Scalar and expression abstraction by function encapsulation

M.Palkovic E.Brockmeyer F.Catthoor*
IMEC Lab., Kapeldreef 75, 3001 Leuven, Belgium.
{palkovic,brockmey,catthoor}@imec.be

Abstract

Modern multimedia applications usually combine well-known, well-optimised kernels. However, the way these are typically mapped on a target architecture results in large inter-kernel buffers. During this process, the kernel functionality is usually inlined into the high-level code that expresses the relation among the kernels. Thus many kernel details make the automated optimisation of interesting inter-kernel buffers mostly infeasible.

In this paper we propose an automated technique to separate and encapsulate the kernel code into functions. This technique allows to raise the level of abstraction and preprocesses the application for inter-kernel buffer optimisation. The technique has been implemented and tested on real-life examples like part of digital audio broadcast application or video conference codec.

1 Introduction

Recent multimedia systems should be implemented as low-power portable embedded systems. This wish is in contradiction with the huge amount of data transfer and large data memories consuming a major part of the energy in the embedded system. To reduce the energy consumption and to optimise global memory accesses, the Data Transfer and Storage Exploration (DTSE) methodology has been developed at IMEC [1].

Before optimising the memory accesses by the DTSE methodology the application has to be hierarchically rewritten [1]. This has two main reasons: to abstract the memory related part and to hide details related to the data-processing related part. During the hierarchical rewriting the application is split into three layers. Layer 1 expresses the relation among different processes (process flow), Layer 2 is related to the memory part (loop hierarchy and indexed signals) and Layer 3 is related to the data-path (arithmetic, logic and data-dependent operations). Usually, the Layer 1 is already well separated in the initial code. However, Layer 2 and Layer 3 are intermingled together. To perform the split for these two layers manually is error-prone and tedious operation.

The hierarchical rewriting is crucial mainly in the loop transformation stage of the methodology [2, 3]. The models used in this stage rely on this rewriting. If not done or not done well, the models either cannot be extracted or the automated optimisations cannot be performed because many details are blocking this phase. This is true especially when handling large and complex real-life applications.

This paper proposes an automated rewriting strategy that is meant as the front-end of the loop transformation optimising tool. Till now, this rewriting (especially the split of Layer 2 and Layer 3) was performed manually. This is an error-prone and tedious task for large and complex real-life applications.

2 Function encapsulation technique

The automated technique proposed in this paper uses the scalar data flow analysis in ATOMIUM [4]. The technique first identifies all array writes in the given function of the input code. This is done by traversing the whole ATOMIUM Abstract Syntax Tree (AST) and collecting all the array writes. Then the technique finds

*Also Professor at Katholieke Univ. Leuven.

automatically all array reads where the given array write is flow dependent on. All the functionality between the reads and flow depending write is encapsulated into the function. The above description provides quite high-level view on the technique and it will be refined in the sequel.

Before we describe the technique in details some basic terms have to be defined. Simplified, in the ATOMIUM scalar dependency analysis the definition corresponds to a write access and the use corresponds to a read access. For both the write and read accesses, we can find back the statement they belong to. We can also collect all uses (read accesses) of the given expression. Note that a statement is also an expression. Further, we can backtrack the flow dependency, i.e. data dependency between a definition and any use of the same variable. Note that the variable has to be defined before it can be used. To backtrack the flow dependency is the same as to find for the given use (read) the corresponding definition (write).

```
for(...) {
  for(...) {
   tmp_r0=fft_Re1[];
   tmp_i0=fft_Im1[];
   ...
   t0i=(tmp_i0+tmp_i2);
S1:t1i=(tmp_i1+tmp_i3);

   fft_Re3[]=(t0r+t1r)>>1;
S3:fft_Im3[]=(t0i+t1i)>>1;
  }
}
```

Figure 1: Original DAB FFT code

```
for(...) {
  for(...) {
   fft_Re3[]=lt_func_2(
                fft_Re1[],...);
S3:fft_Im3[]=lt_func_3(
                fft_Im1[],...);
  }
}
```

```
int lt_func_3(...)
{
 int ...;

 tmp_i3=lt_var_9;
 tmp_i1=lt_var_8;
 t1i=tmp_i1+tmp_i3;
 tmp_i2=lt_var_7;
 tmp_i0=lt_var_6;
 t0i=tmp_i0+tmp_i2;
 return t0i+t1i>>1;
}
```

Figure 2: Transformed DAB FFT code (main and new func)

We demonstrate these basic concepts on an example from the Digital Audio Broadcast (DAB) [1] Fast Fourier Transformation (FFT) (see Figure 1). Let us take the last statement *S3: fft_Im3[]=(t0i+t1i)>>1;*. The statement contains one definition *fft_Im3[]* and two uses *t0i* and *t1i*. If we ask for the statement the definition *fft_Im3[]* belongs to we obtain the original statement *S3: fft_Im3[]=(t0i+t1i)>>1;*. If we decide to collect all uses of the given statement, we obtain two uses *t0i* and *t1i*. If we backtrack the flow dependency of the use *t1i* we obtain the definition of *t1i* in the statement *S1*.

2.1 Scalar copy propagation

The technique is quite simple and boils down to simple copy propagation if we do not cross any control-flow boundaries. A good example is provided in Figure 1. First we have to identify the array writes, i.e. *fft_Im3[]* and *fft_Re3[]*. For each write we ask for the statement this write belongs to (e.g. *S3: fft_Im3[]=(t0i+t1i)>>1;* for *fft_Im3[]*). We copy this statement into a newly created function. For this statement we collect all the uses, i.e. *t0i* and *t1i*. For each of the uses we backtrack the flow dependency and obtain the corresponding definition. For this definition we ask for the statement this definition belongs to and so on. At the end we obtain new code with a higher abstraction level and corresponding functions which encapsulate all low-level details (see Figure 2).

2.2 Innermost condition encapsulation

Every innermost condition detected during backtracking is automatically rewritten to the ternary operator. The operator is then encapsulated into the particular function. We will demonstrate this concept on the example from the Quadtree Structured Difference Pulse Code Modulation (QSDPCM) application [1] (see Figure 3). In the statement *S4* the use of variable *p2* is detected. If we ask for its definition we cross the control-flow boundary. We obtain two potential definitions depending on the *ctrl* control expression. If one potential definition (i.e. one branch of the condition) is missing we add this branch. In the added branch we construct an identity statement (e.g. *p2=p2;*). The full control structure (both branches) is then automatically rewritten to the ternary operator (see statement *S1'* in Figure 4). Finally, the flow dependencies for all the uses in *ctrl* expression and both assignment statements in the branches are backtracked further.

```
for(...) {
  ...
    for(...) {
      p1=sub4_frame[];
      if(ctrl)
        p2=0;
      else
        p2=prev_sub4_frame[];
S4:   dist=dist+abs(p1-p2);
    }
S5: tmp_v4x=f(vx,dist);
  ...
}
```

Figure 3: Original QSDPCM code

```
for(...) {
  ...
    for(...) {
      dist=lt_func_2(
          x4, y4, dist,
          sub4_frame[],...);
    }
  tmp_v4x=lt_func_3(dist,...);
  ...
}
```

```
int lt_func_2(...)
{
  int ...;

  lt_el_var_3=lt_var_21;
  lt_if_var_3=0;
S1':
  p2=(ctrl)?lt_if_var_3:
            lt_el_var_3;
  p1=lt_var_8;
  ...
  return dist+abs(p1-p2);
}
```

Figure 4: Transformed QSDPCM code (main and new func)

This approach works recursively (just as the scalar copy propagation described in the previous Subsection) so nested innermost conditions can be handled as well. Obviously, it is not possible to rewrite other then innermost conditions to the ternary operator. However, mostly the non-innermost conditions have affine and manifest control expressions. Thus they can be extracted to the (high-level) loop transformation models. Otherwise, the different cases (branches) have to be handled separately similar to the scenario approach [5].

2.3 Loop carried scalars handling

When the use of a loop carried scalar (i.e. a scalar that is not a iterator and is dependent on any previous iteration) is detected, the backtracking of the dependencies is stopped. The particular function is closed and a new backtracking (with a newly created function) begins at the point of the loop carried scalar definition. This concept can be explained in Figure 3. In statement *S5* the backtracking is stopped and a new one is started at statement *S4* (see the resulting code on Figure 4).

At this point, it is important to mention that the resulting (newly created) function parameters are not only array writes as we simplified at the beginning of this Section, but also iterators and loop carried scalars (see Figure 4).

3 Conclusions

In this paper we have demonstrated an automated technique for hierarchical code rewriting. After applying this technique we have raised the level of code abstraction only to the loop hierarchy and indexed signals. The remaining information about arithmetic, logic and data dependent operations is hidden in the automatically created functions. We have implemented our technique in a prototype tool and tested it on real-life multimedia applications like the DAB FFT and QSDPCM. After the rewriting we are able to pass this applications to the models used in the loop transformation stage which was not feasible with the original or manually rewritten code. Nowadays our approach can handle scalar copy propagation, innermost conditions and loop carried scalars which is sufficient for current drivers. In our future work we would like to use more drivers to identify and be able to handle more constructs.

References

[1] F.Catthoor, K.Danckaert, C.Kulkarni, E.Brockmeyer, P.G.Kjeldsberg, T.Van Achteren, T.Omnes, "Data access and storage management for embedded programmable processors", ISBN 0-7923-7689-7, Kluwer Acad. Publ., Boston, 2002.

[2] K.Danckaert, "Loop transformations for data transfer and storage reduction on multiprocessor systems", *Doctoral dissertation*, ESAT/EE Dept., K.U.Leuven, Belgium, May 2001.

[3] S.Verdoolaege, F.Catthoor, M.Bruynooghe, G.Janssens, "Feasibility of incremental translation", *Report CW 348*, CW Dept., K.U.Leuven, Belgium, October 2001.

[4] S.Wuytack, "Scalar data flow analysis in ATOMIUM", *Internal report*, IMEC, Leuven, Belgium, 2001.

[5] P.Yang, P.Marchal, C.Wong, S.Himpe, F.Catthoor, P.David, J.Vounckx, R.Lauwereins, "Managing Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems", invited paper in *Proc. 15th ACM/IEEE Intnl. Symp. on System-Level Synthesis* (ISSS), Kyoto, Japan, pp.112-119, Oct. 2002.