Evaluation of a Potential for Automatic SIMD Parallelization of Embedded Applications

Rashindra Manniesing, **Ireneusz Karkowski** and **Henk Corporaal**
Delft University of Technology
Information Technology and Systems
Mekelweg 4, 2628 CD Delft, The Netherlands

Keywords: SIMD parallelism, multimedia instruction set extensions, automatic parallelization.

Abstract

This paper investigates the potential for automatic mapping of typical embedded applications to architectures with multimedia instruction set extensions. For this purpose a (pattern matching based) code transformation engine is used. The experiments show that about 85% of the loops which are suitable for *Single Instruction Multiple Data* (SIMD) parallelization can be automatically recognized and mapped.

1 Introduction

Many modern microprocessors feature extensions of their instruction sets, aimed at increasing the performance of multimedia applications. Examples include the Intel MMX, HP MAX2 and the Sun Visual Instruction Set (VIS) [5]. The extra instructions are optimized for operating on the data types that are used in multimedia algorithms (8-, 16- and 32-bit). The large word size (64 bits) of the modern architectures allows SIMD parallelism exploitation. For a programmer however, a task of fully exploiting these SIMD instructions is rather tedious. This because humans tend to think in a sequential way instead of a parallel. Therefore, ideally, a smart compiler should be used for automatic conversion of sequential programs into parallel ones.

One possible approach involves application of a programmable transformation engine, like for example the **ctt** (Code Transformation Tool) developed at the department CARDIT at Delft University of Technology [2]. The tool was especially designed for the source-to-source translation of ANSI C programs, and can be programmed by means of a convenient and efficient *transformation language*.

The purpose of this article is to show the possibilities and deficiencies of the **ctt**, in the context of optimizations for the multimedia instruction sets. This has been done by analyzing and classifying every FOR loop of a set of benchmarks individually, and comparing these results with the results obtained when **ctt** was employed. Two categories of SIMD transformations have been used. One allowing *reduction* variables in the loop body and one without. As the target architecture, the SUN Visual Instruction Set was selected (it seems to be most representative). The SIMD extensions of this instruction set are the subject of section 2. The **ctt** code transformation tool and the used SIMD transformations are described in detail in section 3. Section 4 presents characteristics of the used benchmarks. After that section 5 presents the experimental results and, finally, section 6 draws the conclusions.

2 The VIS multimedia extension

Multimedia algorithms typically work on small data entities for the pixel elements representation (eg. 8 or 16 bits). Therefore using 64 bit operations on them represents a large waste of resources. Modern processors solve this problem by introducing extra instructions being able to perform operations in a vector style. The complete set of data formats that can be handled by the VIS instructions is 4x8-bit, 2x16-bit, 4x16-bit and 2x32-bit. They fall into five classes: conversion, arithmetic/logical, address manipulation, memory access, and motion estimation [5]. In our context, the arithmetic/logical instructions are most interesting. Figure 1 (taken from [5]) presents a loop code before and after parallelization, illustrating the use of the most important VIS instructions. On the top of the figure, an ANSI C implementation of the 'vector dot product' operation, forming the inner loop of many signal-processing algorithms, is presented. This particular example has been taken, because it resembles a lot of the loops that have been used as source for the SIMD transformations in this work. In fact, if a similar loop occurs in a source, it should be recognized by the **ctt** as suitable for SIMD parallelization.

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

Transformation to VIS instructions (only innerloop)

```
Afh = vis_read_hi(ad4x16);
Bfh = vis_read_hi(bd4x16);
Afl = vis_read_lo(ad4x16);
Bfl = vis_read_lo(bd4x16);

tuh = vis_fmuld8sux16(Afh,Bfh);
tlh = vis_fmuld8ulx16(Afh,Bfh);
tul = vis_fmuld8sux16(Afl,Bfl);
tll = vis_fmuld8ulx16(Afl,Bfl);

tdh = vis_fpadd32(tuh,tlh);
tdl = vis_fpadd32(tul,tll);
rdh = vis_fpadd32(rdh,tdh);
rdl = vis_fpadd32(rdl,tdl);
```

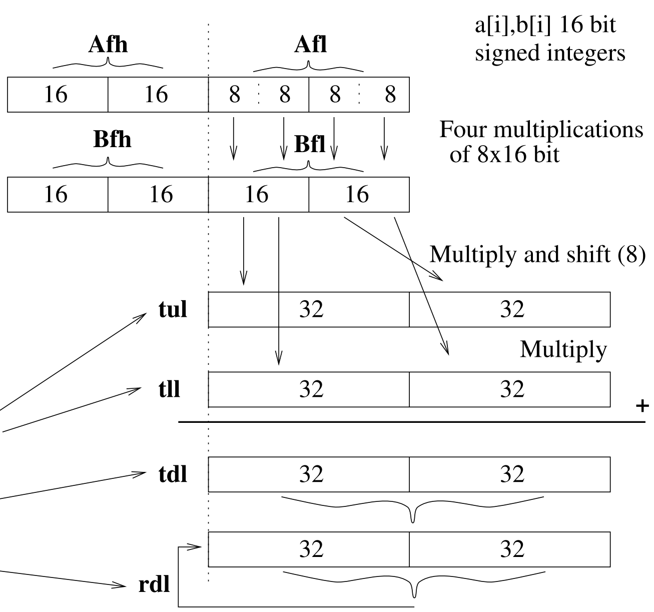

Figure 1: Parallelization of the vector dot product loop.

The remaining part of the figure 1 presents the VIS realization of the inner-loop[1]. The *reduction* variable `dotproduct` is 2x64 bits (rdh and rdl) and the input data `a[i]`, `b[i]` are 16 bits signed integers. The instruction set provides several instructions for multiplications. It has *fmul8x16* for pairwise-, *fmul8x16a(u/l)* for distributed- and *fmul(d)8sux16* for partitioned- multiplications. In all cases, these multiplication instructions are based on four small 8x16-bit multiplier subunits, and VIS uses a pipelined series of three instructions to emulate 16x16 multiplications. For example, fmuld8sux16 performs a (dual) 8x16 multiplication and shifts the result by 8 (that is multiply with 2^8) in order to get the right significant bits of the multiplication. Fmuld8ulx16 simply performs a 8x16 multiplications but together they are able to multiply 16-bit by 16-bit integers in similar manner people do when multiplying numbers with the paper-and-pencil method. For example, when calculating 23 multiplies 45, we first multiply 23 by 5, then 23 by 4, shift the latter result left (multiply 10) and then add the results. These partitioned multiplications require intermediate variables (tuh,tlh, tul, tll) of 32 bits.

Of course the remaining basic operators (add/sub/div) are supported as well. The other instructions of interest are the conversion instructions (fexpand, fpack16, fpack32, fpackfix and fpmerge). When overflow or underflow occurs in the packing instructions, VIS automatically clips them toward maximum register value or to zero, respectively. Furthermore, there are address handling instructions (such as alginaddr, ldd, faligndata etc), the memory access instructions and pixel distance instructions (used for motion estimation). For a detailed discussion of these instructions see [5].

3 Code transformation using cct

In this section we will give an overview of the **ctt** - a programmable code transformation tool, which we used for SIMD parallelization.

3.1 Tool description

Writing separate programs for different transformations can be avoided if we properly organize the process of applying a transformation. Our transformation engine uses such an organization. Its translation process is divided into 3 distinct stages:

- Pattern matching stage: In this stage the engine searches for code that has a strictly specified structure (that matches a specified pattern). Each fragment that matches this pattern is a candidate for the transformation.

- Conditions checking stage: Transformations can pose other (non-structural) restrictions on a matched code fragment. These restrictions include, but are not limited to, conditions on data dependencies and properties of loop index variables.

- Result stage: Code fragments that matched the specified structure and additional conditions are

[1]Do not try to understand all details; you need the VIS manual.

replaced by new code, which has the same semantics as the original code.

The structure of the *transformation language* used by **ctt** closely resembles these steps, and contains three subsections called PATTERN, CONDITIONS and RESULT. As can be deduced, there is a one to one mapping between blocks in the transformation definition and the translation stages. While a large fraction of the embedded systems are still programmed in assembly language, the ANSI C has become a widely accepted language of choice for this domain. Therefore, the *transformation language* has been derived from the ANSI C. As result, all C language constructs can be used to describe a transformation. Using only them would however be too limiting. The patterns specified in the code selection stage would be too specific, and it would be impossible to use one pattern block to match a wide variety of input codes. Therefore the transformation language is extended with a number of meta-elements. Among others the following meta-elements can be used to specify generic patterns, i.e. patterns that represent more than one element in the input C sources:

- Statements: keyword STMT represents any statement.
- Statement lists: keyword STMTLIST represents a list of statements. A statement list may be empty.
- Expressions: keyword EXPR represents any expression.
- Variables: keyword VAR represents any variable (of any type).
- Procedure calls: keyword PROCCALL represents any procedure, which satisfies specific requirements.

Meta-element must be assigned a number (except for variables), which should be included in braces behind their keyword (STMT(1) and STMT(2) represent then different C statements). Some meta-elements take also additional arguments (for example BOUND and STEP_EXPR take also as argument the name of the loop index variable). Figure 2 shows the (slightly simplified) specification of the transformation implementing SIMD parallelization from figure 1. In the next subsection we will analyze it in more detail.

3.2 Example of an SIMD transformation specification

The specification of an example transformation used for SIMD parallelization is shown in figure 2. In this description, some details have been left out (for example inclusion of the header files in the beginning). The specification starts with the search pattern description. In there, the FOR loop (used for matching) assumes well defined boundaries. These can be obtained by applying the pre-processing step which normalizes all FOR loops of the source code. Within the loop body, we can see two statement lists, a statement and a `MARK` meta-element. The statement will match with the multiplication of two 16-bit signed integers, while the accumulator variable `a`, must be 64 bit long. This is an example of the statement with *reduction*. *Reduction* refers to an accumulator variable in the expression within the loop body. The `MARK` meta-element is used to refer later to the statement itself (from within the conditions block).

The result block starts with the creation and definitions of types of intermediate variables. Some of them have been left out, to prevent the figure becoming too large. After that the first, third and fourth FOR loops handle the statement lists 1 and 2, and the remaining iterations of the second loop (`ub` modulo 4).The most interesting part is of course the second loop. It implements the SIMD parallelization of the statement from the pattern block. It is analogous to the parallelization of the vector dot product from figure 1. To make the computations clear, the same name convention has been used for the intermediate variables as in the vector dot product example.

The condition block checks the upper-bound of the loop-index `EXPR(1)` and dependencies between different parts within the loop body. The upper-bound must be a constant, and no dependencies from `STMTLIST(2)` to `STMTLIST(1)` (and the other two) are allowed.

Note that this transformation is actually a combination of two simpler transformations. The first one is the well known loop fission [7] (which allows us to handle loops which have more than one statement). The second one converts a simple statement loop into a SIMD loop.

Some remarks are in order. For simplicity of presentation we ignored the problem of unaligned arrays (the extension is straightforward [3]). Secondly, the above transformation will work only for arrays with elements of type "int". Very similar transformations may be written for other basic types. We ignored the problem of statement lists being empty. It is not serious - the post-processing passes may be used to remove loops with empty bodies (alternatively we could write separate transformations for these cases). Finally, a parallel loop may contain several statements suitable for mapping onto SIMD instructions. To exploit this potential the above transformation (and the others) should be applied repeatedly until no more candidates are found.

```
PATTERN
{
    VAR i;
    VAR a,B[DONT_CARE],C[DONT_CARE];

    for(i=0; i<=EXPR(1); i++) {
        STMTLIST(1);
        MARK(1);
        a= B[i]*C[i];
        STMTLIST(2);
    }
}
RESULT{
    VAR i;
    VAR a,B[DONT_CARE],C[DONT_CARE];
               /* arrays of 16 bit signed integers*/
    VAR bfl,cfl,bfh,cfh;
               /* Intermediate variables each 2x16 bit*/
    VAR bf,cf,ub,tuh,tlh,tul,tll,tdh,tdl,td,aa;
               /* Intermediate variables each 2x32 bit*/
    DEFINE_TYPE_FROM_STRING("ub", "int");
    DEFINE_TYPE_FROM_STRING("bfl", vis_f32_s);
    DEFINE_TYPE_FROM_STRING("cfl", vis_f32_s);
    ...
    DEFINE_TYPE_FROM_STRING("td", vis_d32_s);
    DEFINE_TYPE_FROM_STRING("aa", vis_d32_s);
    for(i=0;i<=EXPR(1);i++){
      STMTLIST(1);
    }
    ub=EXPR(1)/4;
    for(i=0;i<ub;i++){
      bfh = *(vis_f32 *)(B+i*4);      /*load 4x16 bit signed*/
      bfl = *(vis_f32 *)(B+i*4+2);    /*integers, into two steps*/
      cfh = *(vis_f32 *)(C+i*4);      /*Results into variables of*/
      cfl = *(vis_f32 *)(C+i*4+2);    /*2x16 bit (bfh,bfl,cfh,cfl)*/

      tuh = vis_fmuld8sux16(bfh,cfh); /*multiplies 8x16 and shift8*/
      tlh = vis_fmuld8ulx16(bfh,cfh); /*multiplies 8x16, no shift*/
      tul = vis_fmuld8sux16(bfl,cfl); /*for higher (bfh,cfh) and for */
      tll = vis_fmuld8ulx16(bfl,cfl); /*lower (bfl,cfl). Results 2x32 bit*/

      tdh = vis_fpadd32(tuh,tlh);     /*2x32 bits additions*/
      tdl = vis_fpadd32(tul,tll);
      td = vis_fpadd32(tdh,tdl);
      aa = vis_fpadd32(aa,td);        /*Final result into 2x32 bit variable*/
    }
    for(i=4*ub;i<=EXPR(1);i++){
      a=a+B[i]*C[i];
    }
    for(i=0;i<=EXPR(1);i++){
          STMTLIST(2);
    }
}
CONDITIONS {
  var_is_type(a,"long int");
  var_is_type(B,"int []");
  var_is_type(C,"int []");
  expr_is_constant(1);
  not(dep("true DISTANCE>=(1) between stmtlist 2 and stmtlist 1"));
  not(dep("true DISTANCE>=(1) between mark 1 and stmtlist 1"));
  not(dep("true DISTANCE>=(1) between stmtlist 2 and mark 1"));
}
```

Figure 2: Multiplication pattern with reduction.

Bench-marks	**Description**	**FOR loops**	**Outer-loops**	**SIMD**	**Non-SIMD**			
					Fn	In	Cm	Dp
arfreq	Autoregressive freq. estim.	2	0	0		1		1
g722	Adaptive differential PCM	12	0	7		4	1	
instf	Frequency tracking	9	1	3	1	2	1	1
interp3	Sample rate conversion	3	0	0	1			2
mulaw	Speech compression	1	0	0	1			
music	Music synthesis	4	1	0	1	1		1
radproc	Doppler radar processing	7	2	1	2		1	1
rfast	Fast convolution using FFT	9	0	5	3	1		
rtpse	Spectrum analysis	10	2	3	3	1		1
Total		57	6	19	12	10	3	7

Table 1: Overview of DSP benchmarks.

4 Benchmarks characterization

In our experiments we used two sets of benchmark files - the DSP benchmarks [4] and the MPEG2 Encoder [6] benchmark. They consist of 9 and 15 files, and have a total number of 57 and 171 FOR loops, respectively. All the loops have been individually classified. Tables 1 (DSP) and 3 (MPEG) summarize their most important characteristics.

Concentrate on table 1, which presents the DSP benchmarks. After the file's name, the number of FOR loops it includes is denoted, followed by the column "outer-loops". A loop is an "outer-loop" if it contains another FOR loop in its body, but without any other statements. Clearly an outer-loop has no use for parallelization because its body contains nothing else but another FOR loop. In the DSP benchmarks, the maximum depth of nested loops did not exceed two.

The column "SIMD" denotes the number of FOR loops which should be suitable for SIMD parallelization (according to manual inspection). The remaining loops were not suitable for SIMD. Their numbers are captured in the last four columns, and are classified into the following categories:

- *Fn* (function) - the body has a function call or procedure call.
- *Dp* (dependency) - there is inter-iteration dependency in the loop body.
- *Cm* (compare) - IF statement or a SWITCH statement has been used.
- *In* (init) - the only purpose of this loop is to initialize some variables.

Note that this is not an exclusive classification. For example, a FOR loop may simultaneously not be suitable for SIMD because of dependencies (depend +1) and a case statement in the loop body. In the table, only one classification will then be made. This would be 'Cm', because compare (as well as 'In' and 'Fn') allows some parallelization with the right patterns or pre-processing steps, unlike the classification 'Dp'. In other words, the classification 'Dp' has always the highest priority. Following above general rule leads to an easy check-up function within the tables 1 and 3: summation of outer-loops, SIMD and non-SIMD should be equal to the total number of FOR loops.

The loops labeled 'In' can be easily mapped on VIS instructions, but the overall speedup will be small and therefore this particular transformation has been left out. In any case, writing transformation patterns for initialization loops is simple compared to a multiplication or a division pattern.

The results, and other aspects concerning some of the files of the benchmarks, are presented in the following section.

5 Results

In this section we will describe and analyze the experiments that we performed to evaluate the potential for the automatic SIMD parallelization in embedded programs. Let us start with the description of the experimental framework.

5.1 Experimental framework

As was mentioned in the introduction section, we used two types of transformations for SIMD parallelization, one with *reduction* and one without. *Reduction* refers to an *accumulator* variable in the expression within the loop body. Within each type (with or without *reduction*), the pattern block only differs in the operator and this results in 8 different patterns to search for in the source code (we consider $+, -, *, /$ operators only).

To make a successful parallelization possible, a number of pre-processing steps need to be applied on the source code [2]. These steps involve the following: the first step flattens tree expressions inside loops. It will break up long expressions by introducing temporary variables. The next step normalizes all loops,

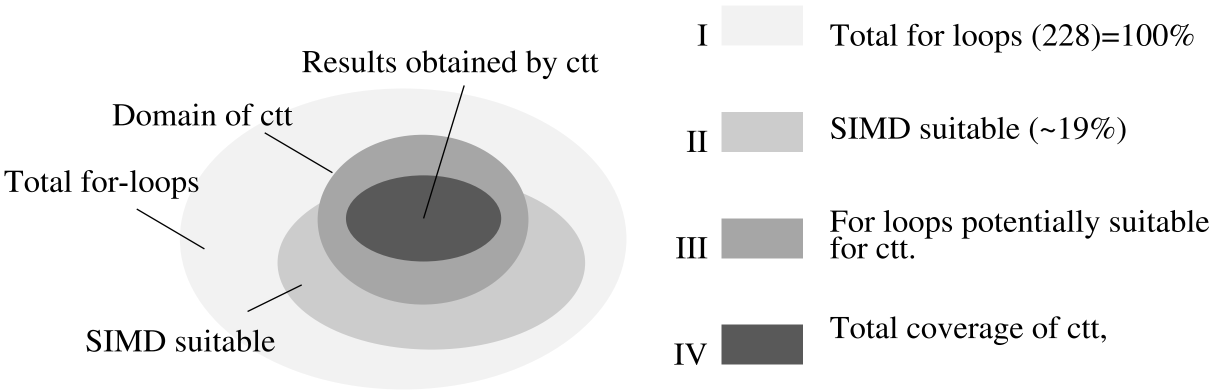

Figure 3: Diagram of all FOR loops from the benchmarks.

resulting in a uniform index descriptions. Finally, the third step expands scalars into arrays inside loops. The last step is necessary to make loop fission transformation [7] (being part of each transformation; recall section 3.2) legal. The fission allows us to handle loops containing more than one statement.

From those steps, only the second one has been applied, because the front-end SUIF trajectory [1], which we use, did not support the others. Fortunately, in order to determine the potential for SIMD parallelization we do need these steps. Instead, we used patterns which have very general expressions. For example, the multiplication expression (the statement from transformation example in figure 2) became `a=a+EXPR(1)*EXPR(2)`. This relaxation is possible because our purpose was to estimate the number of loops that can be automatically parallelized and to compare this number with the number of loops which are actually suitable for the SIMD parallelization. The results obtained in this way will be summarized in three tables in the next section.

5.2 Parallelization results for the DSP benchmarks

The DSP benchmarks consist of 9 files, with a total number of 57 FOR loop of which 19 FOR loops seem to be suitable for SIMD mapping (table 1). Table 2 shows all the matches that were found using **ctt**.

At first glance, there seem to be some contradictions between the tables. For example, the first benchmark *arfreq* has no loops which are suitable for parallelization according to table 1, while according to table 2, four matches were found when running **ctt**. The reason is that table 2 only presents the results of the code selection stage. It shows how well **ctt** performs in pattern matching and describes, for a valid FOR loop, which patterns need to be applied for parallelization. Another problem can arise in reading the tables. For example, the *g722* file has 7 FOR loops which are suitable for parallelization, and **ctt** finds a total number of 9 matches. This is caused by the multiple matches within the same (multi-statement) loop body.

5.3 Parallelization results for MPEG2 Encoder

Table 3 summarizes the results obtained on the MPEG2 encoder benchmark. From a total number of 171 FOR loops only 26 are suitable for SIMD mapping. It is also clear that **ctt** was not able to match any pattern at all for 5 files, even though the patterns were written as general as possible. This concerns a total of 29 FOR loops which cannot be parallelized.

5.4 Discussion

Consider an overview of all the results, presented in figure 3, which illustrates the domain of all FOR loops. This domain consists of four different regions: Region I, the most light-gray circle, shows a total number of 228 FOR loops (100%). From these, approximately 19% are found (by manual inspection) suitable for SIMD mapping (region II). Region III includes all the loops which **ctt** should be able to find (also non-SIMD), and region IV denotes the actual results obtained by **ctt**.

The part of region I outside of region II presents all the loops not suitable for SIMD mapping. It includes the loops classified in section 4 'Dp' (depend, 14%) and 'outer-loop' (28%). The other loop categories ('Fn'-function, 'In'-initialization and 'Cm'-compare) have a certain number of FOR loops which could possibly belong to the domain of **ctt**. Therefore region III (domain of **ctt**) covers part of the FOR loops outside region II.

Region II represents the loops suitable for SIMD mapping and has a large potential for **ctt** to exploit. In [5] four widely used algorithms are described, which should benefit from VIS instructions: the *separable*

Bench-mark	Without Reduction				With Reduction			
	add	sub	mul	div	add	sub	mul	div
arfreq	2	1					1	
g722	4	2	1				2	
instf	5		1				2	
interp3	1							
mulaw	1							
music	2			1				
radproc	4	2	1				1	
rfast	3	1	1					
rtpse	3	1	2				1	
Total	25	7	6	1			7	

Table 2: Pattern matches (with **ctt**).

File	FOR loops	Outer-loops	SIMD	Classification				Matches with ctt			
				Fn	In	Cm	Dp	add	sub	mul	div
fdctref	8	2	0	2	2		2	2		1, 2r	
idct	3	0	0	2		1			-No match-		
motion	33	15	10			8		21, 1r	12	5r	
mpeg2enc	14	0	0	1	3	8	2		-No match-		
predict	25	12	0	2			11	11			
putbits	1	0	0			1			1		
puthdr	2	0	0	2					-No match-		
putmpg	2	0	0			2			-No match-		
putpic	8	1	0	4	3				-No match-		
quantize	6	0	6					2			4
ratectl	4	2	1				1	2		1r	
readpic	20	4	4	6			6	4			
stats	30	14	1	2		11	2	4	1	1r	
transfrm	15	7	4			3	1	5	1	3r	
Total	171	57	26	21	8	34	25	51, 1r	15	1, 12r	4
								No matches: 29 loops			

Table 3: Overview of MPEG2 encoder benchmark (r - with reduction).

convolution, sum of absolute differences, trilinear interpolation and *the vector dot product*. The 8 patterns, which we use, are all derived from the *vector dot product*. The other three algorithms are not covered at all. This explains the part of region II not covered by region III.

As a consequence, the region III has two ways to expand: first, by writing the patterns and all its derivatives, specific for the other three algorithms, resulting in a larger cover of region II. And second, by handling the loops classified as Fn/In/Cm in region I, which can result in larger coverage of region I.

5.4.1 Speedup

As could be seen in both previous sections the coverage of **ctt** is reasonable (85% of SIMD suitable loops are recognized). If we however take into account that the region II represents only 19% of the total number of loops, the question arises if the obtained in this way SIMD parallelization is worth the effort. The answer to this question very much depends on the benchmark in question. The final speedup will namely depend on the fact if we were are to parallelize to most frequently executed parts of a given benchmark. This speedup may be calculated using the following formula:

$$s = \frac{L_{total}}{L_{total} - \sum_{i \in P} L_i + \sum_{i \in P} L_i / s_i} \quad (1)$$

where L_{total} is the total sequential execution time of the benchmark, P the set of parallelized loops, L_i sequential latency of parallelized loop i and s_i local speedup obtained in loop i.

As example consider the **instf** benchmark. One of its two most important parts is the routine *lms*, which includes 3 very frequently executed loops. Two of these loops are perfectly suitable for SIMD parallelization and are without problems parallelized by the **ctt**. Since they both constitute about 40% of the total execution time of the benchmark, the obtained speedup[2] amounts to approx. 1.5.

5.4.2 Improvements

Further we conclude the following:

- The inspection of the benchmarks shows that inter-procedural transformations as a pre-processing steps are justified (12+21 occurrences).

- In the benchmarks, initializations of variables within a loop (that is initialization at zero, or one-to-one copy of another variable or array) does occur often (9+5 occurrences) pleading for parallelizing them as well. Especially, because these transformations are simple to write.

- From the table 2 and 3, we learn also that most matches occur with the addition expression (59% of total matches). The pattern library should therefore at least contain addition transformations for various types of the variables and/or arrays.

- Expressions with reduction are in minority compared to expressions without reduction (20 and 110, respectively), one suggestion could be to break the first type of expressions into several ones (another atomization pre-processing step), limiting this way the size of the transformation library.

6 Conclusions

In this paper we investigated the potential for automatic SIMD parallelization of embedded applications. For this purpose a programmable (pattern matching based) code transformation engine was used. In our experiments we were able to automatically recognize and map about 85% of the loops which were suitable for SIMD mapping. While this number is quite high, in general large coverage does not guarantee the overall speedup in the application. This speedup depends also on the execution time profile, which is independent from number of SIMD suitable loops. While clearly there exists a limit on the number of loops which can be automatically parallelized [5], increasing the coverage of an automatic SIMD parallelizer is certainly advantageous. The application of the inter-procedural transformations, has been identified as the most promising direction.

[2] Overhead of SIMD approach depends on the processor SIMD support and can be substantial.)

References

[1] Saman P. Amarasinghe, Jennifer M. Anderson, Christopher S. Wilson, Shin-Wei Liao, Brian R. Murphy, Robert S. French, Monica S. Lam, and Mary W. Hall. Multiprocessors From a Software Perspective. *IEEE micro*, pages 52–61, June 1996.

[2] Maarten Boekhold, Ireneusz Karkowski, and Henk Corporaal. Transforming and Parallelizing ANSI C Programs Using Pattern Recognition. In *HPCN Europe'99*, Amsterdam, NL, April 1999.

[3] Gerald Cheong and Monica S. Lam. An Optimizer for Multimedia Instruction Sets. In *Proceedings of the Second SUIF Compiler Workshop*, Stanford University, USA, August 1997.

[4] P.M. Embree. *C Language Algorithms for Real-Time DSP*. Prentice-Hall, 1995.

[5] Marc Tremblay et al. Vis speeds new media processing. *IEEE micro*, August, 1996.

[6] MPEG Software Simulation Group, http://www.mpeg.org/index.html/MSSG/#source. *MPEG-2 Video Codec*, 1996.

[7] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.