



An Automated C++ Code and Data Partitioning Framework for Data Management of Data-Intensive Applications

**A. Milidonis¹, G. Dimitroulakos¹, M. D. Galanis¹,
G. Theodoridis², C.Goutis¹, and F.Catthoor³**

¹VLSI Design Lab., Dept. of Elect. and Comp. Eng., Univ. of Patras, Greece

²Dept. of Physics, Aristotle Univ. of Thessaloniki, 54124, Greece

³IMEC, Kapeldreef 75 B-3001 Leuven, Belgium

**8th International Workshop on Software and Compilers for Embedded Systems
SCOPES 2004**

Amsterdam, The Netherlands, September 2 - 3, 2004



Presentation Overview

- Data Management and Code Partitioning
- General Description of the Introduced Framework
- Pre-processing Analysis
- Data Type Partitioning
- Code Rewriting
- Experimental Results
- Conclusions



Introduction

- Characteristics of current multimedia applications:
 - High complexity
 - Diverse functionality
 - Huge amount of data transfers
 - Large data storage requirements
 - Hundreds of thousands code lines



Introduction

- Conventional design procedure:
 - Analysis of the initial code
 - Identification of crucial parts for design factors such as performance and power consumption
 - Refinement and optimization of crucial parts (background memory management)
 - Mapping to predefined or custom developed platforms



Motivation

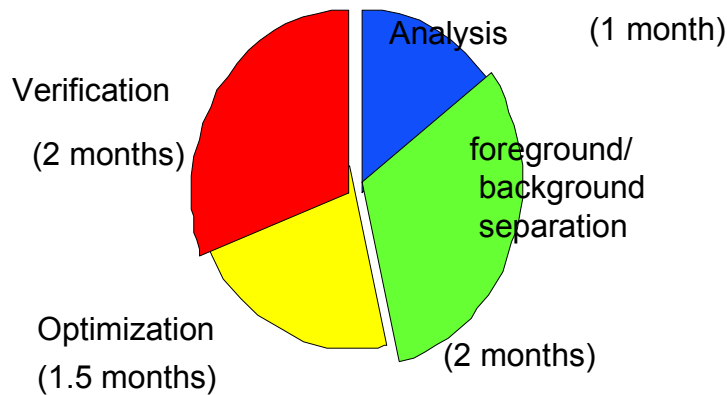
- Only a small amount of code is important for the design factors
- Need for automatic identification of crucial parts and separation from the rest of the code
- Benefits:
 - Reduced complexity
 - Increased exploration freedom
 - Reduced design time



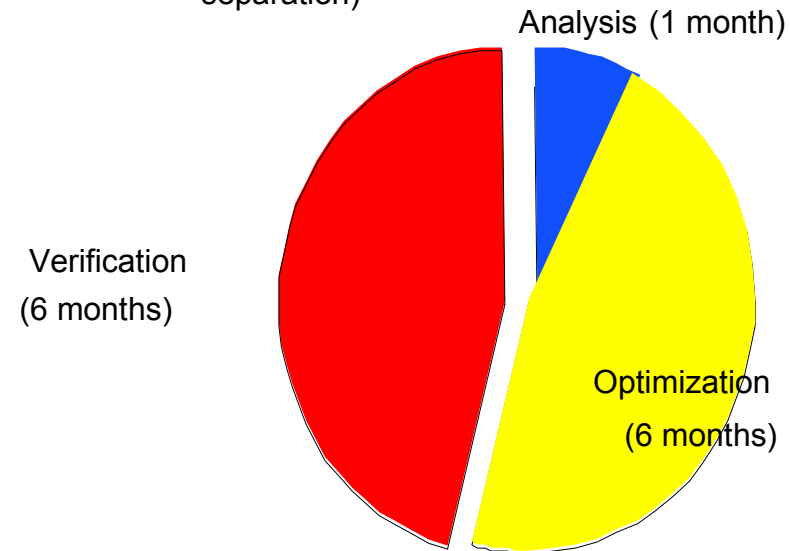
Motivation

MPEG-4

(6.5 months using foreground/background separation)



(13 months without foreground/background separation)



Foreground/background separation shortens the exploration time

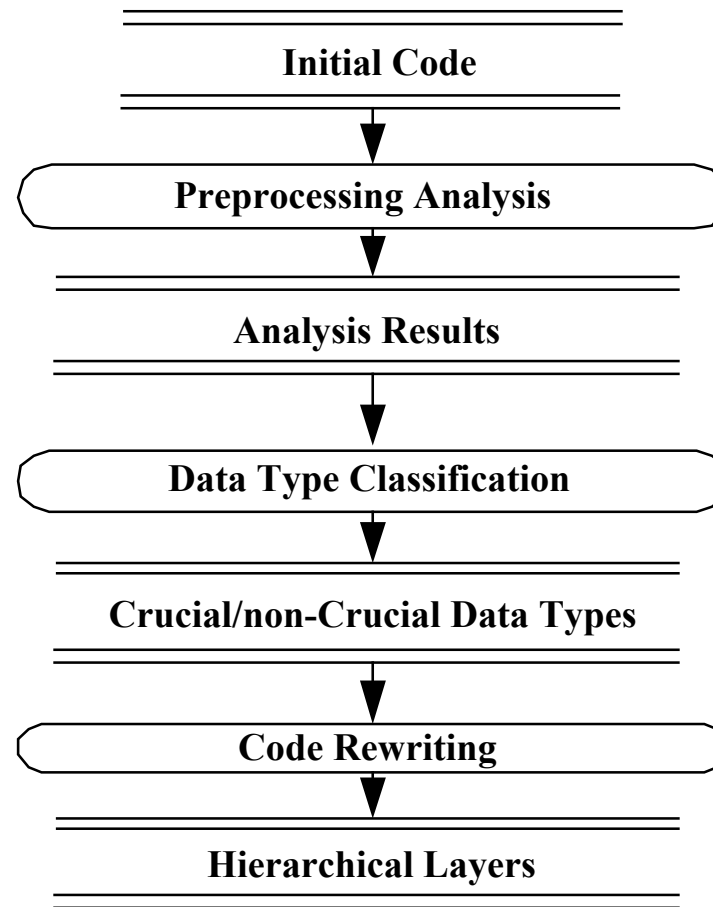


General Description of the Introduced Framework

- Implementation of the methodology for separating the initial code in two layers
- As a data type is considered the array
- Input: C++ code
- Operations: static and dynamic analysis, analysis results processing
- Output:
 - function flow graph
 - crucial data types
 - layers two and three



General Description of the Introduced Framework





Used software suites

- EDG compiler
 - C generating back end for converting C++ code to C
- Suif2 compiler
 - Static analysis for record size extraction and for scanning special expressions (function calls, class declarations etc.)
- LEX
 - Scanning special words and placing extra routines for dynamic analysis and for reporting the results (e.g. counter placement under each data type static access)



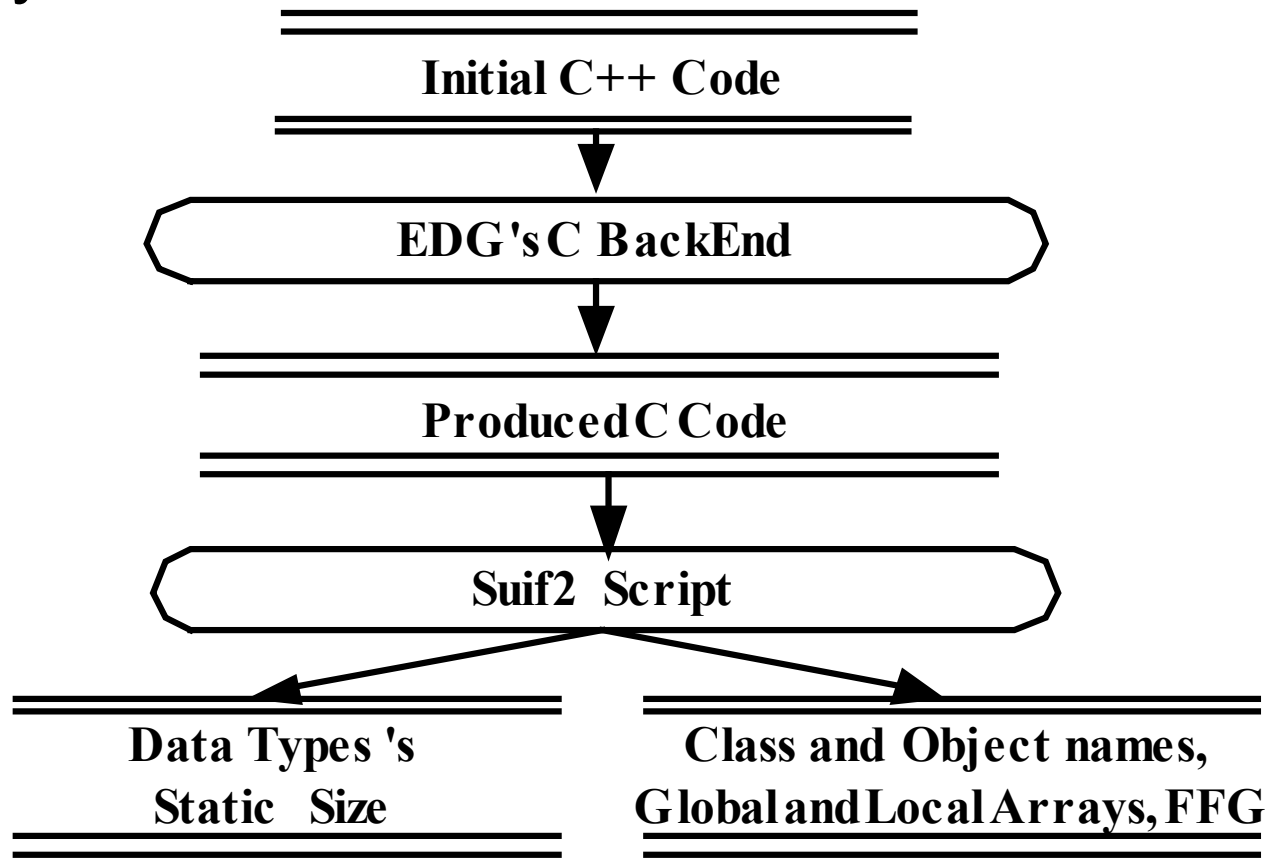
Pre-processing Analysis

- Determination of crucial data types
 - Number of memory accesses
 - Memory storage requirements
- Identification of code portions that manipulate the crucial data types



Pre-processing Analysis

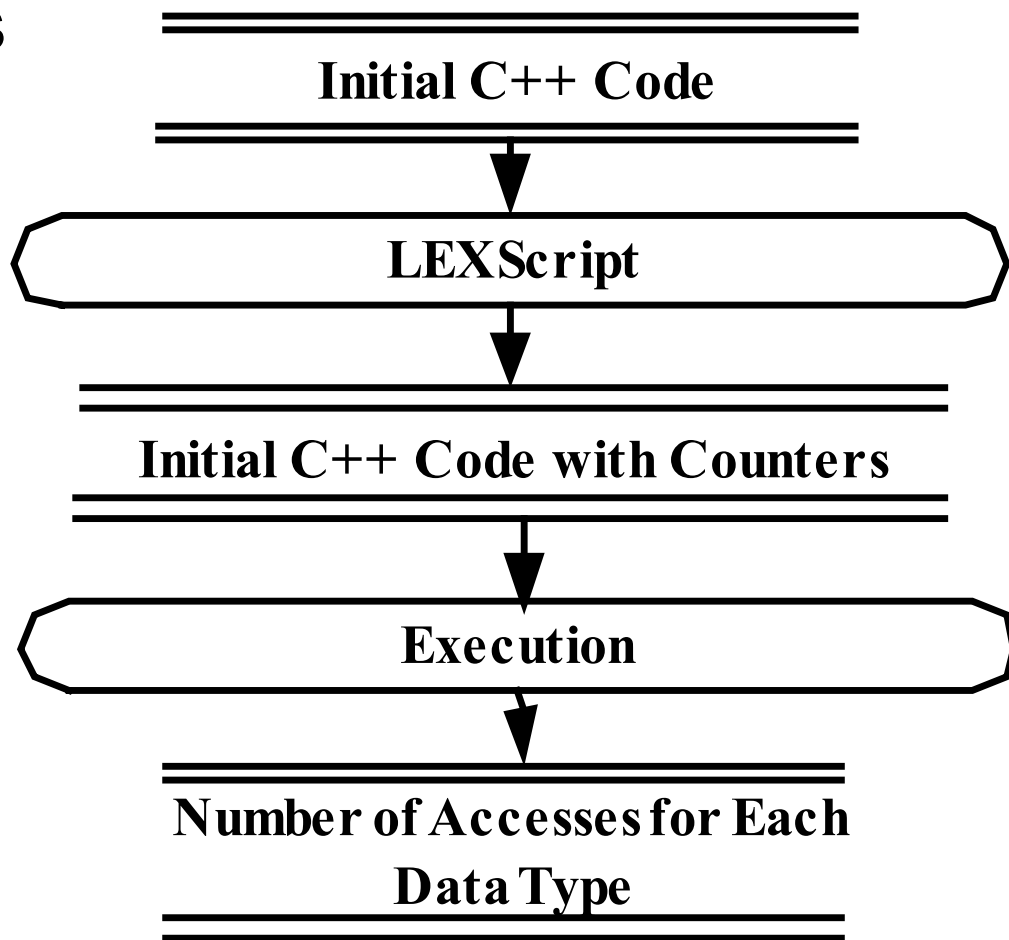
- Static Analysis





Pre-processing Analysis

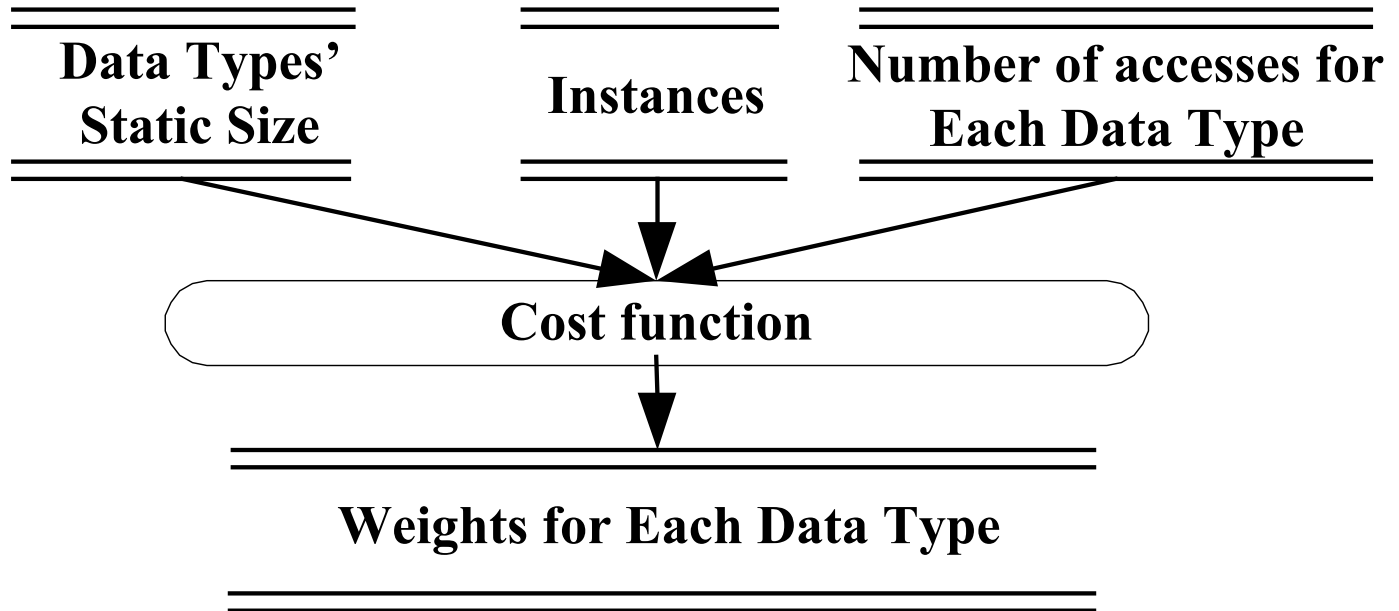
- Dynamic Analysis





Data Type Partitioning

Weight assignment procedure





Data Type Partitioning

– Cost function

- Assigns a weight to each data type for defining how crucial this is
- It must be called for each data type before the execution ending of the code
- At the current version of the tool is defined as:
Weight = static size * number of accesses
- There may be an option for changing this function by placing a different file for its description
- The threshold of weights beyond which all data types are crucial shall be defined by the user



Code Rewriting

- The definition of the weight threshold will reveal the crucial data types
- By scanning the basic blocks of the functions that access crucial arrays using LEX they will be marked by a comment informing which layer they belong to



Code Rewriting

```
class classA{
public:
int b;
void functionA(void);
};
void classA::functionA(void){
inta,c,d;
a=2;
b=3;
c=a+2;
d=b+1;
if(c)
d=c+1;
}
```

```
class classA{
public:
int b;
void f1(int &a,int &c,int&d);
void functionA(void);
};
void classA::functionA(void){
inta,c,d;
f1(a,c,d);
if(c)
d=c+1;
}
```

```
void classA::f1(int &a,
int &c,int &d){ //Layer 3
a=2;
b=3;
c=a+2;
d=b+1;
}
```



Rewriting Example

Initial motion estimation code:

```
void full_s ::fs_motion_estimation(int cur[N][M],int prev[N][M],
int vec_x[N/B][M/B],int vec_y[N/B][M/B]){
int x,y,i,j,k,l,p1,p2,min,dist;
for(x=0;x<N/B;x++) for(y=0;y<M/B;y++){
    min=65280;
    for(i=-p;i<p+1;i++) for(j=-p;j<p+1;j++){
        dist=0;
        for(k=0;k<B;k++) for(l=0;l<B;l++){
            p1=cur[B*x+k][B*y+l];
            if((B*x+i+k)<0 || (B*x+i+k)>(N-1) ||
(B*y+j+l)<0 || (B*y+j+l)>(M-1))
                p2=0;
            else
                p2=prev[B*x+i+k][B*y+j+l];
            dist+=abs(p1-p2);}
        if(dist<min){
            min=dist;
            vec_x[x][y]=i;
            vec_y[x][y]=j;}}}}
```



Rewriting Example

Analysis results for motion estimation code

Data types	# of Accesses	Storage size (bits)	Weight
cur	5702400	811008	4624692019200
prev	5436736	811008	4409236389888
vec_x	2509	3168	7948512
vec_y	2509	3168	7948512



Rewriting Example

```
void full_s ::fs_motion_estimation(int
cur[N][M],int prev[N][M],
int vec_x[N/B][M/B],int vec_y[N/B][M/B]){
int x,y,i,j,k,l,p1,p2,min,dist;
for(x=0;x<N/B;x++) for(y=0;y<M/B;y++){
    min=65280;
    for(i=-p;i<p+1;i++) for(j=-p;j<p+1;j++){
        dist=0;
        for(k=0;k<B;k++) for(l=0;l<B;l++){
            p1=cur[B*x+k][B*y+l];
            if((B*x+i+k)<0 || (B*x+i+k)>(N-1) ||
(B*y+j+l)<0 || (B*y+j+l)>(M-1))
                p2=0;
            else
                p2=prev[B*x+i+k][B*y+j+l];
            dist+=abs(p1-p2);}
        f1(dist,min,vec_x,vec_y,i,j,x,y)
    }}}
//Layer 3:
void f1(&dist,&min,vec_x,
vec_y,&i,&j,&x,&y)
{
    if(dist<min){
        min=dist;
        vec_x[x][y]=i;
        vec_y[x][y]=j;
    }
}
```

Basic block separation to layer 3



Experimental Results

Data types	Accesses (Tool)	Accesses (counters by hand)	Static size (Bytes)	Weight
JPEG				
Dct	786508	786508	525504	4,13313E+11
Quantize	65664	65664	527552	34641174528
Zigzag	131200	131200	528576	69349171200
Entropy	13104	13104	549824	7204893696
OFDM				
Scrambler	69127	69127	1784	123322568
Fec	99876	99876	4080	407494080
Interleaver	15360	15360	2304	35389440
Cp	94176	94176	4608	433963008
VIC				
Decoder	140	140	1536	215040
P64Decoder	13565	13565	198208	2,689E+09
Matcher	104	104	96	9984
TclObject	757	757	128	96896

Tool's analysis results vs counters placed by hand for arrays memory accesses



Object function calls of Quantify

OFDM	
Data types	Function Calls (Quantify)
Scrambler	40
Fec	40
Interleaver	40
Cp	40
JPEG	
Data types	Function Calls (Quantify)
Dct	1024
Quantize	1024
Zigzag	1024
Entropy	1024



Experimental Results

Apps.	Total Lines	Crucial Lines	Crucial Data Types	Memory Requirements (bytes)	Exec. Time (sec)
JPEG	935	140	Dct	22684	21
OFDM	1050	406	Fec,Cp,Scrambler	5204	3
Cavity Detect	576	303	g_image,Gauss_blur	2352	3
			Detect_roots		
ADPCM	417	417	Encoder,Decoder	4124	2
VIC	9208	1417	P64Decoder	5972	5

Tool's memory, time requirements and applications' code size total decrement



Conclusions

- The proposed automated flow implements code data memory partitioning
- Separates the important code for background memory management optimization purposes
- Offers serious design time reduction
- Decreases the verification time



Thank you