, for the statement

```
a[i] = b[i] + c;
```

with scalar `c`, and with array elements of type `short` the non-scalar expression will be $\mathtt{c}'[\,] = \{\mathtt{c}, \mathtt{c}\}$.

Another common operation is computing the sum of the elements of an array. This operation is not directly vectorizable, and a K element auxiliary array is required. Its elements are used to accumulate partial sums in parallel, from which the final sum can then be computed by another addition (accumulator splitting).

As mentioned earlier, to use a block of data for SIMD load and store instructions it must be verified if the block is aligned, i.e., if it starts at an address which is zero modulo the block size. Consider the example in Figure 2. The two statements in the loop body of Figure 2a can be used for SIMD load and store instructions directly, if the addresses of `a[0]`, `b[0]` and `c[0]` are aligned. If this information is not available at compilation time, a dynamic check must be inserted in the program, cf. Figure 2b. At present, our compiler is equipped with an alignment verification mechanism that takes care of the alignment requirements. The verification mechanism integrates both a static alignment analysis and dynamic check. Our static alignment analysis is presented in the next section.

3. ALIGNMENT ANALYSIS

Using a retargetable C compiler, we have to deal with two kinds of pointers: (1) pointers to memory locations used as data values in the original C program, and (2) auxiliary pointers created by various optimizations at the intermediate representation level. In this section we present our machine-independent alignment analysis for both user-defined and auxiliary pointers. The analysis determines alignment information by considering the low order (least significant) bits of pointer values. First, we present an algorithm for collecting alignment information for one procedure. Then, we describe our interprocedural context-sensitive alignment analysis.

3.1 Intraprocedural Alignment Analysis

The goal of the intraprocedural alignment analysis is to annotate definitions of all pointers within a single procedure with information about the possible values of the least significant bits (may analysis). For example, the three least significant bits determine the values of the addresses modulo 2,4,8, and thus determine the alignment information. Therefore, static alignment analysis amounts to pointer analysis, and to developing suitable data structures for the alignment information.

Let us fix a number k, e.g. $k = 4$, and consider addresses modulo k. Intuitively, the idea of our analysis is that for each pointer definition, we keep a set of possible remainders modulo k. If the set is the singleton set $\{0\}$, we know that the pointer is aligned modulo k. If, for a given modulo k at a certain point of the program, more than one remainder value is computed, then all of the remainders are stored. The alignment information is propagated independently for each k.

When a data array is defined or is dynamically allocated, we assume that its first element is aligned modulo 8, i.e., its remainder modulo 8 is equal to 0. Starting from this initial information, we propagate the alignment information though the program.

Let us first consider an example to demonstrate the way we propagate and store the alignment information. Figure 3 represents an example program (a), for which the alignment information modulo 4 is calculated (b). In statements S_1, S_2, and S_3 pointers `a`, `b`, and `c` of type `short` are initialized, and depending on the array element they point to, the alignment information follows. For instance, $\mathtt{c}:\{0\}$ in S_4(b) means that the address value held by pointer `c` at this pro-

```
S0 :  short array[N+1];
S1 :  short *a=&array[0];
S2 :  short *b=&array[1];
S3 :  short *c=&array[0];
S4 :  for (i=0; i < N; i++) {
S5 :      if (i&3) *c=1;
S6 :      else *c=0;
S7 :      c++;
S8 :  }
S9 :  for (i=0; i < N; i+=2) {
S10:      *a=*b+10;
S11:      a=a+1;
S12:      b=b+1;
S13:      *a=*b+10;
S14:      a=a+1;
S15:      b=b+1;
S16:  }
```

(a) example program

```
S0 :
S1 :  a:  {0}
S2 :  b:  {2}
S3 :  c:  {0}
S4 :
S5 :
S6 :
S7 :  c:  {0, 2}
S8 :
S9 :
S10:
S11:  a:  {2}
S12:  b:  {0}
S13:
S14:  a:  {0}
S15:  b:  {2}
S16:
```

(b) alignments

Figure 3: Propagation of intraprocedural alignment information for addresses modulo 4.

gram point is 0 modulo 4. In S_7(b) pointer `c` is annotated with two values: {0, 2}, since both values are possible. In contrast, all remainder values for the pointer definitions in the second loop are known precisely.

Now, we will give a formal definition of the annotation process. Let C be the set of control flow graph (CFG) nodes, and P be the set of pointer variables. For a set X, let $\mathcal{P}(X)$ denote its powerset, i.e., the set of subsets of X. A modulo k annotation is a function

$$a_k : C \times P \longrightarrow \mathcal{P}(\{0, \ldots, k-1\})$$

which associates a set of *possible* values modulo k with each pointer variable at any given node in the CFG. (In Figure 3, we show the annotations only in those places where they are defined, and provide new annotations only where they change.) In this paper, we consider the annotation functions a_2, a_4, a_8 which describe alignment information modulo 2,4,8. The annotations and the analysis are a special case of abstract interpretation. Consider for example the powerset $\mathcal{P}(\{0,1,2,3\})$. Here, $a_4(c,p) = \{0,1,2,3\}$ denotes that each value modulo 4 is possible, i.e., nothing is known about the actual value modulo 4. It corresponds to the $\bot$ element of the lattice. $a_4(c,p) = \mathcal{P}(\{2\})$ denotes that we know that the value of p at CFG node c modulo 4 equals 2. An empty set, such as $a_4(c,p) = \{\}$, means that no modulo values for p have been computed yet and it corresponds to the $\top$ element of the lattice. The meet operation of the lattice corresponds to set union of the alignment sets.

An iterative dataflow algorithm is used to collect the alignment information for each procedure. The algorithm traverses the control flow graph of the procedure in a top-down manner, and propagates the alignment information. Static single assignment form (SSA) [2] is used for basic blocks to collect all previous variable definitions. The right-hand side of each assignment to a pointer variable is analyzed to extract the alignment information. Addition, substraction, and multiplication operations commonly used in address calculation (i.e., "pointer arithmetic") are handled. The algorithm terminates when no set changes its value in the latest iteration.

Finally, to decide whether a data buffer `b` is aligned at a certain point in the program, the alignment information for the last definition of a pointer, which points to `b` is required. Suppose this definition is given at CFG node $c_\mathtt{b}$ and for pointer $p_\mathtt{b}$. Then an exact value modulo k can be obtained if the annotation information $a_k(c_\mathtt{b}, p_\mathtt{b})$ for the definition contains only a single value, i.e., $a_k(c_\mathtt{b}, p_\mathtt{b}) = \{r\}$. Consequently, if $a_k(c_\mathtt{b}, p_\mathtt{b}) = \{0\}$, we conclude that the buffer `b` is aligned modulo k.

3.2 Interprocedural Alignment Analysis

Analyzing each procedure in isolation leads to very conservative estimates. Therefore, in order to estimate the effects of procedure calls on alignment information, we perform a context-sensitive interprocedural alignment analysis. The general idea of the analysis is that the alignment information at the call-site is propagated to the called procedure (i.e., the called C function) to be used as alignment input information. The information is propagated through the body function, and the output obtained is then in turn used to update the alignment information at the call-site.

To perform the interprocedural analysis, we exploit the call graph of the program. The call graph is a directed graph. Each node of the graph represents a function, and each edge of the graph stands for a function call. There is one additional node, which represents all functions for which no source code is available, e.g. functions defined in standard libraries. Names of actual arguments of the caller function differ from names of corresponding formal parameters of the called function. Our call graph allows correct transmission of alignment information from the call-site to the called procedure and vice versa.

Interprocedural alignment analysis captures alignment information across functions influenced by both function parameters and global variables. Note that in the C language, all parameters are passed by value. (A pointer which is passed by value is similar in effect to call-by-reference.) Therefore, the interprocedural alignment information comprises three information sets for

(i) *global pointers* G,

(ii) *actual pointer arguments* P passed by value, and

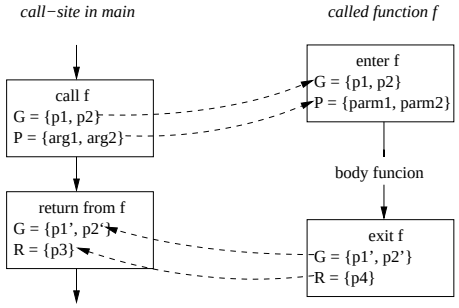
(iii) *return pointers* R returned by the called function.

We first explain the transition of the alignment information across a function call for each of the three information sets. A memory location referenced by a global pointer may be changed by the called function. Consequently, a set representing the alignment information for global variables is constructed, and propagated through the program. In contrast, a formal parameter of a function receives its value by assignment when the function is called. Information about the parameter's alignment flows only into the function; no information flows out at the end of the function. An opposite situation occurs when the function result is a pointer.

```
int *p1, *p2;    /* globals */
main() {
    int *p3, *arg1, *arg2;
    ...
    p3 = f(arg1,arg2);
    ...
}

int* f(int *parm1, int* parm2) {
    int *p4;
    ...
    return p4;
}
```

(a) example program



(b) transition of alignment information

Figure 4: Transition of the alignment information across a function call.

In this case, alignment information flows out only from the called function to the call-site.

Our interprocedural analysis follows the sequence of functions in the program. Analysis of the program starts from the `main` function. Each function call is visited sequentially. Intraprocedural analysis is used to compute the intraprocedural information sets for the function, until a function call is reached. These sets are used to update the corresponding interprocedural sets. After the update, the intermediate intraprocedural information is no longer needed.

Consider Figure 4 for an example program containing two functions, `main()` and `f()`. We show the transition of the interprocedural information across function calls. The call graph of the program is used to store context-sensitive information for each call.

The possibility of recursion amongst the functions means that an iterative approach must be used. If the input–output set of a function differs from the set in the previous iteration, the two sets are merged, and the new input set is used for the next iteration. The analysis algorithm terminates when no changes occur during an iteration.

To use the alignment information for SIMD optimization within a function body, the algorithm merges all possible calling contexts for that function, thus accounting for the multiple calling paths leading to the function.

Merging the different calling contexts may have the effect that different possible values for $a_k(c, p)$ are obtained from different calling paths, so that in some cases no alignment information can be obtained. In these cases, one may clone the function and create new copies of the function which have different alignment requirements for its parameters.

4. EXPERIMENTAL RESULTS AND DISCUSSION

In this section we evaluate the performance of our pointer alignment analysis. We have implemented the described analysis algorithms using the OCE compiler framework. In our experiments, we used several DSP kernels and some typical DSP applications as the C programs to be compiled. These show the effectiveness of the proposed algorithms for realistic multimedia processor tasks.

We have assembled a suite of benchmarks that includes typical DSP routines used in the real world. Our benchmark suite consists of the following DSP kernels: *array assignments*, *dot product*, *matrix operations*, and *filtering*. For the DSP kernel test cases, we compared the SIMD performance using dynamic checks to the performance using our static alignment analysis. The quality measures used are the number of processor cycles (obtained from a DSP processor simulator), and the length of the generated assembler code. The detailed numbers are described below.

We evaluated the quality of the pointer alignment analysis by determining the fraction of pointers in the program for which exact alignment information can be obtained. Alignment information is considered to be exact if the set of possible modulo values contains only one number. To obtain these statistics, we used C implementations of typical DSP applications: *gsm codec*, *fast fourier transform (fft)*, and *cerebella model arithmetic computer (cmac)*. The resulting statistics are provided in Figure 5. The white columns show the total number of pointer definitions in the program, and the black columns denote the number of pointer definitions for which the alignment information is detected by the analysis. For address values modulo 4 the analysis reports that 47% (fft), 61% (cmac), and 43% (gsm) of pointer definitions are annotated with explicit alignment information. Note that the vertical axis in the graph denotes numbers of pointer definitions, and not percentages.

As described above, we used the SIMD optimization to compile a set of kernel programs from the DSPStone benchmark suite. In this set of experiments, we assumed that the input pointers for all functions are naturally aligned. In Table 1 we compare four compilations for each kernel:

1. The first compilation does not use any SIMD instructions; the two columns under "No SIMD" show the processor cycles and the code size for the different programs.
2. In the next case ("no align info"), we do not use any alignment information but we can still sometimes use SIMD instructions to parallelize operations. For example, two independent additions or multiplications can

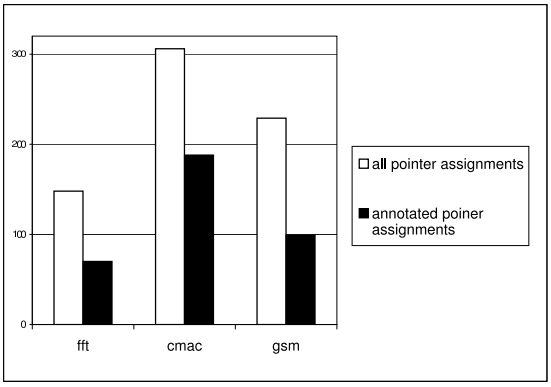


Figure 5: Statistics of pointer annotation.

be performed with SIMD instructions. In this case, we do not generate wide memory accesses. Instead we access the subwords of a register with separate memory access instructions.

3. In the third case ("dynamic align info"), we use dynamic checks to verify that all the pointers in a loop are naturally aligned. If this is the case, the program uses wide memory access instructions. Otherwise, the program uses subword memory access instructions. As in the previous case, SIMD instructions are used for parallelizing independent operations.

4. The fourth case ("static align info") extends the previous algorithm by using the static alignment information to avoid dynamic checks wherever possible.

Table 2 contains essentially the same information as Table 1, but in percentages. The figures are normalized with respect to the results of the non-SIMD compilation. We can see that, on average, the SIMD compilations with dynamic checks and static information show almost the same results in execution cycles, but the code size in the dynamic case is three times as large as in the non-SIMD and static cases.

Finally, in Table 3, we investigate the third case of Table 1 ("dynamic align info") in more detail. In contrast to the above experiments, we do not assume that all input pointers are aligned, but use some arbitrary alignment information; therefore the numbers in Table 1 and Table 3 do in general not coincide. In Table 3, the columns under "partial check" correspond to the dynamic check described in Table 1. In contrast to this, the columns under "exhaustive check" describe a more systematic dynamic check which distinguishes cases when within one loop, some pointers are aligned, and some are not. The small numbers in parentheses are percentage values relative to "partial check".

Table 3 shows that the static analysis gives much better performance than the partial dynamic checks. On the other hand, the numbers of processor cycles are nearly the same for the compilations with exhaustive dynamic check as for static information. However static analysis allows a reduction in the code size by a factor of five on average. In some cases, the factor is as much as 10. The results clearly demonstrate that our optimizations have the potential to greatly improve code quality.

5. RELATED WORK

In this paper we present SIMD optimizations including SIMD instruction generation and interprocedural static alignment analysis. The latter is similar to pointer analysis. Therefore, in this section we discuss related work on each of these: code transformation for the SIMD model, pointer analysis, and alignment issues.

For an extensive overview on the state of the art in code optimization for embedded processors, the reader is referred to [10].

5.1 Code Transformation for the SIMD Model

Recently, many research efforts have focussed on exploiting SIMD instructions for DSPs. The most closely related work to SIMD implementation is in [7]. The basic idea of the algorithm is to unroll the loop a few times depending on the operand type. The loop body is inspected, and structurally equivalent expressions are grouped if there is no data dependency between the elements. A vectorizing compiler similar in style to [7] was presented by Larsen Amarasimghe [8]. The algorithm performs loop unrolling, which is used to detect independent isomorphic statements within a basic block. Such statements are grouped together, and can be executed in parallel. The algorithm is provided with a cost function which estimates the effect of parallel instructions. Govindarajan and Sreraman [11] presented an implementation of a vectorizing compiler for MMX extension. Instead of the unrolling technique, their compiler identifies parallel data sections. To improve the performance of the algorithm, they introduced optimizing code transformations: strip mining, scalar expansion, grouping and reduction, loop fission, and loop distribution. Such code transformations may also be beneficial with our approach.

5.2 Pointer Analysis

Interprocedural data flow analysis has been studied in depth for many years. Different analyses provide different tradeoffs between accuracy and efficiency. The most efficient analyses are *flow-insensitive*, i.e. they compute a conservative summary, and do not take into account control-flow information of a procedure (see e.g. [12]). *Flow-sensitive* algorithms propagate appropriate information in each procedure, and compute information for every point in the program. This approach usually postpones analyzing a called procedure until the analysis of the current procedure converges. A detailed description of a flow-sensitive algorithm can be found, for instance, in [5]. The algorithm, we present in this paper, is context-sensitive. *Context-sensitive* algorithms preserve the calling context along each path of the call graph, which may require that each procedure may be analyzed for each call path. The basic structure of our algorithm is similar in style to the analysis in [4]. We analyze the program sequentially in the order in which function calls appear in the program. An iterative approach is used to handle recursion. We do not currently handle function pointers. Although the algorithm is exponential in nature, there is potential for further optimization.

Table 1: DSPStone kernels (in actual values)

procedure	No SIMD		SIMD					
			no align info		dynamic align info		static align info	
	cycles	code size	cycles	code size	cycles	code size	cycles	code size
sum_assign	3033	12	2532	15	1557	50	1533	13
mult_assign	4025	13	4526	21	3556	56	3526	18
zero_assign	1024	9	1027	12	1035	25	523	10
dot_product	5025	21	4134	35	2547	54	2529	26
n_real_updates	1346	36	955	45	671	91	655	42
startup	246	51	182	58	158	124	120	60
matrix1	5355	31	4088	37	2883	74	2842	36
matrix2	4933	38	3817	46	2309	161	2258	44
mat1x3	2065	17	1644	18	1177	30	1084	18
fir2dim	6567	50	5384	55	3942	245	3684	54
total sum	33619	278	28289	342	19835	910	18754	321

Table 2: DSPStone kernels (in percentages)

procedure	no align info		dynamic align info		static align info	
	cycles	code size	cycles	code size	cycles	code size
sum_assign	83	125	51	417	51	108
mult_assign	112	162	88	431	88	138
zero_assign	100	133	101	278	51	111
dot_product	82	167	51	257	50	124
n_real_updates	71	125	50	253	49	117
startup	74	114	64	243	49	118
matrix1	76	119	54	239	53	116
matrix2	77	121	47	424	46	116
mat1x3	80	106	57	176	52	106
fir2dim	82	110	60	490	56	108
averages	84	123	59	327	58	115

Table 3: Comparison of the exhaustive and partial dynamic check

procedure	dynamic align info				static align info	
	partial check		exhaustive check			
	cycles	code size	cycles	code size	cycles	code size
sum_assign	2558	50	2068 (81)	120 (240)	2033 (79)	13 (26)
dot_product	4150	54	3264 (79)	94 (174)	3229 (78)	26 (48)
startup	188	124	160 (85)	176 (142)	144 (77)	60 (48)
matrix1	4122	74	3284 (80)	104 (141)	3146 (76)	36 (49)
fir2dim	5624	245	4216 (75)	452 (184)	4036 (72)	54 (22)
total sum	16642	547	12992 (78)	946 (173)	12588 (76)	189 (35)

5.3 Alignment Issues

Another related area is the work done in detecting memory addresses. Davidson and Jinturkar [3] present an algorithm for coalescing redundant memory accesses in loops. The algorithm utilizes memory by means of combining narrow load and store instructions into wider memory operations. To perform transformations, the wide memory address has to be naturally aligned. Therefore, they determine whether it is necessary to insert dynamic alignment checks. Larsen et. al [9] present a static analysis for detecting congruence of memory addresses with respect to a modulo value. They also propose transformations which increase the number of congruent references. These methods aim to improve clustered memory designs, e.g., decentralized memory banks. The analysis has been implemented on a low level intermediate representation.

6. CONCLUSIONS

In this paper we have shown how pointer alignment analysis can be used to improve code quality for multimedia processors with SIMD instruction sets. We have described a method which statically determines alignment information for program pointers, and implemented our method using the OCE compiler framework. Initial experiments indicate that our method can significantly improve the quality of the code when compared to dynamic alignment checking. The code size can be reduced up to a factor of 4.5 by removing the dynamic checks. The number of cycles can be reduced by the degree of SIMD parallelism. We believe that avoiding the code duplication which comes with dynamic alignment checks eliminates unnecessary redundancies and complexity in the generated code, and thus facilitates efficient use of other optimization and analysis techniques.

In addition, the alignment information can serve as a basis for program transformations which will be the subject of future work. One possibility is to use alignment information to cause the compiler to layout arrays in memory differently. For example, if the alignment analysis shows that an array is unaligned for access by a loop where SIMD instructions would be appropriate, the array could be placed at a different address and thereby avoid the need for preloop code.

7. ACKNOWLEDGEMENTS

This work was supported by the Christian Doppler Forschungsgesellschaft and Infineon.

8. REFERENCES

[1] Atair. Open compiler environment. `www.atair.co.at`.

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[3] J. W. Davidson and S. Jinturkar. Memory access coalescing: A technique for eliminating redundant memory accesses. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 186–195, 1994.

[4] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, 1994.

[5] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[6] U. Hirnschrott. DSP compiler optimisation. Master's thesis, Institut für Computersprachen, Technische Universität Wien, December 2001.

[7] A. Krall and S. Lelait. Compilation techniques for multimedia processors. *International Journal of Parallel Programming*, 28(4):347–361, 2000.

[8] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *ACM SIGPLAN Notices*, 35(5):145–156, 2000.

[9] S. Larsen, E. Witchel, and S. Amarasinghe. Increasing and detecting memory address congruence. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 18–29, 2002.

[10] R. Leupers. *Code Optimization Techniques for Embedded Processors. Methods, Algorithms, and Tools.* Kluwer Academic Publisher, Boston, 2000.

[11] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 28(4):363–400, 2000.

[12] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[13] K. Vögler. A DSP C-compiler. Master's thesis, Institut für Computersprachen, Technische Universität Wien, April 2002.