CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2004; **16**:303–318 (DOI: 10.1002/cpe.776)

SWARP: A Retargetable Preprocessor for Multimedia Instructions

Gilles Pokam*,†, Stéphane Bihan, Julien Simonnet, and François Bodin

IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France

SUMMARY

In this paper, we propose SWARP, a retargetable preprocessor for exploiting multimedia instructions. The system mixes loop distribution, unrolling and pattern matching to exploit complex multimedia instructions. Contrary to all available systems, it can be extended at the user level. Using with a TriMedia processor we show that our system achieves important good code quality with a set of frequently used loop kernels for multimedia applications. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: Multimedia instructions, compiler optimizations, embedded processors

1. INTRODUCTION

Instruction set extensions [19, 10, 16] have been proposed to fulfill the requirements of computation-intensive multimedia applications. These extensions are based on SIMD instructions that operate on data subwords packed into registers [17]. This better exploits a processor's data path when operating on subwords. Multimedia applications typically use 8-bit (video) or 16-bit data (audio).

Exploiting these SIMD instructions, however, is a challenging issue for compilers. First, vectorizing techniques are needed to uncover the parallel operations to be gathered in a single SIMD instruction. Second, the instruction set imposes constraints on memory data alignment. Finally, many instructions implement complex operations dedicated to computation intensive kernels such as FIR filters. For instance, the TriMedia [9] instruction set offers an instruction capable of simultaneously computing four averages on bytes data. Most application writers directly insert by hand compiler known functions or in-line assembly routines. As a consequence, those applications are not portable to different platforms.

*Correspondence to: Gilles Pokam, IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France.
†E-mail: gilles.pokam@irisa.fr

Contract/grant sponsor: STMicroelectronics
Contract/grant sponsor: SMT Medea; contract/grant number: 199C9010031308011

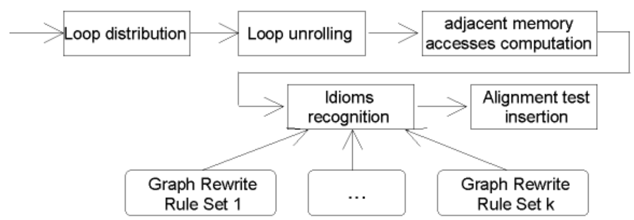


Figure 1. SWARP processing phases

For general purpose processors such as the Intel Pentium, developing an *ad hoc* vectorizing compiler tuned for a specific version of the processor's instruction set can be economically viable. However, such a solution is not acceptable for the fast moving domain of embedded systems.

In this paper, we propose SWARP, a new retargetable C preprocessor, to exploit multimedia instruction sets. This preprocessor can easily be extended by application developers by providing code rewrite rules. An original feature of SWARP is that these rules are directly described using C code patterns.

The remainder of the paper is organized as follows. Section 2 starts with a brief overview of SWARP followed by a description of its main processing phases: loop distribution, loop unrolling, idiom recognition and alignment test insertion. Section 3 addresses the rewriting system which underlies the idiom recognition phase. In this section, we first present a short background on the employed graph formalism. We then detail idiosyncrasies of the graph structure used to represent the rewrite rules and the C code expressions. Finally, we show how rules can be written and the system efficiently retargeted to a new ISA. For this particular purpose, a motivating example will be used to drive the demonstration on two different processors with multimedia extension capabilities. Section 4 illustrates the use of SWARP by means of some experiments. Related works are presented in Section 6 and Section 7 concludes.

2. OVERVIEW OF SWARP

SWARP is based on loop distribution, loop unrolling and pattern-matching to identify sets of expressions that can benefit from multimedia instructions. An adjacent memory computation phase computes the address stride between data accesses. The target-dependent part of the process is the idiom recognition phase. It is parameterized by a set of rewrite rules. The last phase of SWARP deals with data alignment. All these phases are illustrated in Figure 1. In the remainder of this section we describe each phase and illustrate them with a simple example shown in Figure 2.

2.1. Loop Distribution Phase

The core of the loop distribution phase is the Allen and Kennedy algorithm [2]. The algorithm's main purpose is to restructure the body of a loop in such a way that more parallelism can be extracted. Based on a data dependence graph, the statements of the loop are first gathered into strongly connected components. Then, a loop is created for each strongly connected component. This enables us to isolate statements of the loop that correspond to vector loops (loops with non-acyclic data dependence graph). This has two advantages. First, it simplifies the description of patterns corresponding to the multimedia instructions; and second, it reduces the complexity of the pattern-matching used in the idioms recognition phase. Before the loop distribution is performed, a forward substitution pass is realized in order to create larger expressions that may correspond to more complex multimedia instructions. Figure 2(b) exemplifies the results obtained for that phase.

2.2. Loop Unrolling Phase

Loop unrolling [4] is generally used as an ILP enhancing technique to expose more parallelism. It mainly consists in reducing the number of loop iterations by replicating the body of the loop for a given number of times. In SWARP, we use loop unrolling to discover multiple data that could be efficiently processed by a single SIMD multimedia instruction. To do this, instances of the same instruction belonging to successive iterations of the loop are unrolled. The unrolling factor is chosen according to the data types used in the loop and the size of the machine registers. For instance, if the target machine has 32-bit registers and the loop accesses character data then it is unrolled four times. An example that shows how the loop unrolling phase is undertaken in SWARP is presented Figure 2(e). This step is then completed by the adjacent memory access computation phase described in Section 2.3.

2.3. Adjacent Memory Access Computation Phase

This phase computes the memory stride between two accesses of array elements. An edge is added in the SWARP intermediate graph form to identify accesses to contiguous array elements. Figure 2(d), shows the adjacent edges, in bold, in the expression tree of the loop body. These dependencies are used to recognize sets of data that can be candidate for some multimedia instructions. If data elements are not contiguous in memory, they have to be loaded separately and then packed together into one register in order to be operand of a multimedia instruction. In a similar way, the result of the multimedia operation should be unpacked. Such a combination of instructions, SIMD plus pack and unpack operations, is costly and might cancel the benefits of using SIMD instructions. By expressing the fact that accessed data have to be adjacent in the rewrite rule, the system allows us to avoid this costly scenario. But it is still the rule writer's responsibility to conceive rules that effectively lead to performance improvement.

2.4. Idiom Recognition Phase

The idiom [15] recognition phase replaces sets of expressions by calls to multimedia instructions. An example is given in Figure 2(e). This part must be very flexible as it is specific to the target machine. The rewrite engine is based on [3]. Rewriting is not directly applied on the front-end's abstract syntax tree but on an extensible specific intermediate form (see Section 3.2) better suited to the rewrite process. Rewriting rules are expressed in the same form. Producing rewrite rules is performed using

(a)

```
for (i = 0 ; i < 100 ; i += 1) {
    a[i] = b[i] + c[i];
    sum = sum + a[i] * d[i];
}
```

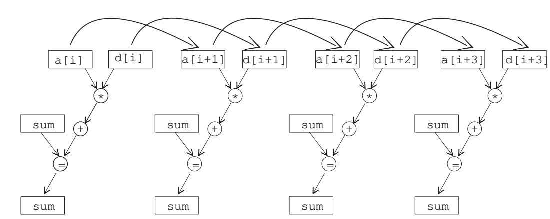
(b)

```
for (i = 0 ; i < 100 ; i += 1) {
    a[i] = b[i] + c[i];
}
for (i = 0 ; i < 100 ; i += 1) {
    sum = sum + a[i] * d[i];
}
```

(c)

```
for (i = 0 ; i < 100 ; i += 4) {
    a[i] = b[i] + c[i];
    a[i + 1] = b[i + 1] + c[i + 1];
    a[i + 2] = b[i + 2] + c[i + 2];
    a[i + 3] = b[i + 3] + c[i + 3];
}
for (i = 0 ; i < 100 ; i += 4) {
    sum = sum + a[i] * d[i];
    sum = sum + a[i + 1] * d[i + 1];
    sum = sum + a[i + 2] * d[i + 2];
    sum = sum + a[i + 3] * d[i + 3];
}
```

(d)



(e)

```
...

for (i = 0 ; i < 100 ; i += 4) {
        temp1_1 = IFIR16(packCont2i(a[i], a[i + 1]),
           packCont2i(d[i], d[i + 1]));
        temp2_1 = IFIR16(packCont2i(a[i+2], a[i + 3]),
           packCont2i(d[i + 2], d[i + 3]));
        temp3_1 = sum + temp1_1 + temp2_1;
        sum = temp3_1;
}
```

(f)

```
 ...
 if (is16bAligned(a[0]) && is16bAligned(d[0])) {
     for (i = 0 ; i < 100 ; i += 4) {
        temp1_1 = IFIR16(packCont2i(a[i], a[i + 1]),
           packCont2i(d[i], d[i + 1]));
        temp2_1 = IFIR16(packCont2i(a[i+2], a[i + 3]),
           packCont2i(d[i + 2], d[i + 3]));
        temp3_1 = sum + temp1_1 + temp2_1;
        sum = temp3_1;
     }
 }
```

Figure 2. SWARP processing phases a) original loop b) Loop after distribution c) Loop after unrolling d) Array access dependencies computation e) Pattern matching driven by the rule depicted Figure 5 f) Data alignment test insertion.

annotated C programs. This later feature allows the user to easily retarget the system to a new machine without changing SWARP.

2.5. Alignment Test Insertion Phase

The alignment phase inserts run time tests to ensure data items are word-aligned. This first version of SWARP generates two versions of each loop, the original and the transformed loop. Around the transformed loop a test is inserted for all data items that need to be aligned. If, at run-time, some of the data items are not word-aligned, the original loop is executed. This step may be improved by the loop distribution phase because it may also distribute the alignment constraints on the set of resulting loops. Figure 2(f) shows the example loop after the insertion of the word alignment test. More sophisticated techniques such as the ones proposed in [18, 11, 14] can be used.

3. REWRITE SYSTEM

This section presents the core of our rewrite engine. The terminology employed in the graph rewrite system is first introduced along with some basic notions. The graph structure which underlies the representation of rules and C code patterns is then closely looked into. Finally, the system of rule description is presented and illustrated with examples.

3.1. Background

The graph rewrite formalism is directly issued from [3]. In particular, we exploit two interesting properties of this graph rewrite system as follows.

- **Negated edges:** this allows us to express the non-existence of relations between objects, that would inhibit the matching. A typical example is aliasing or data dependencies.
- **Stratification:** this allows us to organize rules in strata, e.g. a list of rule sets applied in a specific order determined by the user. In the rule description, if no strata is specified, then the rule is assumed to belong to all strata. In a given strata, rewrite rules are assumed to terminate (formal criteria can be found in [3]). We have enhanced the stratification mechanism by allowing arbitrary chaining of rewrite engines. Multiple rewrite alternatives can therefore be tested.

A rewrite rule in our system consists of a left-hand side (LHS) and a right-hand side (RHS) part which are represented as annotated C code (see Section 3.3) and whose graphical representation is expressed by means of an extensible intermediate format (see Section 3.2). The RHS is not allowed to contain negated edges. When a pattern matches the LHS part of a rule restricted to the non-negated edges, the pattern may be replaced with the RHS part of the rule. The matched pattern is then called a *redex* and the nodes of the pattern designated as *redex* nodes. If not explicitly referenced in the RHS of a rule, the unaltered part of the corresponding LHS pattern is preserved by means of an invariant rule implicitly associated to each rewrite rule.

As an illustration of the graph rewrite system, let us look closely at Figure 3 which shows some of the elementary operations that can be expressed by a graph rewrite rule. The first rule in the figure shows the deletion of nodes and edges. It removes a v:2-node and a l2-edge between a redex node, v:3, which is connected to two other redex nodes, v:1 and v:2, by an l1- and an l2-edge respectively. The second rule is applied only when an l1-edge is matched between two redex nodes, v:1 and v:2, and there is no l2-edge between these nodes. In this case, the v:1-node and the edges are deleted.

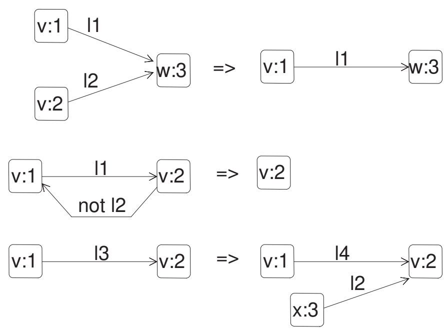


Figure 3. Example of a simple rewriting system

In the last rule, if an l3-edge is matched between two redex nodes, v:1 and v:2, the edge is removed and replaced by an l4-edge. A new node x:3 is allocated and an l2-edge is created between the new allocated node and the redex node incident to the removed edge (v:2). Note that if the l2-edge and the redex node x:3 already existed, the application of the rule would have only resulted in replacing the l3-edge by an l4-edge, preserving the unaltered part of the LHS from being modified.

3.2. Intermediate Representation

The intermediate representation (IR) form is intended to be an abstract representation of the program suitable for the recognition of idioms corresponding to multimedia instructions potential usages. It has been specifically engineered to represent innermost loops body and rewrite rules. It is not intended to represent arbitrary programs, therefore it can exhibit some characteristics that suit our purpose well. For instance, the IR form allows us to express scalar operations easily. Furthermore, associative and communicative operators are extended to n-array operators, when not specified in the code. This allows us to better perform recognition of operations such as multiply-add operations.

The IR form is based on a graph structure that describes the relations between the program instructions. To do this, nodes have associated attribute fields that characterize properties of a C expression we want to match. The current implementation of SWARP contains several definitions of node attributes. Among them are attributes describing operators, data types, variable names, loop section, etc. Specific attribute values are used to match any value ("*" wild card value) or to match the value of another node attribute. The relations between nodes can also be specified by means of several types of edges representing data flow relations, memory data dependences, same variable, adjacent memory and lexical ordering. The attribute formalism of the IR form is easily extensible to accommodate for new attribute types, allowing for fast retargeting. Figure 4 shows the IR form of part of a simple rule example - only the attribute fields of nodes are highlighted (see Section 3.3 for a description of the other keywords).

For the IR graph to ensure a safe reconstruction of the produced code, the rewrite engine should at least satisfy two properties. These properties are listed below.

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

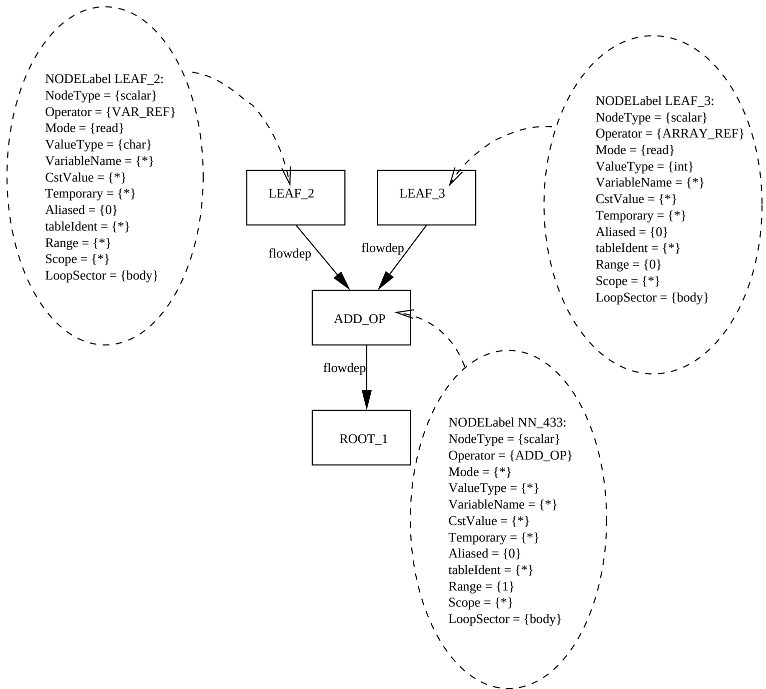


Figure 4. IR form of part of a rule

2. The IR graph restricted to the data flow and non-loop carried memory dependency edges must be a DAG.

The first property aims at restricting the complexity of the pattern matching algorithm on graphs to that on trees. This enables an easier implementation of the pattern matching algorithm. The second property ensures that rewrite rules do not introduce cycles in the produced graph. The application of a rewrite rule that produces a code that violates the constraints described above results in cancelling the application of the rule.

3.3. System of Rule Description

In this section, we put emphasis on the rule writing definition and on the matching mechanism of SWARP. We first detail how rules are written in C and how the system can be retargeted to a new ISA. Then, we show, through a few examples, how patterns can match complex C expressions, notably by combining rules together.

```
void ifir16Rule(){
    int i;
    int dest;
    short tabsrc1[LB], tabsrc2[LB];

    /* Left Hand Side */

    for(i=0;i<LB-1;i+=4){
       ROOT_1(LEAF_5(dest)=LEAF_9(dest)+LEAF_13(tabsrc1[i])*LEAF_17(tabsrc2[i]));
       ROOT_2(LEAF_6(dest)=LEAF_10(dest)+LEAF_14(tabsrc1[i+1])*LEAF_18(tabsrc2[i+1]));
       ROOT_3(LEAF_7(dest)=LEAF_11(dest)+LEAF_15(tabsrc1[i+2])*LEAF_19(tabsrc2[i+2]));
       ROOT_4(LEAF_8(dest)=LEAF_12(dest)+LEAF_16(tabsrc1[i+3])*LEAF_20(tabsrc2[i+3]));
    }

    /* Right Hand Side */

    for (i=0;i<LB-1;i+=4){
         NEWVAR_temp1=IFIR16(packCont2i(LEAF_13(tabsrc1[i]),LEAF_14(tabsrc1[i+1])),
               packCont2i(LEAF_17(tabsrc2[i]),LEAF_18(tabsrc2[i+1])));
         NEWVAR_temp2=IFIR16(packCont2i(LEAF_15(tabsrc1[i]),LEAF_16(tabsrc1[i+1])),
               packCont2i(LEAF_19(tabsrc2[i]),LEAF_20(tabsrc2[i+1])));
         NEWVAR_temp3=LEAF_9(dest)+NEWVAR_temp1+NEWVAR_temp2;
         ROOT_1(LEAF_8(dest)=NEWVAR_temp3);
    }
}
```

Figure 5. A TriMedia ISA rule example.

3.3.1. Rule definition

Because the same graph representation is used for describing both the input loop expressions and the rule patterns, we have extended SWARP to parse annotated C rule files. The C annotations are a convenient way to set the values of the rule node attributes. Describing the graph representation of the rule along with setting the node attribute values by hand would have been a tedious task otherwise.

We use function call-like notations to set the values of node attributes. This is done in such a way that the generated pattern can abstract among several different C expressions. The function call-like notations also aim at referencing nodes in both the LHS and the RHS of a rule. For example, in Figure 5, `LEAF_14(tabsrc1[i+1])`, which is referenced in both LHS and RHS, refers to the same node pattern. Some of the main notations in use are listed below.

- **ROOT_Y:** this meta node marks the beginning of a pattern definition in the LHS and the RHS.
- **LEAF_X:** this meta node abstracts the expression it encloses. The generated pattern node has the same abstracted properties as the enclosed expression. In the example of Figure 5, the node in LHS corresponding to `LEAF_14(tabsrc1[i+1])` matches any array access with the same type of element as `tabsrc1`. Relations between leaves are also generated in the graph. For instance, an edge that indicates that `LEAF_13(tabsrc1[i])` and `LEAF_14(tabsrc1[i+1])` are adjacent memory accesses is added. Some variants of `LEAF_X` exist to specify different degrees of abstraction (any variable or array element, any expression type, etc.).

- **NEWVAR_ident:** this meta node specifies the creation of a new variable in the right hand side of a rule. For each match, a new variable is created.

The definition of rules can also make use of macros to identify specific expressions, or to perform operations in the RHS. Some of them are system known and allow, for instance, the detection of alignment constraints.

A first set of macros, called *soft macros*, includes all the macros that define C expressions, like `saturatedShortOp()`, `packCont2i()` or `is16bAligned()`. For instance, `packCont2i()` packs two contiguous and 16-bit aligned data into one 32-bit register. Because elements are contiguous, the macro uses a C cast operator to convert a 16-bit type into a 32-bit type. The occurrence of this macro in the rewritten loop is later detected to insert alignment constraints that condition the execution of the transformed loop. An example using such macros in a combination of two rules is detailed in Section 4.1.

A second set of macros, called *hard macros*, implements processor intrinsics, for instance, bit-shift registers or multiply-accumulate sets of data. Intrinsics are usually provided with the processor ISA to enable the use of assembly instructions in C codes.

3.3.2. New ISA retargeting process

Retargeting SWARP mainly consists of writing the RHS part of pre-defined rules contained in a library of LHS definitions, to take into account the target machine intrinsics. For example, Figures 5 and 6 show the same rule targeted to two ISA, the TriMedia ISA and the VIS SPARC ISA extension. Both rules have the same LHS that recognizes four consecutive multiplication-accumulation operations on 16-bit contiguous array elements, usually found in FIR kernels. The RHSs differ by the call to the `IFIR16()` intrinsic in the TriMedia rule and by calls to the `vis_fmul()` intrinsics in the SPARC VIS rule to both perform the same operation.

Therefore, the cost related to the system retargeting process consits of adapting the RHS part of rules that involve target processor instructions. The number of such rules approximately corresponds to the number of specific processor instructions. Other rules are dedicated to the recognizing of saturated operations, etc. As the system allows the combination of rules, the less complex they become and the more effective is their reusability in exploiting different SIMD.

3.3.3. Matching mechanism

Pattern matching is not only performed on the IR graph restricted to the data flow edges of the patterns, but also on the values of the node attributes, twhich serve in abstracting expressions, and on the values of other edge attributes, which take into account other types of node relationships. All these three matching factors allow us to match different C expressions from a unique LHS pattern as represented in Figure 7.

In this example, the pattern still defines four consecutive multiplication-accumulation operations on four contiguous array elements. All the three cases (a)-(c) respect the lexical order of the multiplication, the addition and the result assignment. The `LEAF`s, operands of the multiplication operation, abstract array elements that can be of one or two dimensions. Data flow edges matching is done, first between the variables on the left and right side of the assignment that have to be the same in each of the four expressions, and second between the consecutive array accesses that have to be contiguous. Note that expressions in cases (a) and (b) do not match as they are, they have first to be replicated four times in a loop unrolling phase. Finally, by using a variant of `LEAF` that abstracts `dest` to match either a variable or an array element, the pattern would also match the expression in case (d).

```
void ifir16Rule(){
    int i;
    int dest;
    short tabsrc1[LB], tabsrc2[LB];

    /* Left Hand Side */

    for(i=0;i<LB-1;i+=4){
       ROOT_1(LEAF_5(dest)=LEAF_9(dest)+LEAF_13(tabsrc1[i])*LEAF_17(tabsrc2[i]));
       ROOT_2(LEAF_6(dest)=LEAF_10(dest)+LEAF_14(tabsrc1[i+1])*LEAF_18(tabsrc2[i+1]));
       ROOT_3(LEAF_7(dest)=LEAF_11(dest)+LEAF_15(tabsrc1[i+2])*LEAF_19(tabsrc2[i+2]));
       ROOT_4(LEAF_8(dest)=LEAF_12(dest)+LEAF_16(tabsrc1[i+3])*LEAF_20(tabsrc2[i+3]));
    }


    /* Right Hand Side */

    for (i=0;i<LB-1;i+=4){
         NEWVAR_tuh=vis_fmuld8sux16(packCont2i(LEAF_13(tabsrc1[i]),LEAF_14(tabsrc1[i+1]),
               packCont2i(LEAF_17(tabsrc2[i]),LEAF_18(tabsrc2[i+1])));
         NEWVAR_tlh=vis_fmuld8ulx16(packCont2i(LEAF_13(tabsrc1[i]),LEAF_14(tabsrc1[i+1]),
               packCont2i(LEAF_17(tabsrc2[i]),LEAF_18(tabsrc2[i+1])));
         NEWVAR_tul=vis_fmuld8sux16(packCont2i(LEAF_13(tabsrc1[i+2]),LEAF_14(tabsrc1[i+3]),
               packCont2i(LEAF_17(tabsrc2[i+2]),LEAF_18(tabsrc2[i+3])));
         NEWVAR_tll=vis_fmuld8ulx16(packCont2i(LEAF_13(tabsrc1[i+2]),LEAF_14(tabsrc1[i+3]),
               packCont2i(LEAF_17(tabsrc2[i+2]),LEAF_18(tabsrc2[i+3])));
         NEWVAR_tdh=vis_fpadd32(NEWVAR_tuh, NEWVAR_tlh);
         NEWVAR_tdl=vis_fpadd32(NEWVAR_tul, NEWVAR_tll);
         NEWVAR_td=vis_fpadd32(NEWVAR_tdh, NEWVAR_tdl);
         NEWVAR_temp1=vis_fpadd32(NEWVAR_temp1, NEWVAR_td);
         ROOT_1(LEAF_8(dest)=NEWVAR_temp1);
    }
}
```

Figure 6. A SPARC VIS ISA rule example.

In contrast, there can be several LHS rule definitions for the same RHS pattern. This permits the detection of syntactically different but semantically equivalent expressions. Furthermore, LHS rule patterns should be as simple as possible to allow one, by combining the rules, to match more complex patterns. For instance, rules that detect saturation test expressions should be the first of a sequence of rules that deal with saturated arithmetics, as explained in Section 4.1.

4. EXPERIMENTATIONS

We have validated SWARP with the TriMedia multimedia instruction set. This target is interesting because it features complex multimedia instructions (for instance an operation that computes four separate averages of the four pairs of bytes). SWARP has been tested on a set of multimedia kernels (dot product, block move, FIR filter, auto-correlation, etc) to recognize the occurrences of frequently used code patterns. These kernels represent the most time consuming part of many embedded applications. In some cases they have been partially rewritten to clean pointer arithmetics (changed to array

```
/* Left Hand Side */

for(i=0;i<LB-1;i+=4){
   ROOT_1(LEAF_5(dest)=LEAF_9(dest)+LEAF_13(tabsrc1[i])*LEAF_17(tabsrc2[i]));
   ROOT_2(LEAF_6(dest)=LEAF_10(dest)+LEAF_14(tabsrc1[i+1])*LEAF_18(tabsrc2[i+1]));
   ROOT_3(LEAF_7(dest)=LEAF_11(dest)+LEAF_15(tabsrc1[i+2])*LEAF_19(tabsrc2[i+2]));
   ROOT_4(LEAF_8(dest)=LEAF_12(dest)+LEAF_16(tabsrc1[i+3])*LEAF_20(tabsrc2[i+3]));
}
```

(a)

```
for(i=0;i<10;i+=1){
   for(j=0;i<100;j+=1)
      a = a + B[i][j] * C[i][j];
}
```

(b)

```
for(i=0;i<100;j+=1)
     a = a + B[i] * C[i];
```

(c)

```
...
   a = a + B[1] * C[1];
   a = a + B[2] * C[2];
   a = a + B[3] * C[3];
   a = a + B[4] * C[4];
...
```

(d)

```
for(i=0;i<10;i+=1){
   for(j=0;i<100;j+=1)
      A[j] = A[j] + B[i][j] * C[i][j];
}
```

Figure 7. A LHS pattern with some possible matching expressions

accesses) and annotations (for instance, restricted pointers) have also been introduced in some cases to specify that no aliasing exists between arrays. The first difficulty we have addressed, with SWARP, is saturated arithmetics. This is described in Section 4.1. The second issue, data alignment, is described in Section 4.2. Finally, in Section 5, we give performance results on the Philips TriMedia TM1000 [9].

4.1. Dealing with Saturated Arithmetics

Figure 8 shows a common way for users to express saturation. These tests impact a lot on the performance of the kernel. One benefit of the multimedia instructions on the TriMedia is to provide saturated arithmetics. The detection and code transformations to remove the saturation test in SWARP is split in two rules: the first rule detects the saturation expression, the second rule propagates the saturation to operators. After loop unrolling, the preprocessor can then replace the `saturatedShortOp(tab1[i]+tab2[i])` expression by the appropriate DSP operation.

(a)

```
short tab1[UB];
short tab2[UB];
short dest[UB];
int temp;

for (i = 0; i < UB; i = i + 1) {
    temp = tab1[i] + tab2[i];
    dest[i] = temp > 32767 ? 32767 : temp < -32768 ? -32768 : temp;
}
```

(b)

```
for (i = 0 ; i < UB ; i = i + 1) {
    temp = saturatedShortOp(tab1[i] + tab2[i]);
    dest[i] = saturatedFilterShort(temp);
}
```

(c)

```
for (i = 0 ; i < UB ; i = i + 2) {
    TT1 = DSPIDUALADD(packCont2i(tab1[i], tab1[i + 1]),
       packCont2i(tab2[i], tab2[i + 1]));
    blockCopyCont2i(dest[i], dest[i + 1], TT1);
}
```

Figure 8. Frequent implementation of saturation. The same code after saturation test detection and after use of the multimedia DSPUADD instruction.

4.2. Data Alignment Issue

Data alignment is a difficult issue for many kernels as shown in Figure 9(a). In this kernel, accesses to the `x` array are shifted by one character at each iteration of the outer loop. As a consequence, an optimized version can only be exploited every four iterations of the `j` loop, because of the alignment constraints due to the array element loads. A hand-rewritten version of this kernel used four copies of array `x` in such a way that an aligned version of array `x` is available for every iteration of the `j` loop.

5. PERFORMANCE

The experiments were conducted on a set of multimedia loop kernels including the square dot product, the dot product, the auto-correlation, the saturated addition, FIR8, FIR4, the block move kernel and the average kernel. All these applications were compiled using the TriMedia compiler *tmcc* with the highest level of optimization enabled. Two data sizes were chosen. The first one is small enough so that the data fit in the cache (benchmarks suffixed with "*Medium*" in the figure). The second data size is large enough to highlight the impact of data cache misses on the performance of the loop kernels (benchmarks suffixed with "*Large*" in the figure). The objectives of the experiences were twofold.

First, we wanted to assess SWARP capabilities of automatically recognizing critical code patterns on real code examples. This objective was achieved as most code patterns were successfully recognized with a reasonable number of rules (about 20 rules were sufficient on average). Second, real potential speedup could then be measured on the transformed code, ensuring the quality and effectiveness of the tool.

The results of the experiments are given in Figure 9. Because of space concerns, only results for the FIR8, the square dot product and the average kernel are presented. In the figure, the x-axis represents the original code as is (labeled "*Original*" in the figure), the code with pointer arithmetics cleaned (labeled "*Cleaned*" in the figure), and finally the code as it has been produced by SWARP (labeled "*Optimized*" in the figure). The y-axis represents the execution time given in cycles.

As shown in the figure, the speedup obtained after the code has been processed by SWARP always yields good performance improvements. On average, all the applications have seen their total execution time improve by a factor of 2.04 when the chosen data size was medium and by a factor of 1.82 in the case corresponding to a large data size. When data alignment issue is taken into account, the improvement factor increases up to 3.33 for the FIR8 kernel (see Figure 9). An interesting benchmark in the figure is the average kernel because it mixes the main two features addressed by a wide range of multimedia instructions: saturated arithmetics and complex instructions. The resulting transformed program behaves well in both cases of large and medium data sizes compared with the original program. Sometimes however, pointer arithmetics can seriously complexify the pattern matching process, the main concern being aliasing. For example, this is the case for the square dot product kernel, Figure 9(b), which heavily makes use of this feature. To circumvent it, the code was first annotated beforehand ("Cleaned" on the x-axis) to specify that no aliasing exists between pointers. As it can be seen from the figure, the performance improvement on the medium sized code is marginal, a case that should warn us to apply the transformation there. In contrast, in the large-sized code, the performance improvements are satisfactory.

6. RELATED WORKS

Various works have been addressing the issue of exploiting multimedia instructions. However, very few of them have tackled, in a systematic way, the issue of the retargetability and extensibility at the user level. In this section, we review the main approaches used so far and discuss the differences and/or resemblances with our work.

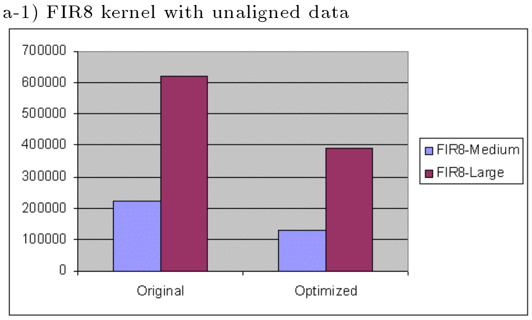
The natural approach to address the issue of multimedia instructions is to exploit vectorizing technology as a basis for discovering SIMD parallelism [18, 11, 5, 8, 7]. An alignment analysis phase is added to take this issue into account. In these works, the idiom recognition phase is *ad hoc* and hard coded in the compiler. This is a realistic approach for superscalar processors. However, this is not acceptable for media processors where the instruction set can rapidly be tuned from one set of applications to another. Our work completely removes this constraint. The user is provided with a powerful tool for fast retargeting by means of a pre-defined library of rule descriptions which is easily extensible.

Larsen *et al.* [12] propose a method based on unrolling at the source code level. They first unroll the loop and perform an alignment analysis and an adjacent memory reference analysis. Based on these analyses, the packing of instructions is performed to instantiate multimedia instructions. This work, however, does not address the code generation phase which is specific to each target machine. The method defined by Leupers [13] is applied at code selection time. It formulates the problem as an integer linear problem. The method relies on unrolling to expose multiple instances of an operation.

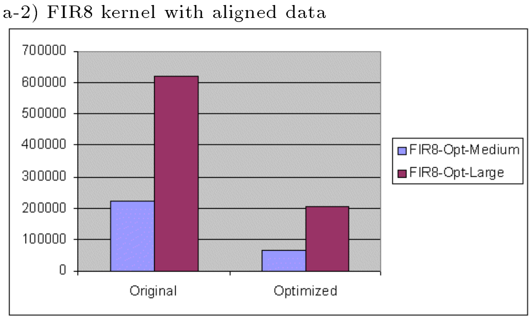
a) FIR8 kernel

```
void fir8(char x[], char h[], char y[], int N, int M) {
 int i, j, sum;
 for (j = 0; j < M; j++) {
  sum = 0;
  for (i = 0; i < N; i++)
   sum += x[i + j] * h[i];
   y[j] = sum >> 15;
}
```

a-1) FIR8 kernel with unaligned data

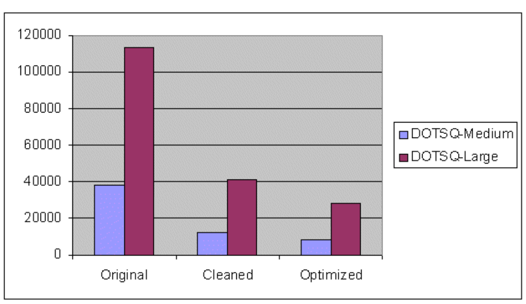


a-2) FIR8 kernel with aligned data



b) Square dot product kernel

```
tmpPtr2 = wiPtr;
for (endPtr2=tmpPtr2 + S_LEN; tmpPtr2<endPtr2;
tmpPtr2++){
 *R += *tmpPtr * *tmpPtr2;
 G += *tmpPtr * *tmpPtr;
 tmpPtr++;
}
```



c) Average kernel

```
for (i = 0 ; i < UB ; i = i + 1) {
 temp = ((back[i] + forward[i]+1) >> 1)+ idct[i];
 dest[i]= temp > 256 ? 256 : temp < 0 ? 0 : temp;
}
```

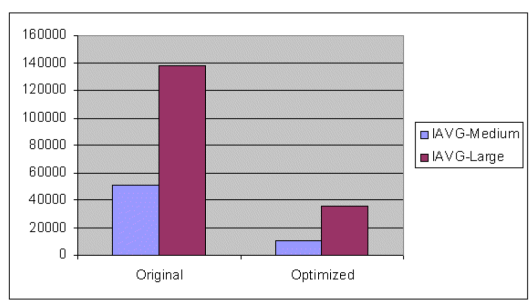


Figure 9. Performance results

Data alignment is not taken into account, however, as opposed to our approach. The method is also retargetable as new patterns can be added to the code generator (based on a modification of [1]).

The approaches in [14, 6] bear some similarity with our work. A C-based source-to-source transformation engine is proposed which relies on three processing stages: a pattern matching stage, a condition checking stage, and a result stage. However, there are some differences compared with our work that are relevant here. First, in their work, the transformations which aim at extracting the parallelism must be embodied in the result stage which defines the actions to be taken on the matched pattern. In ours, this is completely separated from the rule description. We believe this is a crucial difference since separating the parallelism extraction phase from the idiom recognition phase, viewed from a system perspective, is less constraining. Furthermore, our approach has the advantage of preventing the user from dealing with transformations which may be difficult to implement. Another difference comes from the employed matching mechanism. In our case, the matching is applied on abstract expressions and the relations between them (abstracted through edge attributes). In their case, the matching mechanism is tied to syntactic-equivalent statements. For instance, as a result, whereas in our approach only one LHS pattern is used to recognize several different C expressions as shown in Section 3.3.3, in their approach, to our understanding, multiple LHS patterns for each different C expression would be required instead.

7. CONCLUSION

In this paper we present SWARP, a retargetable preprocessor for exploiting multimedia instructions. It mixes loop distribution, unrolling and pattern matching to achieve the exploitation of complex multimedia instructions. Contrary to all available systems, it can be extended at the user level. This allows us to take into account new instructions or coding practices used by application programmers. Still, the efficiency of SWARP is very dependent on the coding style, aliasing and data alignment properties. The system allows us to partially deal with saturated arithmetics that most systems simply ignore. These features make SWARP better suited for media processors than currently available systems used with superscalar processors.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for providing useful comments and suggestions on this paper.

REFERENCES

1. Aho, A.V., Ganapathi, M., and Tijang, S.W.K. Code Generation Using Tree Matching and Dynamic Programming. *ACM Transaction on Programming Languages and Systems*, 11(4):491–516, 1989.
2. Allen, R., and Kennedy, D. Automatic Transformations of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.
3. Assmann, U. Graph Rewrite Systems For Program Optimization. Technical Report RR-2955, INRIA Rocquencourt, 1996.
4. Bacon, D.F., Graham, S.L., and Sharp, O.J. Compiler Transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.
5. Bik, A., Girkar, M., Grey, P., and Tian, X. Experiments with Automatic Vectorization for the Pentium 4 Processors. In *Proceedings of the Ninth International Workshop on Compilers for Parallel Computers (CPC2001)*, 2001.
6. Boekhold, M., Karkowski, I., and Corporaal, H. Transforming and Parallelizing ANSI C Programs Using Pattern Recognition. In *Proceedings of the HPCN Europe'99*, Amsterdam, NL, April 1999.
7. Cheong, G., and Lam, M.S. An Optimizer for Multimedia Instruction Sets. In *Proceedings of the Second SUIF Compiler Workshop*, Stanford University, August 1997.
8. DeVries, D.J. A Vectorizing SUIF Compiler: Implementation and Performance. Master's thesis, University of Toronto, June 1997.
9. Philips Electronics. *TM1000 Preliminary Data Book*, 1997.
10. Motorola Inc. *AltiVec Technology Programming Environments Manual*, November 1998.
11. Krall, A., and Lelait, S. Compilation Techniques for Multimedia Processors. *International Journal of Parallel Programming*, 28(4):347–361, August 2000.
12. Larsen, S., and Amarasinghe, S. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, BC, June 2000.
13. Leupers, R. Code Selection for Media Processor with SIMD Instructions. In *Proceedings of DATE*, 2000.
14. Manniesing, R., Karkowski, I., and Corporaal, H. Automatic SIMD Parallelization of Embedded Applications Based on Pattern Recognition. In *Proceedings of the 6th International Euro-Par Conference*, Munich, 2000.
15. Pottenger, B., and Eigenmann, R. Idiom Recognition in the Polaris Parallelizing Compiler. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 444–448, 1995.
16. Rubinfeld, P., Rose B., and McCallig, M. *Motion Video Instruction Extension for Alpha*. Digital Equipment Corporation, October 1996.
17. Ruby, L. Subword Parallelism. *IEEE Micro*, 16(4):51–59, August 1996.
18. Sreraman, N., and Govindarajan, R. A Vectorizing Compiler for Multimedia Extensions. *International Journal of Parallel Programming*, 28(4):363–400, August 2000.
19. Tremblay, M., O'Connor, M.J., Narayanan, V., and He, L. VIS Speeds New Media Processing. *IEEE Micro*, 16(4):10–20, August 1996.

