A Retargetable Preprocessor for Multimedia Instructions*

Gilles Pokam, Julien Simonnet, and François Bodin

IRISA
Campus Universitaire de Beaulieu,
35042 Rennes Cedex, France
{gpokam,jsimonne,bodin}@irisa.fr

Abstract

Request for more computation power in the media computing domain has led to multimedia extensions in the instruction sets of modern processors. This has resulted in the implementation of new instructions which handle short data types and exploit subword parallelism for improving the performance. Compiler developments which aim at providing some degree of support to these instructions now emerge and usually fall into three categories: vectorization, idiom recognition and code generation. Unfortunately there is no consensus among constructors on the definition of multimedia instructions. Therefore, very few works have addressed the general issue of exploiting multimedia instructions from high level languages.

In this paper, we present a prototype preprocessor for exploiting multimedia instructions in "C" code. Our preprocessor relies on idioms recognition techniques for detecting potential usages of multimedia instructions and exploits a rule-based graph abstraction for the rewriting process.

1 Introduction

In order to respond to the high computation power demand of multimedia applications, multimedia instructions have been added to recent microprocessors [15, 16, 17]. These new instructions exploit subword parallelism to improve the performance of multimedia applications. Subword parallelism [3] provides a mean for multiple data packed in a register to be processed in parallel using multiple functional units. Current use of these instructions is mostly confined to two approaches:

1. the use of optimized library routines often written in assembly code,
2. and the insertion of compiler intrinsics that give access to the multimedia instructions as illustrated in the TriMedia code of Figure 1.

To exploit multimedia performance, application programmers have to manually detect potential usages of these instructions or/and deal with assembly language. Developments in compilers aiming to alleviate this difficulty focus on three major issues:

1. vectorization techniques [11] and/or some loop transformations [1] such as loop unrolling,
2. idioms recognition [5, 10],
3. and code generation [12].

The vectorization mainly consists in implementing a vectorizer based on Kennedy's algorithm [11]. Such vectorizers are usually based upon loop analysis and code transformation techniques for detecting parallel sections of a loop's body that are candidates for vectorization. Common transformation techniques used by these vectorizers include strip mining, scalar expansion, loop distribution and the lowering of the iteration space [7, 13]. In other studies, e.g in [9], it is also proposed to rely on loop unrolling to recover the parallelism hidden in scalar statements.

*This research has been supported in part by STMicroelectronics.

```
void example1(char *back,char *forward,char *idct,char *dest)
{
   int i;
   for (i = 0; i < 64; i += 1)
      dest[i] = ((back[i] + forward[i] + 1) >> 1) + idct[i];
}
```

a) "C" non-optimized code

```
void example2(char *back,char *forward,char *idct,char *dest)
{
   int i,temp;
   int *iback=back;int *ifor=forward;int *iidct=idct;int *idest=dest;
   for (i = 0; i < 16; i += 1)
   {
     temp = QUADAVG(iback[i], ifor[i]);
     idest[i] = DSPUQUADADDUI(temp, iidct[i]);
    }
}
```

b) Trimedia hand-optimized code of a)

Figure 1: Example of a multimedia kernel (MPEG code reconstruction)

When vectorization techniques usually fail or when some specialized operations (e.g sum of absolute differences, dot product and so on) are available as multimedia kernels, idioms recognition algorithms can be used to extract such specific source code patterns and replace them by the available optimized versions [5].

The last issue, code generation, usually deals with emitting the appropriate optimized instructions for each vector or pattern instruction that is encountered. By doing this, it must address several other issues, among which: data alignment [7, 8, 14], register management [7, 8], and packing of operations [9].

However, the issues discussed above are not efficient enough for exploiting multimedia instructions from high-level languages. Three main reasons make this goal difficult to achieve. First, multimedia instructions differ largely in their types. For instance, some multimedia processors provide saturated arithmetic whereas others do not. Secondly, within a single processor family, the evolution of multimedia ISA is very rapid. For example, on the Intel processors family, the MMX multimedia extension has successively evolved to the SSE in the Pentium III and SSE2 in the Pentium IV. Finally, in a more general manner, multimedia instruction sets vary frequently from one processor family to another. This variability of the multimedia ISA poses a challenge to compilers. Thus, retargetability becomes a major issue when new multimedia instructions have to be added or when switching from one target architecture to another with different multimedia ISAs as often encountered in application domains such as embedded computing.

Very few works have addressed this latter issue in a systematic manner. In [7] and [14] for instance, retargetability is addressed at the code generation level. However, the technique employed is to some extent *ad hoc,* since some modifications to the source code are still required in order to port the code from one plateform to another. This inevitably makes the retargeting process difficult and user-unfriendly.

In this paper, we present a prototype preprocessor that provides a *user level retargetable* engine for idioms recognition and code rewriting in innermost loops. Idioms recognition is the most specific part of the compilation process that is target dependent. It is thus best exposed to retargetability than vectorization and code generation which remain globally target independent.

The preprocessor is based on an intermediate form which was defined to be suitable for graph rewriting. Scalar operations as well as vector operations can be expressed in the intermediate form. Conditionals are represented using guards. The preprocessor performs source to source code transformation and can therefore be used as a preprocessing phase for actual code generation. The **CoSy** [2] compiler infrastructure has been used to build our prototype.

The remainder of this paper is organized as follows. In Section 2, we give a brief overview of our system. In section 3, we describe the intermediate representation form on which our system relies. The unparsing engine is then presented in Section 4. Section 5 is devoted to a description of the rewriting engine used towards the system. Finally, we present our preliminary results in Section 6 and conclude in Section 7.

2 System Overview

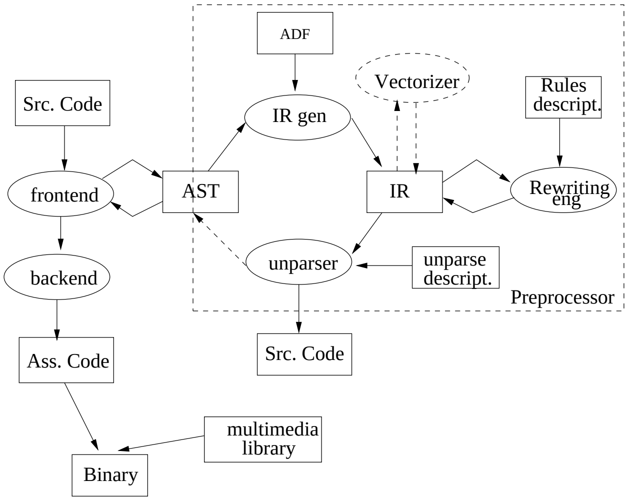

Figure 2: Engines of the system. The dashed lines indicate engines currently under development.

The preprocessor consists of three engines: the intermediate representation[1] generator, the unparser and the rewriting engine.

The purpose of the IR generator is to produce an IR form of the source code. This is done by traversing the AST[2] representation of the compiler's front-end. When traversing the AST, the IR generator constructs the IR graph by relying on an attribute declaration file (ADF) in which the attributes for nodes and edges are defined. These attributes are used to express sets of properties that nodes and edges of the IR may have. For instance, it is necessary to know if a node of the IR represents a scalar operator, a vector operator, a scalar or a constant. Thus, a node attribute that holds this set of values can be defined to this purpose. This extremly simplifies the retargeting process since labeling a node or an edge of the IR graph with a new attribute value only requires the user to add a new attribute or attribute value to the attributes description file. As the system focuses on innermost loops which are the main candidates for proposing such parallelism, only these loops are targeted during the construction of the IR graph.

The rewriting engine has the task of applying some local transformations on the produced IR graph. As in the case of the IR generator, the rewriting engine also relies on a description file in which are stated the rules that describe the idioms to be recognized and the transformations to be operated on the underlining IR graph.

The IR graph obtained from the rewriting engine can then be output as "C" statements by the unparsing engine in order to produce an optimized version of the source code or also be used to rebuild the AST representation of the compiler's front-end. The unparsing decision is based on a custom script language whose implementation is given in an unparser decsription file, which is user defined.

Figure 2 shows an overview of the system. As illustrated in the figure, several iterations between the rewriting engine and the IR graph are possible. This peculiarity enables the chaining of rewriting engines and will be further discussed in Section 5.3.

3 The Intermediate Representation

IR is the intermediate format used by the preprocessor for representing program instructions. As we focus on innermost loops, the intermediate form is not intended to describe general control flow instructions. The IR format is based on a graph structure that allows to represent the relations amongs program instructions. We use attribute declarations for defining nodes and edges of the IR graph.

This section is organized as follows. The first part provides a description of the attributes used for defining nodes and edges of the IR graph. In the last part of the section we state the global constraints that an IR graph must satisfy.

[1] Denoted IR in the remainder of the paper.

[2] Abstract Syntax Tree

kind → ∅ | [EDGES] | [NODES] | [*]
default → ∅ | DEFAULT value
elematt → kind name:typelist={fieldvalues}default;
type → ENUM|STRING|INTEGER|*
typelist → type|typelist,type
value → enumsymb|STRING
|INTEGER|BOTTOM|*|+
fieldvalues → value|fieldvalues,value
attributelist → elematt|attributelist elematt
attributesdecl → ATTRIBUTES attributelist

Figure 3: Attribute declaration language

3.1 Attribute Declaration

We use attribute declarations for annotating specific characteristics to nodes or edges of the IR graph. The attributes employed for describing nodes and edges are declared in a user-defined file, called ADF which stands for attribute declaration file in Figure 2.

The declaration of an attribute adheres to the grammar presented in Figure 3. A default value can be given for a node or an edge attribute following the keyword `DEFAULT`. If no default value is indicated, the default value is BOTTOM.

The grammar allows to easily declare new attributes or add new values to existing attributes. This is particularly useful as modifications in the ADF can be directly taken into account by the system, just after recompiling. Consider for example Figure 4 which shows an example of node and edge attributes declarations file. If one needs to define a new node characteristic, the following steps are required. First, the node attribute has to be declared in the ADF. If the attribute takes some values, these values should be defined and possibly the default value too. Declaring for instance a new node attribute that gives the scope of a variable within a procedure is done as follows :

[NODES] Scope : ENUM = {parameter,local,global} DEFAULT local;

In order to integrate this new attribute into the system it is only required to recompile it afterwards.

As illustrated in Figure 4, several attributes can be used for describing nodes and edges of the IR graph. Two particular attributes of nodes and edges are of significance for easing the rewriting process. The "type" edge attribute defines the nature of the relation that binds nodes of the IR graph. Several such relations have been defined, among which are the data flow edge (denoted flowdep) and the memory dependency edge (denoted memdep). The first denotes the flow of data in the program, whereas the second express the partial order constraints on memory accesses. The "LoopSector" node attribute specifies which part of the loop an expression belongs to. This attribute can take one of the following values *decl, prolog, epilog, init, test, incr, body* which correspond respectively to the declaration part, the loop prolog, the loop epilog, the loop initialization, the loop exit, the loop test, the loop increment and the loop body sections. Matching the whole loop or only a selected section of it is done by referring to this attribute. The unparsing engine relies on this attribute to rebuild the code of the loop.

A node or edge label is defined using a set of attributes. The grammar for describing labels is presented in Figure 5. Figure 5 also highlights the grammar for describing the graph structure, which is obtained by simply concatenating the edges. The same grammar is used for describing the program as well as the LHS and the RHS part of the rewriting rules.

A field value having {*} in this grammar indicates that the field can have any value including BOTTOM, + indicates that the field value can have any value except BOTTOM. To describe in the left hand side of rewriting rules that a match can happen only when no relation exists, a special attribute *negated* must precede an edge declaration. The *ident* value is used to index the node that is matched in the LHS and the RHS part of the rewriting rule. It is usually redundant when describing the program statements. The $*$ notation allows to factor edges. For instance, `(flowdep Content1:4 Plus:* 0)` indicates edges from node with *ident* value 4 and label name `content1` to all nodes with label name `Plus`, whatever the *ident* value. The INT token indicates the rank of an edge.

Declarations	**Comments**
$[NODES]\, NodeType : ENUM = \{operator, scalar, cst, ...\}$	Described the type of node
$[NODES]\, LoopSector : ENUM =$ $\{decl, prolog, epilog, init, test, incr, body\}\ DEFAULT\, body;$	Defines the loop section
$[EDGES]\, type : ENUM = \{flow, memdep, same, ...\};$	Attribute `flow` indicates a dataflow relationship. Attribute `memdep` indicates a data dependencies between memory accesses. Attribute `same` indicates that two references to symbol corresponds to the same symbol.

Figure 4: Example of node and edge attribute declarations

Edge declaration	node $\rightarrow$ NODELabel name : fieldlist edge $\rightarrow$ EDGELabel name : fieldlist edgeslist $\rightarrow$ | edgelist edge edgesdecl $\rightarrow$ EDGES edgeslist
Node declaration	fieldlist $\rightarrow$ field | fieldlist field field $\rightarrow$ name = values values $\rightarrow$ * | {fieldvalues} nodesdecl $\rightarrow$ NODES nodelist
Graph structure	kind $\rightarrow$ $\emptyset$ | [negated] ident $\rightarrow$ INTEGER | * | INTEGER.ident nodename $\rightarrow$ * | STRING edge $\rightarrow$kind(edgename nodename:ident nodename:ident INT) edgelist $\rightarrow$ edge | edgelist edge

Figure 5: Grammar used for declaring node/edge label and the graph structure

3.2 Intermediate Form Constraints

The rewriting engine and the IR generator must fulfill two properties in order to ensure the reconstruction of the produced code. These properties are the following:

Property 1. The IR graph restricted to the data flow edges is a forest of trees.

Property 2. The IR graph restricted to the data flow and memory dependency edges must be a DAG.

The first property aims at restraining the complexity of pattern matching algorithms on graphs to that on trees. This enables an easier implementation of pattern matching algorithms. The second property ensures that rewriting rules do not introduce cycles in the produced graph. If applying a rule breaks these properties, the rewriting rule is not applied.

4 The Unparser

The IR graph can contain arbitrary nodes, each node being able to define various types of attributes. To make the preprocessor fully configurable, two unparsers have been provided in the system. The first one unparses the IR graph in text so that it can be processed by another rewriting engine. This unparser is directly built from the node and edge attribute declarations. The other unparser aims at unparsing the IR graph back to "C" code. It is based on a very simple scripting language, each script being specified for all possible operators defined in the attribute declaration file.

5 Rewriting System

This section presents the core of our system, i.e the rewriting system. We first introduce some preprocessing computations that precede the rewriting phase. Then, we present a description of the rewriting rules that form the rewriting system. In the last part of this section we show how rewriting engines can be chained together to simplify the use of the rewriting rules.

5.1 Preprocessing Phases

Before the IR graph is processed by the rewriting engine, we first apply some code transformations on the program. These transformations essentially aim at ensuring high reliability of the next phase. They include:

1. search and normalization of loop induction variable,
2. recognition and elimination of the temporaries,
3. forward substitution of constant and temporary variables,
4. simplification and normalization of some algebraic expressions.

The IR graph could also be pre-processed by a vectorization engine as illustrated in Figure 2.

5.2 Rewriting System Description

The graph rewriting system is directly issued from [6]. In particular, two main characteristics of this work are of particular interest for us:

1. **negated edges**: This allows to express the absence of relations between objects that would inhibit the matching. A typical case is alias existence.
2. **stratification**: This allows to organized the rules in strata. In the rule description, if no strata is specified, then the rule is assumed to belong to all strata. In a given strata, rewriting rules are assumed to terminate (Formal criteria can be found in [6]).

lhs → edgelist
rhs → edgelist | ∅
lang → ”C” | “C++”
strata → [INTEGER]
code → textcode
rule → RULE strata name : lhs ⇒ rhs code
rulelist → rule | rulelist rule
ruledecl → RULES rulelist

Figure 6: Grammar for the rewriting rules.

	NODELabel Assign: $NodeType = \{operator\}$ $Operator = \{assg\}$ $ValueType = \{int\}$ $VariableName = \{*\}$ $LoopSector = \{body\}$	NODELabel v: $NodeType = \{scalar\}$ $Operator = \{obj\}$ $ValueType = \{int\}$ $VariableName = \{*\}$ $LoopSector = \{body\}$	NODELabel Content: $NodeType = \{scalar\}$ $Operator = \{ref\}$ $ValueType = \{int\}$ $VariableName = \{*\}$ $LoopSector = \{body\}$
Nodes declaration	NODELabel Plus: $NodeType = \{operator\}$ $Operator = \{add\}$ $ValueType = \{int\}$ $VariableName = \{*\}$ $LoopSector = \{body\}$	NODELabel Shift: $NodeType = \{operator\}$ $Operator = \{sright\}$ $ValueType = \{int\}$ $VariableName = \{*\}$ $LoopSector = \{body\}$	NODELabel IntCst: $NodeType = \{cst\}$ $Operator = \{val\}$ $ValueType = \{int\}$ $VariableName = \{*\}$ $LoopSector = \{body\}$ $Mode = \{read\}$ $CstValue = \{1\}$
Edges declaration		EDGELabel flowdep: $type = \{flow\}$ EDGELabel samedep: $type = \{same\}$ EDGELabel lexorder: $type = \{order\}$	
Rules	RULE [1] PlusToShift:	(samedep v:3.1 v:3.2 0) (samedep v:3.2 v:3.1 0) (flowdep Content:5.1 Plus:2 1) (flowdep Content:5.2 Plus:2 2) (flowdep Plus:2 Assign:1 2) (flowdep v:3.3 Assign:1 1) (flowdep v:3.1 Content:5.1 0) (flowdep v:3.2 Content:5.2 0) ⇒ (flowdep v:3.3 Assign:1 1) (flowdep Shift:4 Assign:1 2) (flowdep v:3.1 Shift:4 1) (flowdep IntCst:6 Shift:4 2)	

Figure 7: Rewriting system for $b = a + a$; into $b = a \ll 1$;

The grammar we use to describe the rewriting rules is presented in Figure 6. The edges and nodes in this grammar are described using the same formalism as for the nodes and edges of the IR graph (see Figure 5). But, contrary to the program description, in rule descriptions the value $*$ or $BOTTOM$ can be specified as attribute values. A segment of “C” or “C++” code can be appended to the rule description in order to implement a particular side effect on the IR graph[3].

[3]This part is however not yet implemented.

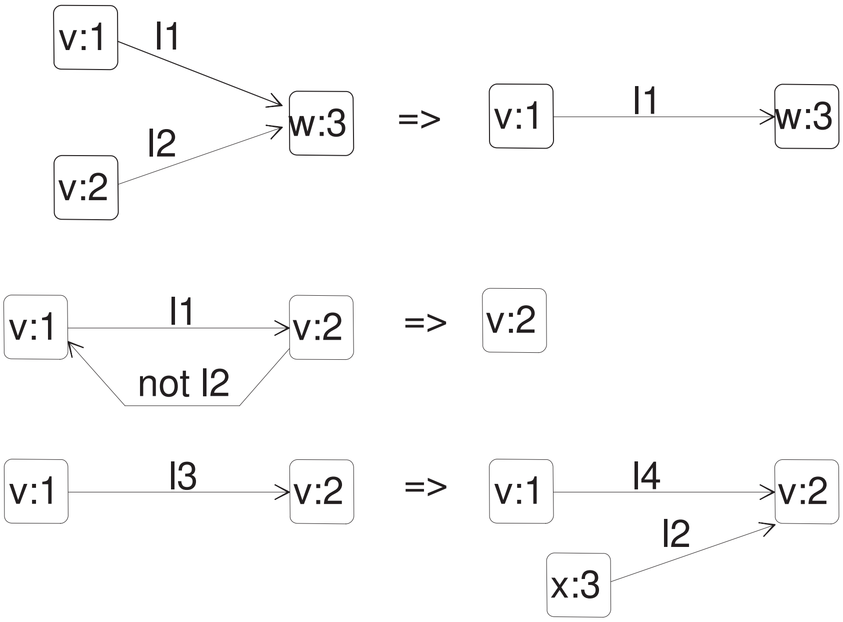

Figure 8: Example of simple graph rewriting systems

Figure 8 illustrates the elementary rewriting operations that can be expressed by the rules. The first rule in the figure shows the deletion of nodes and edges. The rule removes an l2-edge between two redex nodes connected by an l1- and an l2-edge. The second rule is applied only when an l1-edge is matched between two redex nodes and there is no l2-edge between these nodes. In the last rule, if an l3-edge is matched between two redex nodes, the edge is removed and replaced by an l4-edge. A new node x:3 is also allocated and an l2-edge is created between the new allocated node and the target redex node of the removed edge (v:2). Note that if an l2-edge already exists between the redex nodes x:3 and v:2, the application of the rule will only result in replacing the l3-edge by an l4-edge.

As an example, considers the simple "C" code $b = a + a$ which is illustrated in Figure 9. Now assumes we want to rewrite it to $b = a \ll 1$. The steps for doing this will include declaring the nodes and edges of the graph and stating the rewriting rules that will carry the transformations on the produced graph. We have illustrated these steps in Figure 7.

5.3 Chaining Rewriting Systems

Grouping rewrite rules to form specialized rewriting systems yields to the chaining of rewriting systems. This has two implications. It simplifies the writing of complicated rules and gives rise to several possibilities for using the engine by allowing several rewriting paths.

The first point enables to split complicated rewriting rules into small pieces dedicated to specific recognitions. An example of this is illustrated in the first example of the next section.

The second point yields a high degree of flexibility to the user because it enables the application programmer to apply multiple rewrite systems. This leads to as many optimization outcomes of the same source program as there are rewriting paths. For instance, if vector memory accesses are needed to recognize some special kernel, it is still possible to do that while allowing another path that deals with scalar accesses.

6 Preliminary Results

In the following, we will consider three examples of multimedia kernels that often occur in multimedia applications. For all the examples treated below, we assume the presence of a multimedia library that provides an implementation for each multimedia idiom that is been given to the system.

The first example treats the dot-product, which is a well-know case in the multimedia application domain. We have looked at two different occurrences of this form, one that involves array accesses and the other that relies on pointer arithmetics. The original code for both forms looks as follows:

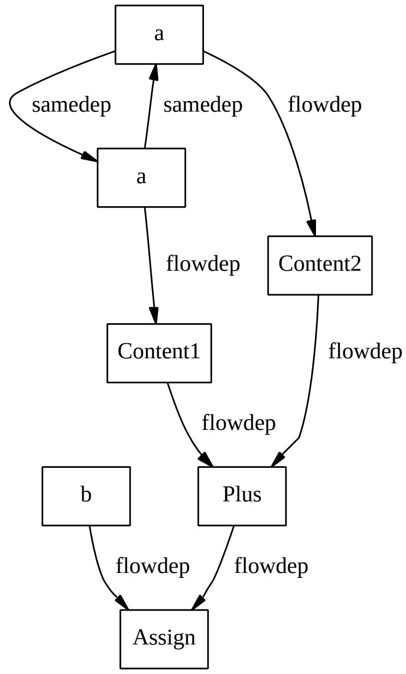

Figure 9: Graphical form of the “C” statement b=a+a.

```
int dotproduct(short a[],short b[],int N )
{
   int i,sum=0;
   for (i = 0; i < N; i ++)
      sum += (a[i] * b[i]);
   return sum;
}

int dotproduct(short *a,short *b,int N )
{
   int i,sum=0;
   for (i = 0; i < N; i ++)
      sum += ((*a)++ * (*b)++);
   return sum;
}
```

Prior to the rewriting phase, some pre-transformations as described in Section 5.1 are applied on both programs. As the above programs only differ syntactically but not semantically, there is no need to write distinct rewrite systems for the dot-product. Instead of this, as illustrated in the previous section, we first apply a specialized rewrite system on the IR graph obtained at the end of the pre-transformation phase. This pass aims at transforming every array or pointer access into a common expression that preserves the initial semantics. For a pointer or array access to a given type X, the common expression used in our system is a vector access to a memory location of base address A and displacement i, denoted Vector_X(A,i). The intermediate code obtained after applying this transformation is shown below, only the code portion corresponding to the innermost loops is highlighted:

```
for (i = 0; i < N; i = i + 1)
   sum = sum + (int)(Vector_short(a,i)) * (int)(Vector_short(b,i));
```

In a second pass, the normalized IR form is finally operated on by the rewrite system that is in charge of recognizing the dot-product occurrence. The produced code is shown below (it is the same for both programs).

```
int dotprod_decl=0;
for (i = 0; i < N; i = i + 4)
    dotprod_decl = dotprod_macro(dotprod_decl,a,b,i);
sum = call_epilog(dotprod_decl,a,b,N);
```

The *dotprod_macro* function is the call to the multimedia library interface that provides an optimized version of the dot-product. The call to the *call_epilog* function can be used to implement some side effects that are incident to the outcome of the dot-product implementation. The *dotprod_decl* is a scalar variable that is been introduced by the rewriting system.

We now consider the code shown in Figure 1-a. Optimizing this code on the Trimedia processor [18] can be done by relying on two specialized instructions: a *QUADAVG* instruction that is able to compute a vector addition, a scalar increment and a shift operation on four separate pairs of 8-bits data elements in parallel, and a *DSPUQUADADDUI* instruction which enables to carry a vector addition on four separate pairs of 8-bits data elements. The steps required for the transformations are the following. We first apply the pre-transformations and the normalization operations on the original code. We obtain the following code for the innermost loop:

```
for (i = 0; i < 64; i = i + 1)
    Vector_int(idest,i) = (int)( ((((int)(Vector_int(iback,i)) +
                                (int)(Vector_int(ifor,i))) + 1) >> 1)
                                + (int) (Vector_int(iidct,i)));
```

Next, we specify a multimedia idiom that corresponds to the statement of the innermost loop of the above intermediate code. This is done by conveniently stating a LHS part of a rewriting rule as described in Section 5. The rewriting rule is then completed by a RHS part which undertakes the substitution of the multimedia idiom resulting in the code shown in Figure 1-b.

Finally, we consider the example shown in Figure 10 which is taken from [9]. Although according to the authors it is extremely difficult to vectorize the given code, we are able to extract a form of parallelism from it. To do this, we rely on the pack and unpack operations and on the specialized Trimedia *DSPUQUADADDUI* instruction introduced previously. The resulting code is shown in Figure 11

```
void example(char *dest,char *end,char *src1,char *src2)
{
   int tmp;
   int addr;
   for (dest = addr; dest != end; dest += 4, src1 += 4, src2 += 4)
    {
     dest[0] = (src1[0] + src2[0]) >> 1);
     dest[1] = (src1[1] + src2[1]) >> 1);
     dest[2] = (src1[2] + src2[2]) >> 1);
     dest[3] = (src1[3] + src2[3]) >> 1);
    }
}
```

Figure 10: Example of multimedia kernel

```
for (dest = (long)addr;dest < end;dest = &dest[4],
     end = &end[4],src1 = &src1[4],src2 = &src2[4])
{
    tmp = DSPUQUADADDUI(PACK(src1,src1+1,src1+2,src1+3),
                        PACK(src2,src2+1,src2+2,src2+3));
   dest[0] = UNPACK(tmp,0) >> 1;
   dest[1] = UNPACK(tmp,1) >> 1;
   dest[2] = UNPACK(tmp,2) >> 1;
   dest[3] = UNPACK(tmp,3) >> 1;
}
```

Figure 11: Optimized version of the loop of Figure 10

7 Conclusion

In this paper we presented the design and the implementation of a preprocessor for multimedia instructions. We used idioms recognition techniques for detecting multimedia patterns hidden in source codes. To do this, we based our approach on a rule-based graph abstraction which also allowed us to provide users with a powerful mean for easing the retargeting process.

The state of the preprocessor implementation is still in progress. However, the preliminary results are promizing. We have shown three types of multimedia kernels that could be recognized by our preprocessor. The system can not only recognize specific multimedia kernels, but can also be used in a flexible manner by allowing the user to specify the patterns to find.

References

[1] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for high-performance computing. In *ACM Computing Surveys*, 26 - 4:345-420, December 1994.

[2] ACE Associated Compiler Experts bv. CCMIR Definition, Specification in SDL, Description and Rationale. Ref. CoSy-8002-ccmir, 1999

[3] Lee, Ruby. Subword Parallelism. In *IEEE Micro*, Volume 16, Number 4, pages 51-59, August 1996.

[4] Michael Forster, Andreas Pick and Marcus Raitner. The Graph Template Library (GTL) Manual. http://www.infosun.fmi.uni-passau.de/GTL/.

[5] Shlomit S. Pinter and Ron Y. Pinter. Program optimization and parallelization using idioms. In *ACM Transactions on Programming Languages and Systems*, 16 - 3:305 - 327, May 1994.

[6] Uwe Aßmann. Graph Rewrite Systems For Program Optimization. Technical Report RR-2955, INRIA Rocquencourt, 1996.

[7] N. Sreraman and R. Govindarajan. A Vectorizing Compiler for Multimedia Extensions. In *International Journal of Parallel Programming*, Volume 28, Issue 4 : 363-400, August 2000.

[8] Andreas Krall and Sylvain Lelait. Compilation Techniques for Multimedia Processors. In *International Journal of Parallel Programming*, Vol 28, No 4, 2000.

[9] Samuel Larsen and Saman Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation,* Vancouver, BC, June 2000.

[10] Bill Pottenger and Rudolf Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 444–448, 1995.

[11] R. Allen, and D. Kennedy. Automatic Transformations of FORTRAN Programs to Vector Form. In *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 4, October, 1987.

[12] P. Marwedel and G. Goossens, editors. Challenges In Code Generation for Embedded Processors. *Kluwer Academic Publishers*, 1995.

[13] Derek J. DeVries. A Vectorizing SUIF Compiler: Implementation and Performance. Master's thesis, University of Toronto, June 1997.

[14] 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.

[15] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan and Liang He. VIS Speeds New Media Processing. In *IEEE Micro*, Vol. 16, No 4, August 1996, pp. 10-20.

[16] Motorola Inc. AltiVec Technology Programming Environments Manual. November 1998.

[17] Paul Rubinfeld, Bob Rose and Michael McCallig. Motion Video Instruction Extension for Alpha. Digital Equipment Corporation. October 1996.

[18] TM1000 Preliminary Data Book. Philips Electronics, 1997.