Automatic SIMD Parallelization of Embedded Applications Based on Pattern Recognition

Rashindra Manniesing[1], Ireneusz Karkowski[2], and Henk Corporaal[3]

[1] CWI, Centrum voor Wiskunde en Informatica,
P.O.Box 94079, 1090 GB Amsterdam, The Netherlands,
r.manniesing@cwi.nl
[2] TNO Physics and Electronics Laboratory,
P.O. Box 96864, 2509 JG Den Haag, The Netherlands,
karkowski@fel.tno.nl
[3] Delft University of Technology, Information Technology and Systems,
Mekelweg 4, 2628 CD Delft, The Netherlands,
h.corporaal@its.tudelft.nl

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, which involves a three-step process of matching, condition checking and replacing of the source code. Experiments with DSP and the MPEG2 encoder benchmarks, 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) [1]. The extra instructions are optimized for operating on the data types that are typically used in multimedia algorithms (8, 16 and 32 bits). The large word size (64 bits) of the modern architectures allows SIMD parallelism exploitation. For a programmer however, task of fully exploiting these SIMD instructions is rather tedious. This because humans tend to think in a sequential way instead of a parallel, and 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 capabilities and deficiencies of CTT, in the context of optimizations for the multimedia instruction sets. This has

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 349–356, 2000.

been done by analyzing and classifying every FOR loop of a set of benchmarks manually, and comparing these results with the results obtained when CTT was employed.

The remainder of this paper is organized as follows. The CTT code transformation tool and the used SIMD transformations are described in detail in Sect. 2. After that, Sect. 3 describes the experimental framework, Sect. 4 presents the results and discussion. Finally, Sect. 5 draws the conclusions.

2 Code Transformation Using CTT

The CTT – a programmable code transformation tool, has been used for SIMD parallelization. The transformation process involves three 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 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 a 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*, which are used to specify generic patterns. Examples of meta-elements, among others, are the keyword STMT representing any statement, the keyword STMTLIST representing a list of statements (which may be empty), the keyword EXPR representing any expressions, etc. We refer to [2] for a complete overview, and proceed with a detailed example of a pattern specification.

Example of a SIMD Transformation Specification

The example given describes the *vectordot product loop* [1]. The *vectordot product loop* forms the innerloop of many signal-processing algorithms, and this particular example is used, because we base our experiments on this pattern and a number of its derivatives.

```
PATTERN{     VAR i,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 signed int.(16 bits)*/
             VAR bfl,cfl,bfh,cfh;                       /* Intermediate var.(2x16 bits)*/
             VAR bf,cf,ub,tuh,tlh,tul,tll,tdh,tdl,td,aa; /* Intermediate var.(2x32 bits)*/
             DEFINE_TYPE_FROM_STRING("ub", "int");
             DEFINE_TYPE_FROM_STRING("bfl", vis_f32_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 bits 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 bits*/
                 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 (2x32 bits)*/}
             a = getL(aa)+getH(aa);
             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"));}
```

Fig. 1. Vectordot product pattern with reduction

Figure 1 shows the specification in which 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 preprocessing 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 bits signed integers, while the accumulator variable `a`, must be 64 bits long. This is an example of a 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 condition 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.

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 [3] (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 [4]). 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.

3 Experimental Framework

From this pattern similar ones have been derived to form a class of patterns to search for by CTT in the experiments. We used two types of transformations for SIMD parallelization, one with reduction in the loop body and one without. Furthermore, within each type, the pattern block differs in the operator which results in a total of 8 different patterns (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 expression trees inside loops. It will break up long expressions by introducing temporary variables. The next step normalizes all loops, resulting in uniform index descriptions. Finally, the third step expands scalars into arrays inside loops. The last step is necessary to make loop fission [3] (being part of each transformation; recall section 2) legal. Loop 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 [5], which we use, did not support the others. Unfortunately, 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 Fig. 1) 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

Table 1. Benchmarks characterization ('r' – with reduction)

Bench-marks	Description	FOR loops	Outer-loops	SIMD	Non-SIMD *Fn*	*In*	*Cm*	*Dp*	CTT matches *add*	*sub*	*mul*	*div*	SIMD map
arfreq	Autoregr. freq. estim.	2	0	0		1		1	2	1	1r		0
g722	Adaptive diff. PCM	12	0	7		4	1		4	2	1,2r		6
instf	Frequency tracking	9	1	3	1	2	1	1	5		1,2r		3
interp3	Sample rate conversion	3	0	0	1			2	1				0
mulaw	Speech compression	1	0	0	1				1				0
music	Music synthesis	4	1	0	1	1		1	2			1	0
radpr	Doppler radar proc.	7	2	1	2		1	1	4	2	1,1r		1
rfast	Fast FTT convolution	9	0	5	3	1			3	1	1		4
rtpse	Spectrum analysis	10	2	3	3	1		1	3	1	2,1r		3
mpeg2	Video/MPEG2-enc	171	57	26	21	8	34	25	51,1r	15	1,12r	4	22
Total		228	63	45	33	18	37	32	76,1r	22	7,19r	5	39

to compare this number with the number of loops which are actually suitable for the SIMD parallelization. The results obtained this way will be summarized in one table in the next section.

4 Results and Discussion

In our experiments we used two sets of benchmark files – the DSP benchmarks [6] and the MPEG2 Encoder [7] benchmark. They consist of 9 and 15 files, and have a total number of 57 and 171 FOR loops, respectively. All loops have been individually classified. Table 1 summarizes their most important characteristics.

Concentrate on Table 1. After the file's name, the number of FOR loops it includes is denoted, followed by the column *outer-loop*. A loop is defined as 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 all 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 column "Non-SIMD", and are classified into the following categories:

- *Fn* (function) – the body has a function call or procedure call.
- *In* (init) – the loop initializes some variables by setting them to fixed values.
- *Cm* (compare) – IF statement or a SWITCH statement has been used.
- *Dp* (dependency) – there is inter-iteration dependency in the loop body.

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 possibly because of 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 preprocessing 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 table: summation of outer-loops, SIMD and non-SIMD should be equal to the total number of FOR loops.

The remaining columns denote the actual results obtained by running CTT ('r' in the table is 'reduction'), of which the last column might be the most interesting as it directly shows how well CTT performs with this particular transformation library of 8 patterns (the "SIMD map" results are manually verified).

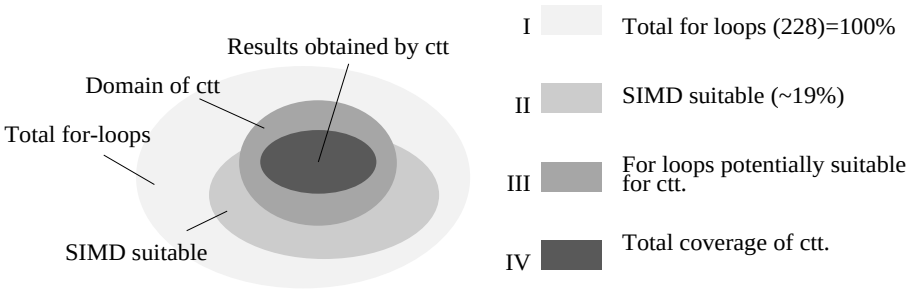

Fig. 2. Diagram of all FOR loops from the benchmarks

Discussion

At first glance, there seems to be some contradictions within the table. For example, the first benchmark *arfreq* has no loops which are suitable for parallelization, while according to "CTT matches" in the table, four pattern matches were found when running CTT. The reason is that the experiments 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, when reading the table. For example, the *g722* file has 7 FOR loops which are suitable for parallelization, while CTT finds a total number of 9 matches. This is caused by the multiple matches within the same (multi-statement) loop body.

Consider a graphical overview of all the results, presented in Fig. 2, 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 the table as '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 [1] four widely used algorithms are described, which should benefit from VIS instructions: the *separable convolution, sum of absolute differences, trilinear interpolation* and *the vectordot product*. The 8 patterns, which we use, are all derived from the *vectordot 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 coverage 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.

Speedup As could be seen in both previous sections the coverage of CTT is reasonable (approximately 85% of SIMD suitable loops are recognized). However, if we take into account that the region II represents only 19% of the total number of loops, the question arises if the SIMD parallelization obtained this way 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 are able to parallelize the most frequently executed parts of a given benchmark. This speedup may be calculated using the following formula:

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

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 an 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 CTT. Since they both constitute about 40% of the total execution time of the benchmark, the obtained speedup[1] amounts to approximately 1.5.

Improvements Further we conclude the following:

- The inspection of the benchmarks shows that inter-procedural transformations as a pre-processing steps are justified (33 occurrences).

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

- 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 (18 occurrences) pleading for parallelizing them as well. Especially, because these transformations are simple to write.
- From Table 1, 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 (19+1=20 and 76+22+7+5=110, respectively). A suggestion is to break the first type of expressions into several ones (another atomization pre-processing step), limiting this way the size of the transformation library.

5 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 the number of SIMD suitable loops. While clearly there exists a limit on the number of loops which can be automatically parallelized [1], increasing the coverage of an automatic SIMD parallelizer is certainly advantageous. The extension of the inter-procedural transformations, has been identified as the most promising direction.

References

1. Marc Tremblay et al. Vis speeds new media processing. *IEEE micro*, August, 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. Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.
4. 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.
5. 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.
6. P.M. Embree. *C Language Algorithms for Real-Time DSP*. Prentice-Hall, 1995.
7. MPEG Software Simulation Group, http://www.mpeg.org/index.html/MSSG/#source. *MPEG-2 Video Codec*, 1996.