

Compilers for Embedded Systems

Summer Term 2022

Heiko Falk

Institute of Embedded Systems
Electrical Engineering, Computer Science and Mathematics
Hamburg University of Technology

Chapter 5

HIR Optimizations and Transformations

Outline

1. Introduction & Motivation
2. Compilers for Embedded Systems – Requirements & Dependencies
3. Internal Structure of Compilers
4. Pre-Pass Optimizations
- 5. HIR Optimizations and Transformations**
6. Code Generation
7. LIR Optimizations and Transformations
8. Register Allocation
9. WCET-Aware Compilation
10. Outlook

Chapter Contents

5. HIR Optimizations and Transformations

- Motivation
- Function Specialization / Procedure Cloning
 - General-Purpose Functions in specialized Contexts
 - Interlude: Standard Optimizations & WCET Estimation
 - Function Specialization and WCETs
 - Results ($WCET_{EST}$, ACET, Code Size)
- Parallelization for Homogeneous Multi-DSPs
 - Introduction, Multi-DSP Architectures
 - Program Recovery
 - Data Partitioning & Strip Mining
 - Parallelization
 - Memory Assignment, Array Descriptors, DMA
 - Results

Motivation of HIR Optimizations

High-Level IRs:

- Very close to source language
- High-level language constructs (esp. loops, function calls with parameter passing, array accesses) preserved

High-Level Optimizations

- Exploit particularly these HIR features
- Focus on strong reorganization of loops and function relationships
- Are difficult to realize at lower abstraction levels, since required high-level information (e.g., loop bounds or function call relationships) get lost and are difficult to reconstruct

Chapter Contents

5. HIR Optimizations and Transformations

- Motivation
- Function Specialization / Procedure Cloning
 - General-Purpose Functions in specialized Contexts
 - Interlude: Standard Optimizations & WCET Estimation
 - Function Specialization and WCETs
 - Results ($WCET_{EST}$, ACET, Code Size)
- Parallelization for Homogeneous Multi-DSPs
 - Introduction, Multi-DSP Architectures
 - Program Recovery
 - Data Partitioning & Strip Mining
 - Parallelization
 - Memory Assignment, Array Descriptors, DMA
 - Results

Function Specialization

Also Called “Procedure Cloning”

- Rather old and well-known compiler technique (1993)
- Goals: To enable other optimizations, to reduce the overhead to pass parameters during function calls
- Approach: Of particular functions, specialized copies (“clones”) are created which have less parameters than the original function

Specialization so-called Interprocedural Optimization

- Explicit consideration of calling relationships between functions during an optimization

In Contrast to Intraprocedural Optimizations

- Optimization just locally inside a function f , without consideration of functions that f calls, or by which f is called

Typical Function Calls

```
int f( int *x, int n, int p ) {  
    for (i=0; i<n; i++) {  
        x[i] = p * x[i];  
        if ( i == 10 ) { ... }  
    }  
    return x[n-1];  
}
```

```
int main() {  
    ... f( y, 5, 2 ) ...  
    ... f( z, 5, 2 ) ...  
    return f( a, 5, 2 );  
}
```

Observations

- `f` is called several times
- `f` has 3 arguments
- Arguments `n` and `p` are instantiated several times with constants 5 and 2

More Observations

```
int f( int *x, int n, int p ) {
    for (i=0; i<n; i++) {
        x[i] = p * x[i];
        if ( i == 10 ) { ... }
    }
    return x[n-1];
}
```

```
int main() {
    ... f( y, 5, 2 ) ...
    ... f( z, 5, 2 ) ...
    return f( a, 5, 2 );
}
```

- f is so-called *general-purpose function*: Via parameter n , an array x of arbitrary size can be treated
- Control flow in f depends on function parameter
- f is used in `main` in a *special-purpose context*: Processing of arrays of size $n = 5$.

Specialization of General-Purpose Functions

```
int f( int *x, int n, int p ) {  
    for (i=0; i<n; i++) {  
        x[i] = p * x[i];  
        if ( i == 10 ) { ... }  
    }  
    return x[n-1];  
}
```

```
int main() {  
    ... f_5_2( y ) ...  
    ... f_5_2( z ) ...  
    return f_5_2( a );  
}
```

```
int f_5_2( int *x ) {  
    for (i=0; i<5; i++) {  
        x[i] = 2 * x[i];  
        if ( i == 10 ) { ... }  
    }  
    return x[4];  
}
```

Expected Effects of Specialization

Reduction of the Average-Case Execution Time ACET due to

- Enabling of other standard compiler optimizations inside the specialized function
- Less code to be executed for parameter passing when calling the specialized function

Standard Optimizations after Cloning (1)



Constant Folding

- Pre-computation of expressions with purely constant operands already at compile-time
- Example:

Original code

```
a = p * n;
```

+ *Cloning*

```
a = 2 * 5;
```

+ *ConstFold*

```
a = 10;
```

Standard Optimizations after Cloning (2)



Constant Propagation

- Replacement of accesses to variables that provably have constant content by the constant itself
- Example:

| <i>Original code</i> | + <i>Cloning</i> | + <i>ConstFold</i> + <i>ConstProp</i> |
|-------------------------------------|--|---|
| <code>a = p * n;</code> | <code>a = 2 * 5;</code> | <code>a = 10;</code> |
| <code>for (...; i<a; ...)</code> | <code>for (...; i<a; ...)</code> | <code>for (...; i<10; ...)</code> |

Standard Optimizations after Cloning (3)



Strength Reduction

- Replacement of computationally expensive computations (e.g., multiplications, divisions, ...) by “cheaper” operations (e.g., additions, subtractions, shifts, ...)
- Example:

Original code

```
x[i] = p*x[i];
```

+ Cloning

```
x[i] = 2*x[i];
```

+ StrengthRed

```
x[i] = x[i]<<1;
```

Standard Optimizations after Cloning (4)

Control Flow Optimizations

- Determination of possible value ranges of variables after constant folding and propagation; removal of redundant if-statements whose conditions are always true or always false

( see also chapter 4 – Loop Nest Splitting / Condition Satisfiability)

- Example:

| <i>Original code</i> | <i>+ Cloning</i> | <i>+ ControlFlowOpt</i> |
|--|--|---------------------------------------|
| <code>for (i=0 ; i<n ; i++)</code> | <code>for (i=0 ; i<5 ; i++)</code> | <code>for (i=0 ; i<5 ; i++)</code> |
| <code>{</code> | <code>{</code> | <code>{</code> |
| <code> ... ;</code> | <code> ... ;</code> | <code> ... ;</code> |
| <code> if (i==10) ... ;</code> | <code> if (i==10) ... ;</code> | <code>}</code> |
| <code>}</code> | <code>}</code> | |



Expected Effects of Specialization

Reduction of the Average-Case Execution Time ACET due to

- Enabling of other standard compiler optimizations inside the specialized function
- Fewer code for parameter passing to be executed when calling the specialized function

Increase of Code Size due to

- Generation of potentially many novel, specialized clones

Impact of Cloning on WCET Estimation

Function Specialization and WCET considered only in very recent research

Recall

- Worst-Case Execution Time WCET:
Upper bound of a program's run-time over all possible input data

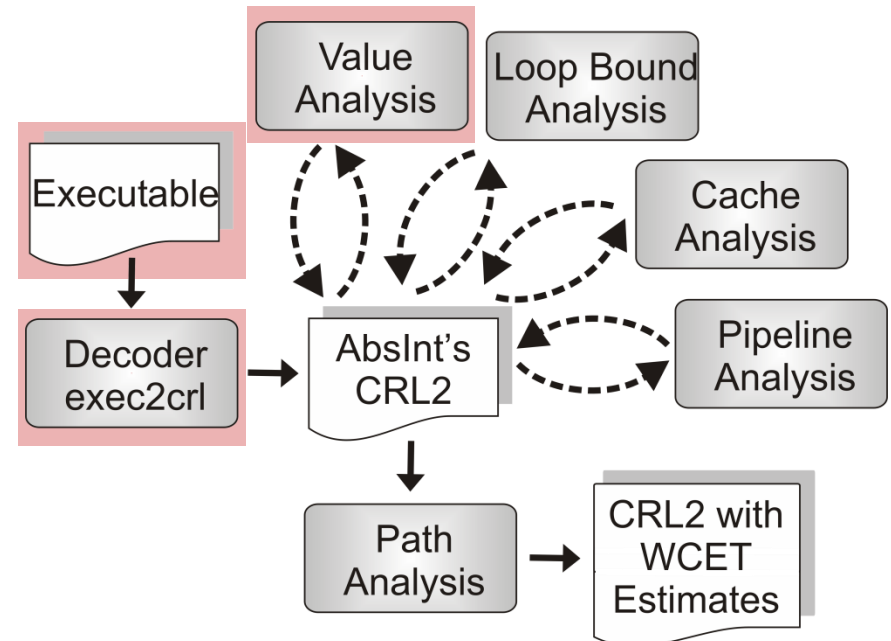
WCET Estimation

- Via static analysis of the binary code of executable programs
- In the following: Workflow of the static WCET Analyzer aiT

*[AbsInt Angewandte Informatik GmbH,
<http://www.absint.com/ait>, Saarbrücken, 2020]*

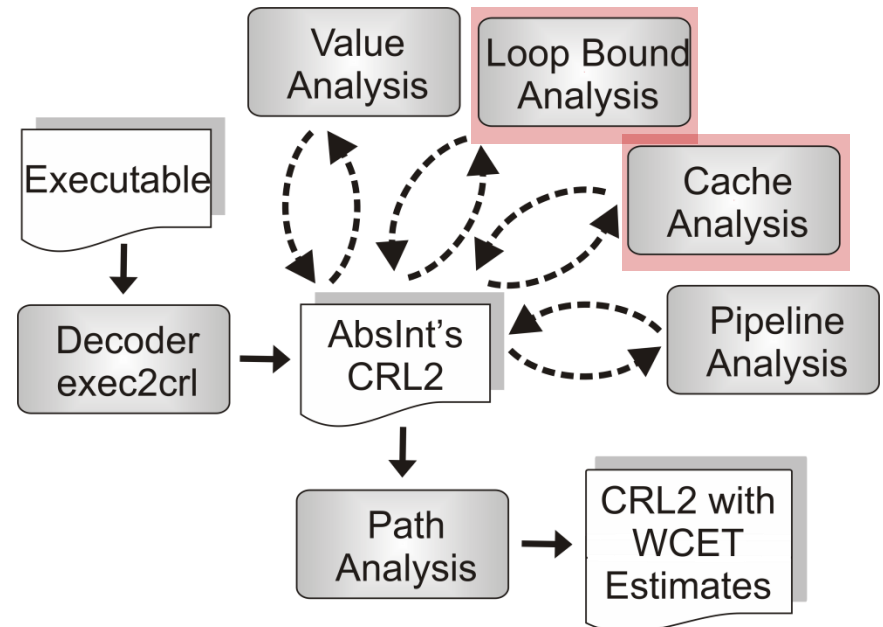
Workflow of the WCET Analyzer aiT (1)

- **Input:** Binary executable P to be analyzed
- **exec2crl:** Disassembler, translates P into aiT's low-level intermediate representation CRL2
- **Value analysis:** Computes possible contents of processor registers for any point in time during P 's execution.
*Note: P is never executed by aiT!
 P is „only“ analyzed.*



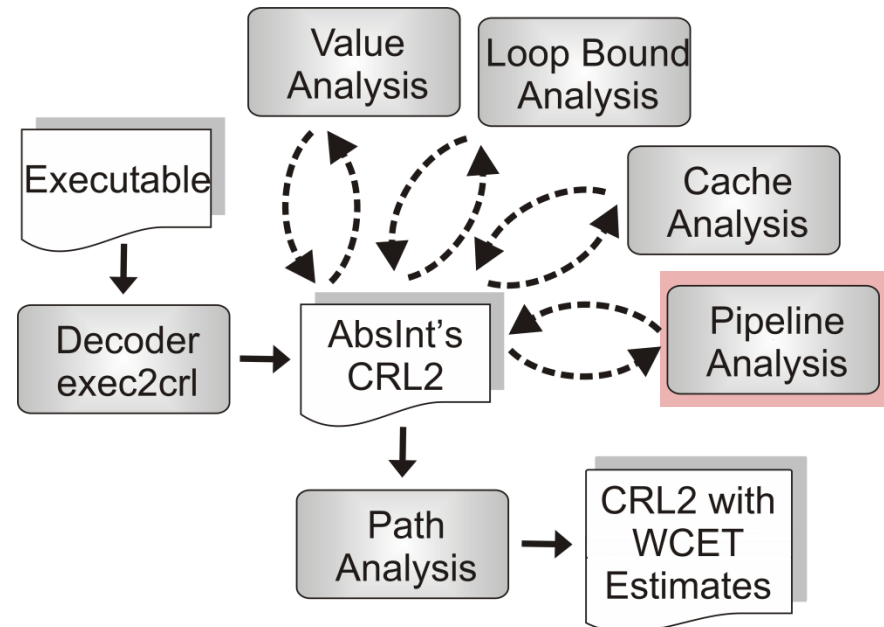
Workflow of the WCET Analyzer aiT (2)

- **Loop bound analysis:** Tries to determine lower and upper bounds for the number of iterations of each loop in P .
- **Cache analysis:** Makes use of a formal cache model, classifies each memory access in P as definite cache hit, definite cache miss, or unknown.



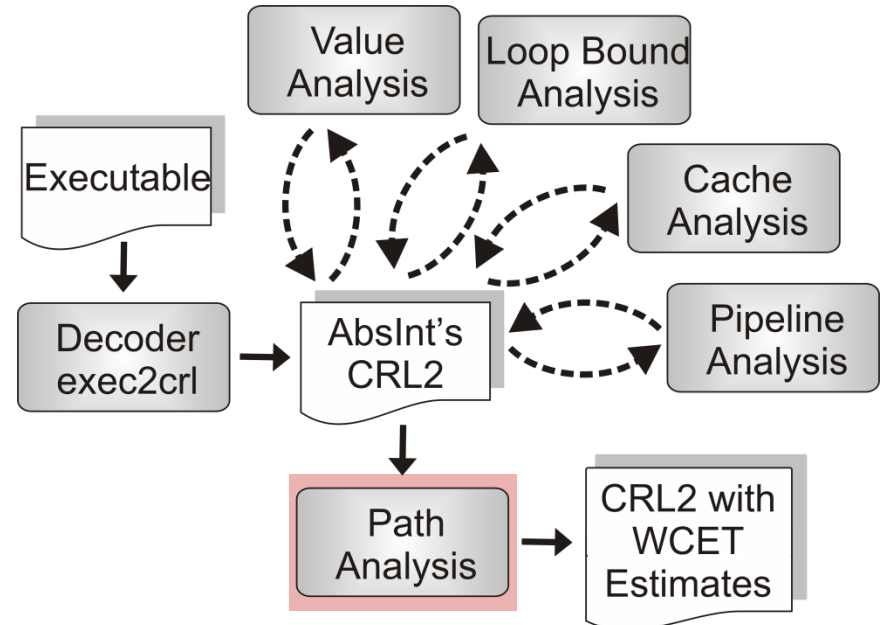
Workflow of the WCET Analyzer aiT (3)

- **Pipeline analysis:** Includes an accurate model of the processor's pipeline. Depending on the pipeline's initial state, possible cache states etc., possible states of the pipeline at the end of each basic block of P are determined. Result of the pipeline analysis is the $WCET_{EST}$ of each individual basic block.



Workflow of the WCET Analyzer aiT (4)

- **Path analysis:** Models all possible execution paths within P under consideration of the $WCET_{EST}$ of all basic blocks. Determines the longest possible execution path within P which leads to the overall $WCET_{EST}$ of P .
Result of the path analysis is, e.g., the length of this longest path, i.e., P 's $WCET_{EST}$.



Workflow of the WCET Analyzer aiT (5)

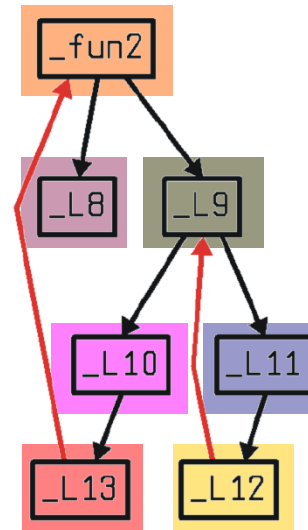
Control Flow Graph (CFG)

- Fundamental code representation during path analysis

Definition: $CFG = (V, E, s)$ is a directed graph with

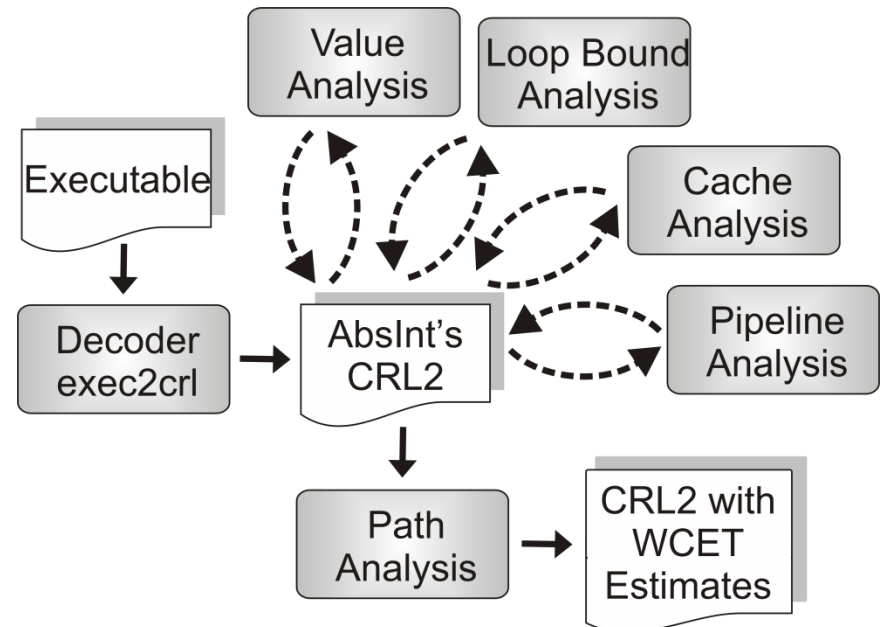
- $V = \{ b_1, \dots, b_n \}$ (set of all basic blocks, cf. chapter 2)
- $E = \{ (b_i \rightarrow b_j) \mid \text{basic block } b_j \text{ can be executed immediately after } b_i \}$
- $s \in V =$ the unique start node of the CFG

```
int fun2() {
    for (; a1<30; a1++ ) {
        for (; b1<a1; b1++ )
            printf( "%d\n", b1 );
        printf( „%d\n“, a1 );
    }
    return a1; }
```



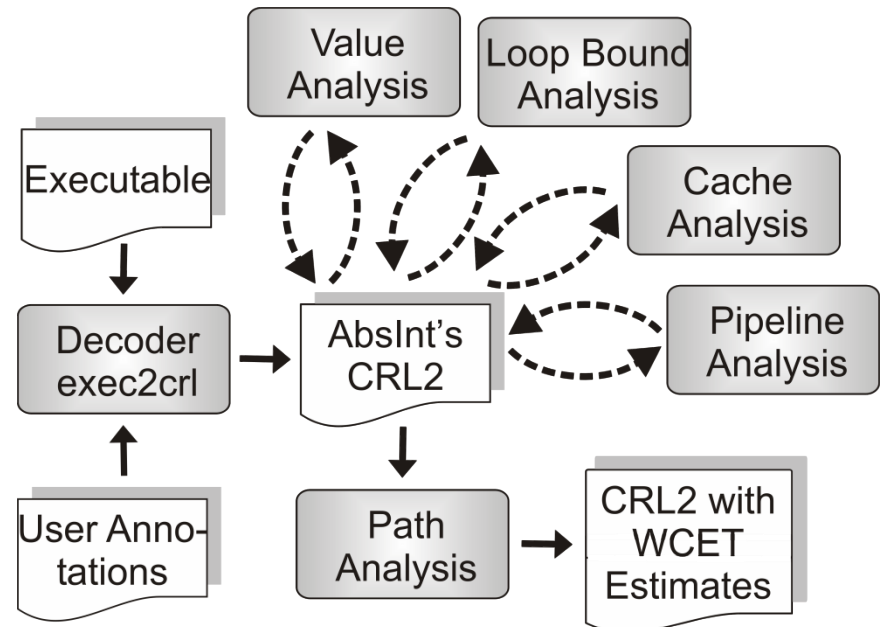
Complexity Issues

- **Problem:** WCET analysis is not computable with today's machines! If WCET were computable, one could decide in $O(1)$ if $WCET < \infty$ and thus solve the Halting problem.
- **Reason:** It is not computable how long P stays in loops. Automatic loop bound analysis is applicable only to simple classes of loops. (Analogously for recursive function calls.)



Annotations for WCET Analysis

- **Solution:** The user of aiT must mandatorily provide information about, e.g., minimal and maximal iteration bounds of loops and recursion depths.
- **Annotation file:** Contains such user-provided annotations (“flow facts”) and is – besides the program P to be analyzed – another external input to aiT.



Why Specialization and WCET_{EST}?

Motivation

- Frequent occurrences of general-purpose functions in special-purpose contexts in embedded software
- Loop bounds are particularly often controlled by function parameters
- Loop bounds are particularly critical for WCET estimates
- Procedure Cloning allows the extremely precise annotation of loop bounds for WCET analysis

[P. Lokuciejewski. Influence of Procedure Cloning on WCET Prediction. CODES+ISSS, Salzburg, 2007]

Loop Bound Annotations & Cloning (1)

```
int f( int *x, int n, int p ) {
    // loopbound min 0, max n
    for (i=0; i<n; i++ ) {
        x[i] = p * x[i];
        if ( i == 10 ) { ... }
    }
    return x[n-1];
}
```

Original code

- Loop annotation extremely imprecise since annotation must cover all invocations of f .

- If f is called somewhere with $n = 2,000$ as maximal value, 2,000 loop iterations have to be considered always.
 - For calls like, e.g., $f(a, 5, 2)$, 2,000 loop iterations are likewise assumed
- ☞ *Heavy WCET overestimation*

Loop Bound Annotations & Cloning (2)

```
int f_5_2( int *x ) {
    // loopbound min 5, max 5
    for (i=0; i<5; i++ ) {
        x[i] = 2 * x[i];
        if ( i == 10 ) { ... }
    }
    return x[4];
}
```

Cloned code

- Loop annotation extremely precise since annotation covers all invocations of `f_5_2` exactly.

- For calls like, e.g., `f_5_2(a)`, exactly 5 iterations are assumed
- ☞ *Exact WCET estimation*
- Original function `f` or other additional clones of `f` are fully independent of annotations in `f_5_2`
- ☞ *No interdependencies between all these versions of `f` w.r.t. WCET analysis*

Realization of Function Specialization (1)

Caution when Applying Function Specialization

- Increase of code size during cloning must not be neglected!
- Cloning of every possible function which is called at least twice with the same constant as argument is usually unacceptable

Parameter-Driven Application of Function Specialization

- Size G : Never clone a function f which is larger than G
- Amount of constant arguments K : clone a function f only if at least $K\%$ of all calls of f contain constant arguments that can be used for cloning

Realization of Function Specialization (2)

Given

- Function f to be specialized
- *Dictionary* arg that maps the arguments of f to be specialized to the constant values to be used

Approach Using HIR ICD-C

- Create new function type T' that corresponds to the Type T of f , but without those arguments from arg to be specialized
- Create new function symbol S' of type T' ; add S' to that symbol table in which f is declared
- Create a copy f' of the code of f with function symbol S' ; add f' to the same compilation unit that also holds f

Realization of Function Specialization (3)

Given

- Function f to be specialized
- *Dictionary* arg that maps the arguments of f to be specialized to the constant values to be used

Approach Using HIR ICD-C (ctd.)

- For each argument $a \in arg$ to be specialized:
 - Create new local variable v' in f'
 - Create assignment $v' = \langle arg[a] \rangle$; at the very beginning of f'
 - Replace each occurrence of a in f' by v'

Realization of Function Specialization (4)

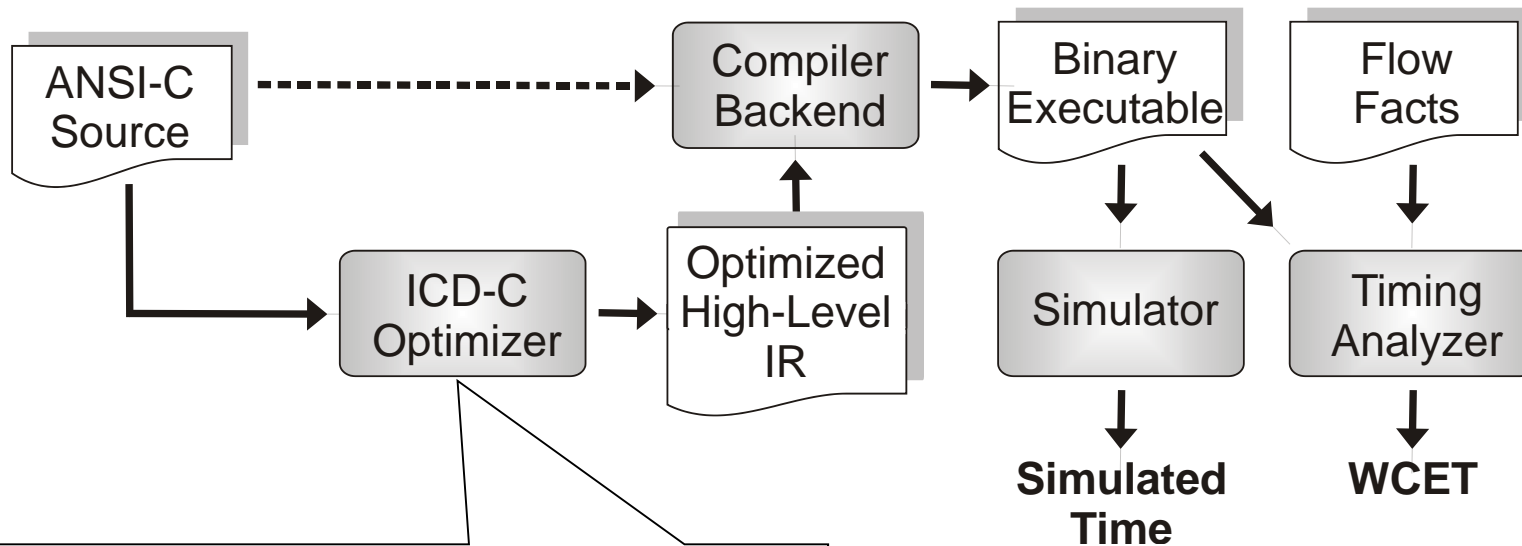
Given

- Function f to be specialized
- *Dictionary* arg that maps the arguments of f to be specialized to the constant values to be used

Approach Using HIR ICD-C (ctd.)

- For each function call c of f to be specialized:
 - Remove all arguments $a \in arg$ to be specialized from the parameter list of c
 - Replace called function f by f' within c

Workflow of Function Specialization

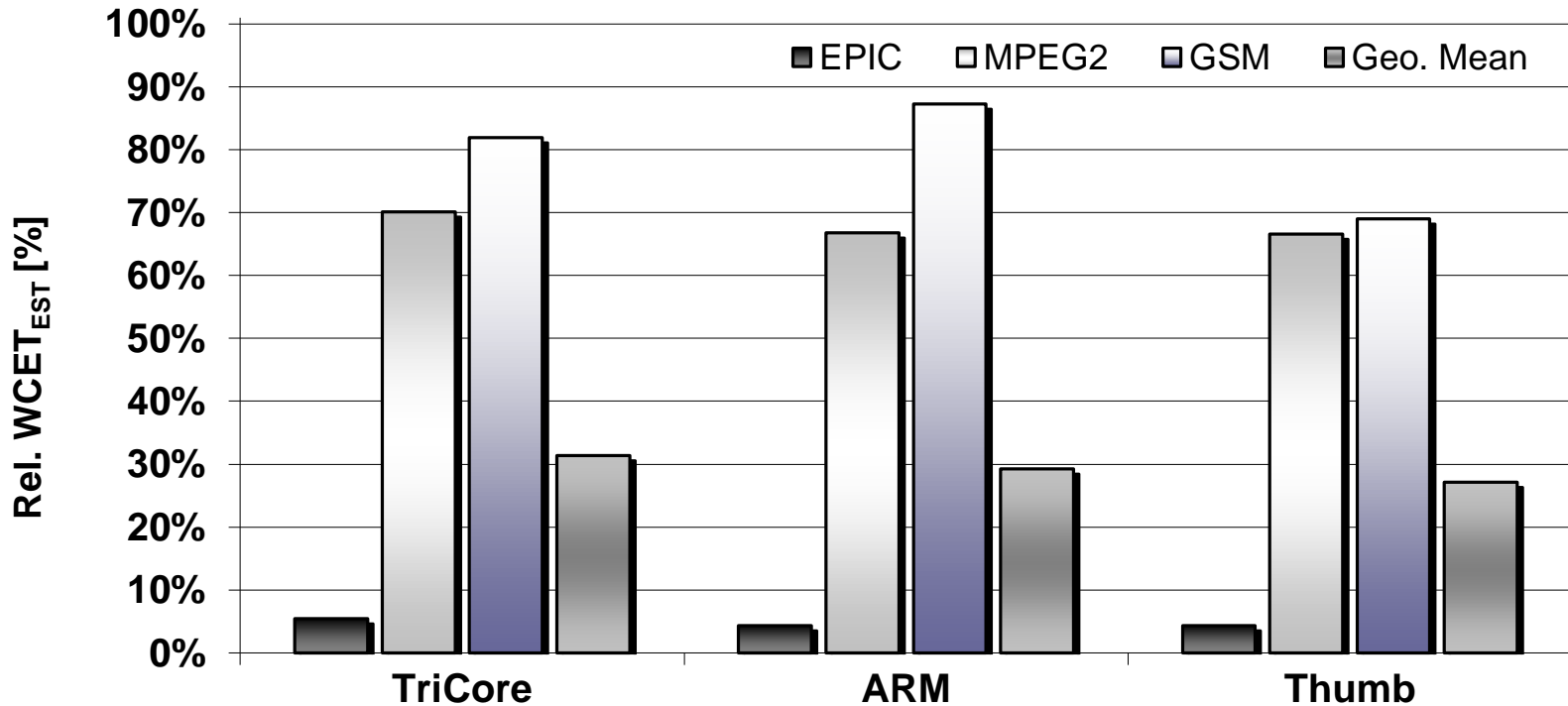


- Function specialization
- Constant folding & propagation
- Strength reduction
- If-statements
- $G = 2,000$ ICD-C expressions
- $K = 50\%$

Considered Processors

- Infineon TriCore TC1796
- ARM 7 TDMI (ARM and THUMB instruction sets)

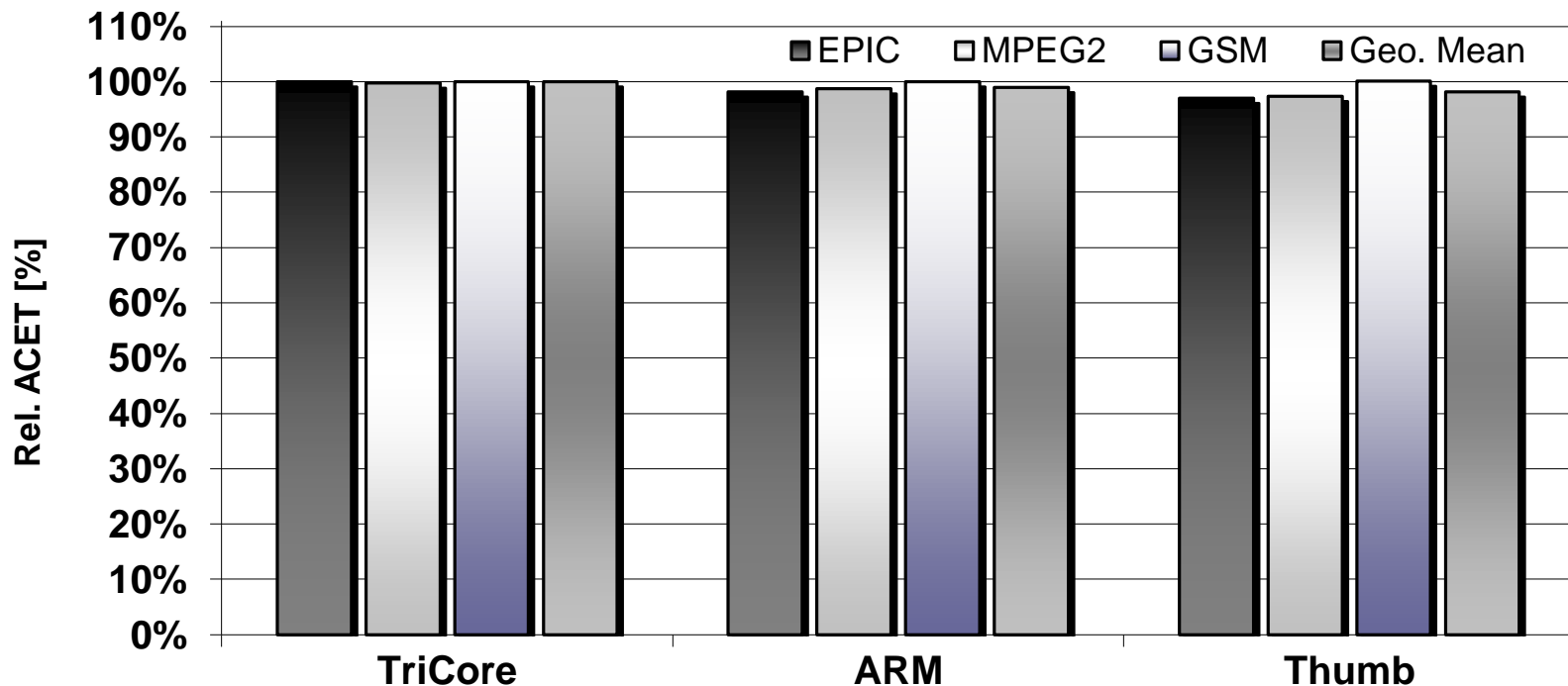
Relative WCET_{EST} after Procedure Cloning



– WCET_{EST} reductions from 13% up to 95%!



Relative ACETs after Procedure Cloning



- Marginal ACET reductions of 3% max.
- ☞ Impact of overhead for parameter passing and of successive optimization on ACET apparently negligible.

Relationship between $WCET_{EST}$ and ACET

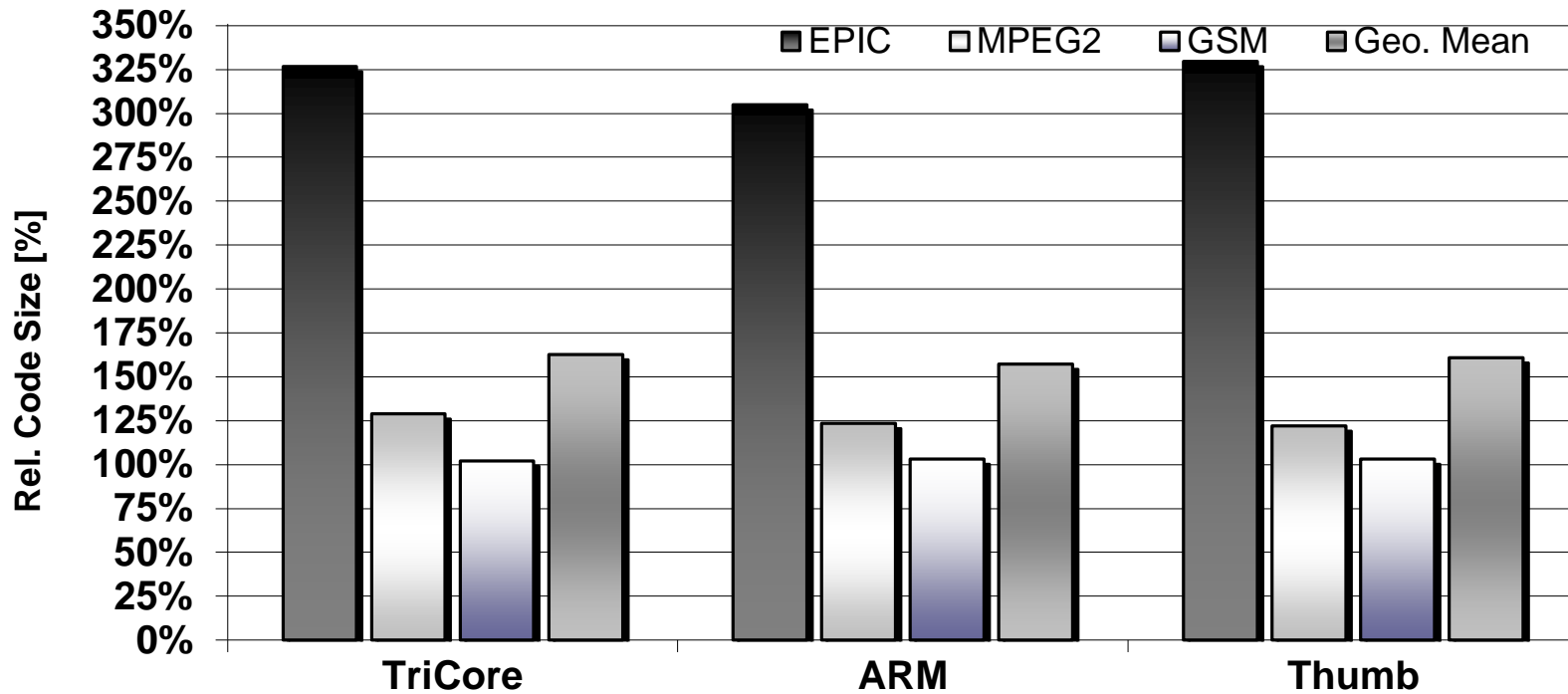
Very Astonishing

How can one and the same optimization have such a different influence on seemingly similar criteria like $WCET_{EST}$ and ACET?

Reasons

- Code before cloning only poorly annotatable
 - ☞ aiT computes quite imprecise $WCET$ estimates
- Code after cloning precisely annotatable
 - ☞ aiT computes much more precise $WCET$ estimates
- 👍 Specialization obviously improves tightness of $WCET_{EST}$ considerably (☞ *cf. chapter 3 – Objective Functions*)
- 👎 The real (and unknown) $WCET$ s are expected to be reduced only as marginally as ACETs are

Relative Code Sizes after Procedure Cloning



– Code size increases from 2% up to 225%!



Chapter Contents

5. HIR Optimizations and Transformations

- Motivation
- Function Specialization / Procedure Cloning
 - General-Purpose Functions in specialized Contexts
 - Interlude: Standard Optimizations & WCET Estimation
 - Function Specialization and WCETs
 - Results ($WCET_{EST}$, ACET, Code Size)
- Parallelization for Homogeneous Multi-DSPs
 - Introduction, Multi-DSP Architectures
 - Program Recovery
 - Data Partitioning & Strip Mining
 - Parallelization
 - Memory Assignment, Array Descriptors, DMA
 - Results

Parallelization for Multi-DSPs

Material kindly made available by

Björn Franke and Michael O'Boyle
University of Edinburgh, UK
School of Informatics



Parallelization for Multi-DSPs

Motivation

- Performance demands of complex systems often exceed capabilities of a single processor
(*e.g., radar, sonar, medical image processing, HDTV, ...*)
- Multiple DSPs operating in parallel provide enough performance, but...
 - 👉 little or no hardware support for the parallel execution of software
 - 👉 even less support of parallel programming paradigms by development tools
 - 👉 existing source codes are often written in low-level style which impedes an effective parallelization

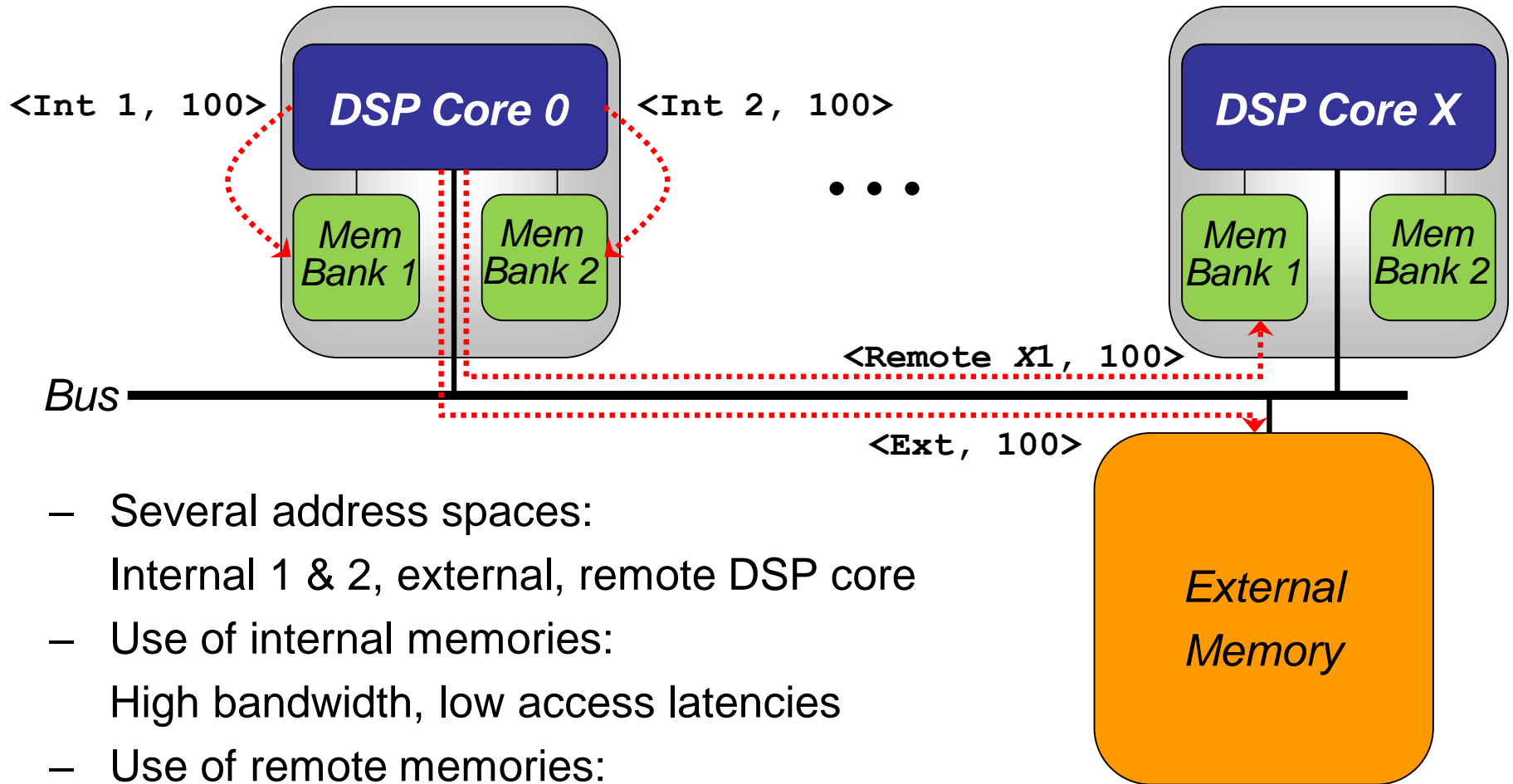
Parallelizing Compilers

Research Area “High-Performance Computing”

- Research on vectorizing compilers for more than 40 years
- Traditionally Fortran compilers
- These vectorizing compilers are usually not suitable for Multi-DSPs, since their assumptions on the memory hierarchy are unrealistic:
 - 👉 Communication between processes via shared memory
 - 👉 Memory has only one single, shared address space
 - 👉 Caches can be local and/or shared, but cache coherence issues are solved in hardware

👉 **De facto no parallelizing compilers available for Multi-DSPs**

Multi-DSPs



- Several address spaces:
Internal 1 & 2, external, remote DSP core
- Use of internal memories:
High bandwidth, low access latencies
- Use of remote memories:
ID of remote DSPs must be known

Workflow of Parallelization

Program Recovery

- Removal of undesired low-level constructs from code
- Replacement by high-level constructs

Detection of Parallelism

- Identification of loops amenable for parallelization

Partitioning and Allocation of Data

- Minimization of communication overhead between DSPs

Locality Improvement of Memory Accesses

- Minimization of accesses to memories of remote DSPs

Optimization of Memory Transfers

- Use of DMA for burst transfers

Code Example for 2 parallel DSPs

```
/* Array declarations */  
int A[16], B[16], C[16], D[16];  
  
/* Declaration & initialization of pointers */  
int *p_a = A, *p_b = &B[15], *p_c = C, *p_d = D;  
  
/* Loop over all array elements */  
for (i = 0; i < 16; i++)  
    *p_d++ = *p_c++ + *p_a++ * *p_b--;
```

- Low-level array accesses via pointers; explicit pointer arithmetic (☞ *cf. chapter 2, auto-increment addressing*)
- Disadvantageous for parallelization since ad hoc no structure in array accesses visible and analyzable

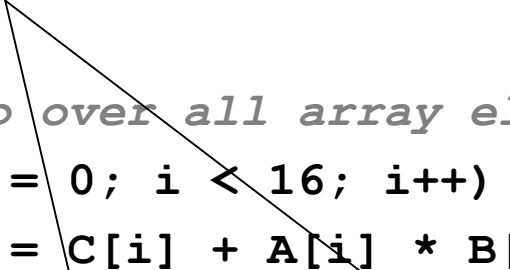
Program Recovery

```
/* Array declarations */  
int A[16], B[16], C[16], D[16];  
  
/* Loop over all array elements */  
for (i = 0; i < 16; i++)  
    D[i] = C[i] + A[i] * B[15-i];
```

- Replacement of pointer accesses by explicit array operators []
- Structure of array accesses better visible, more amenable for subsequent analyses

Program Recovery

```
/* Array declarations */  
int A[16], B[16], C[16], D[16];  
  
/* Loop over all array elements */  
for (i = 0; i < 16; i++)  
    D[i] = C[i] + A[i] * B[15-i];
```



- One-dimensional “flat” arrays too unstructured for parallelization for Multi-DSPs
- Partitioning of arrays onto available parallel DSPs unclear

Data Partitioning

```
/* Partitioned array declarations */  
int A[2][8], B[2][8], C[2][8], D[2][8];  
  
/* Loop over all array elements */  
for (i = 0; i < 16; i++)  
    D[i/8][i%8] = C[i/8][i%8] +  
                A[i/8][i%8] * B[(15-i)/8][(15-i)%8];
```

- New, two-dimensional array declarations
- First dimension equals number of available parallel DSPs
- Original flat arrays now partitioned in disjoint areas that can be processed independently from each other

Data Partitioning

```
/* Partitioned array declarations */  
int A[2][8], B[2][8], C[2][8], D[2][8];  
  
/* Loop over all array elements */  
for (i = 0; i < 16; i++)  
    D[i/8][i%8] = C[i/8][i%8] +  
                A[i/8][i%8] * B[(15-i)/8][(15-i)%8];
```

- Extremely costly and complex addressing required
- Reason: Arrays are multi-dimensional; loop index variable *i* used to index the arrays is purely sequential

Strip Mining of *i*-Loop

```
/* Partitioned array declarations */  
int A[2][8], B[2][8], C[2][8], D[2][8];  
  
/* Nested loop over all array elements */  
for (j = 0; j < 2; j++)  
    for (i = 0; i < 8; i++)  
        D[j][i] = C[j][i] + A[j][i] * B[1-j][7-i];
```

- Splitting of sequential iteration space of *i* into two independent, two-dimensional iteration spaces
- Iteration spaces of new loop nest now matches nicely with data layout
- Only affine expressions for array accesses

Strip Mining of i-Loop

```
/* Partitioned array declarations */  
int A[2][8], B[2][8], C[2][8], D[2][8];  
  
/* Nested loop over all array elements */  
for (j = 0; j < 2; j++)  
    for (i = 0; i < 8; i++)  
        D[j][i] = C[j][i] + A[j][i] * B[1-j][7-i];
```

- How can this code be parallelized for two DSPs?

Parallelization (for Processor 0)

```
/* Definition of the processor ID */
```

```
#define MYID 0
```

- Insertion of an explicit processor ID

```
/* Partitioned array declarations */
```

```
int A[2][8], B[2][8], C[2][8], D[2][8];
```

```
/* Simple loop over all array elements for DSP No. MYID */
```

```
for (i = 0; i < 8; i++)
```

```
    D[MYID][i] = C[MYID][i] + A[MYID][i] * B[1-MYID][7-i];
```

- Array accesses now make use of processor ID
- For N parallel processors, N different HIR codes are generated each of which with a unique processor ID

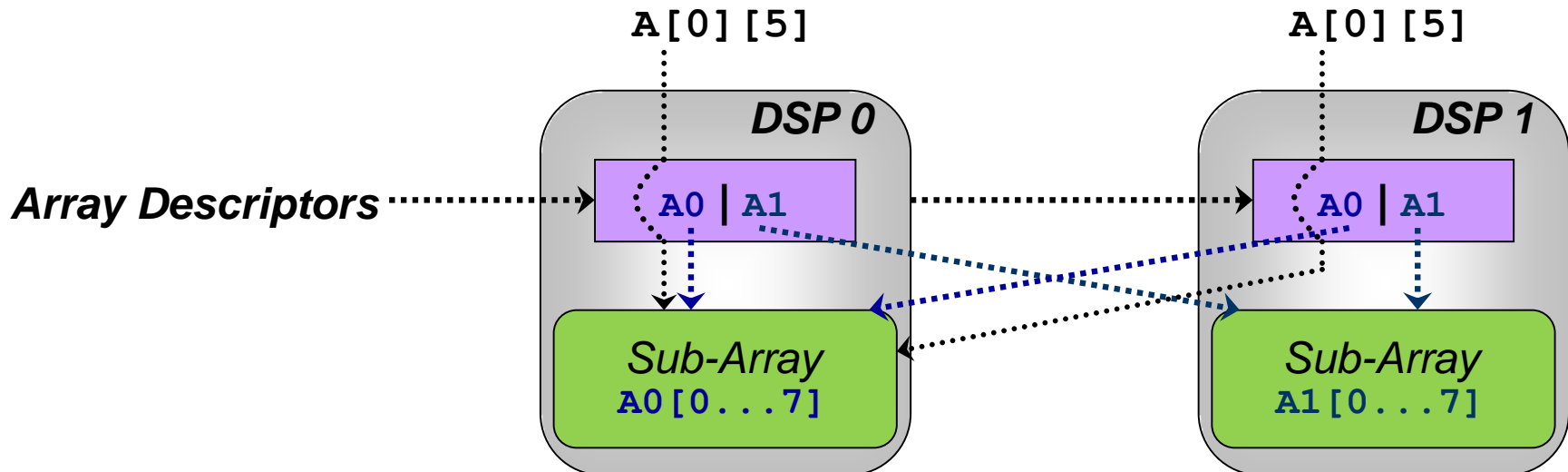
Parallelization (for Processor 0)

```
/* Definition of the processor ID */  
#define MYID 0  
  
/* Partitioned array declarations */  
int A[2][8], B[2][8], C[2][8], D[2][8];  
  
/* Simple loop over all array elements for DSP No. MYID */  
for (i = 0; i < 8; i++)  
    D[MYID][i] = C[MYID][i] + A[MYID][i] * B[1-MYID][7-i];
```

- This structure makes explicit which code will run on which DSP
- It still remains unclear how the arrays are distributed onto local memory banks or on external memories, and how accesses to memory banks of remote DSPs happen

Array Descriptors

- Two-dimensional array $\mathbf{A}[2][8]$ is partitioned into two sub-arrays $\mathbf{A0}$ and $\mathbf{A1}$ along the first dimension
- Each sub-array \mathbf{A}_n is assigned to the memory of processor n
- Original, two-dimensional array accesses need to be re-directed towards $\mathbf{A0}$ and $\mathbf{A1}$ using array descriptors



Memory Assignment (for Processor 0)

```
/* Definition of the processor ID */  
#define MYID 0
```

– Arrays in DSP-internal and remote memories

```
/* Partitioned array declarations & array descriptors */  
int A0[8]; extern int A1[8]; int *A[2] = { A0, A1 };  
int B0[8]; extern int B1[8]; int *B[2] = { B0, B1 }; ...
```

```
/* Simple loop over all array elements for DSP No. MYID */  
for (i = 0; i < 8; i++)  
    D[MYID][i] = C[MYID][i] + A[MYID][i] * B[1-MYID][7-i];
```

– Array accesses via descriptors in unchanged syntax

Memory Assignment (for Processor 0)

```

/* Definition of the processor ID */
#define MYID 0

/* Partitioned array declarations & array descriptors */
int A0[8]; extern int A1[8]; int *A[2] = { A0, A1 };
int B0[8]; extern int B1[8]; int *B[2] = { B0, B1 }; ...

/* Simple loop over all array elements for DSP No. MYID */
for (i = 0; i < 8; i++)
    D[MYID][i] = C[MYID][i] + A[MYID][i] * B[1-MYID][7-i];

```

- Descriptor access to local arrays inefficient due to additional indirection
- Scheduling problems: $A[i][j]$ can have varying access latency if i references local or remote memory

Locality Improvement for Array Accesses

```
/* Definition of the processor ID */
```

```
#define MYID 0
```

```
/* Partitioned array declarations & array descriptors */
```

```
int A0[8]; extern int A1[8]; int *A[2] = { A0, A1 };
```

```
int B0[8]; extern int B1[8]; int *B[2] = { B0, B1 }; ...
```

```
/* Simple loop over all array elements for DSP No. MYID */
```

```
for (i = 0; i < 8; i++)
```

```
    D0[i] = C0[i] + A0[i] * B[1-MYID][7-i];
```

- Direct accesses to local arrays; refrain from accesses via descriptors whenever possible
- Maximal exploitation of high bandwidths of local memories

Locality Improvement for Array Accesses

```
/* Definition of the processor ID */
```

```
#define MYID 0
```

```
/* Partitioned array declarations & array descriptors */
```

```
int A0[8]; extern int A1[8]; int *A[2] = { A0, A1 };
```

```
int B0[8]; extern int B1[8]; int *B[2] = { B0, B1 }; ...
```

```
/* Simple loop over all array elements for DSP No. MYID */
```

```
for (i = 0; i < 8; i++)
```

```
    D0[i] = C0[i] + A0[i] * B[1-MYID][7-i];
```

- 8 sequential accesses to consecutive array elements in remote memory
- Inefficient, since 8 full bus cycles are required

Insertion of DMA Burst Transfers

```
/* Definition of the processor ID */
```

```
#define MYID 0
```

```
/* Partitioned array declarations & array descriptors */
```

```
int A0[8]; extern int A1[8]; int *A[2] = { A0, A1 };
```

```
int B0[8]; extern int B1[8]; int *B[2] = { B0, B1 }; ...
```

```
/* Temporary DMA buffer */
```

```
int temp[8];
```

```
DMA_get( temp, &(B[1-MYID]), 8 * sizeof( int ) );
```

– Block-wise loading of a local buffer from remote memory via DMA

```
/* Simple loop over all array elements for DSP No. MYID */
```

```
for (i = 0; i < 8; i++)
```

```
    D0[i] = C0[i] + A0[i] * temp[7-i];
```

– Array accesses in loop use only local memory

Experimental Setup

Multi-DSP Hardware

- 4 parallel Analog Devices TigerSHARC TS-101 @250 MHz
- 768 kB local SRAM per DSP, 128 MB external DRAM

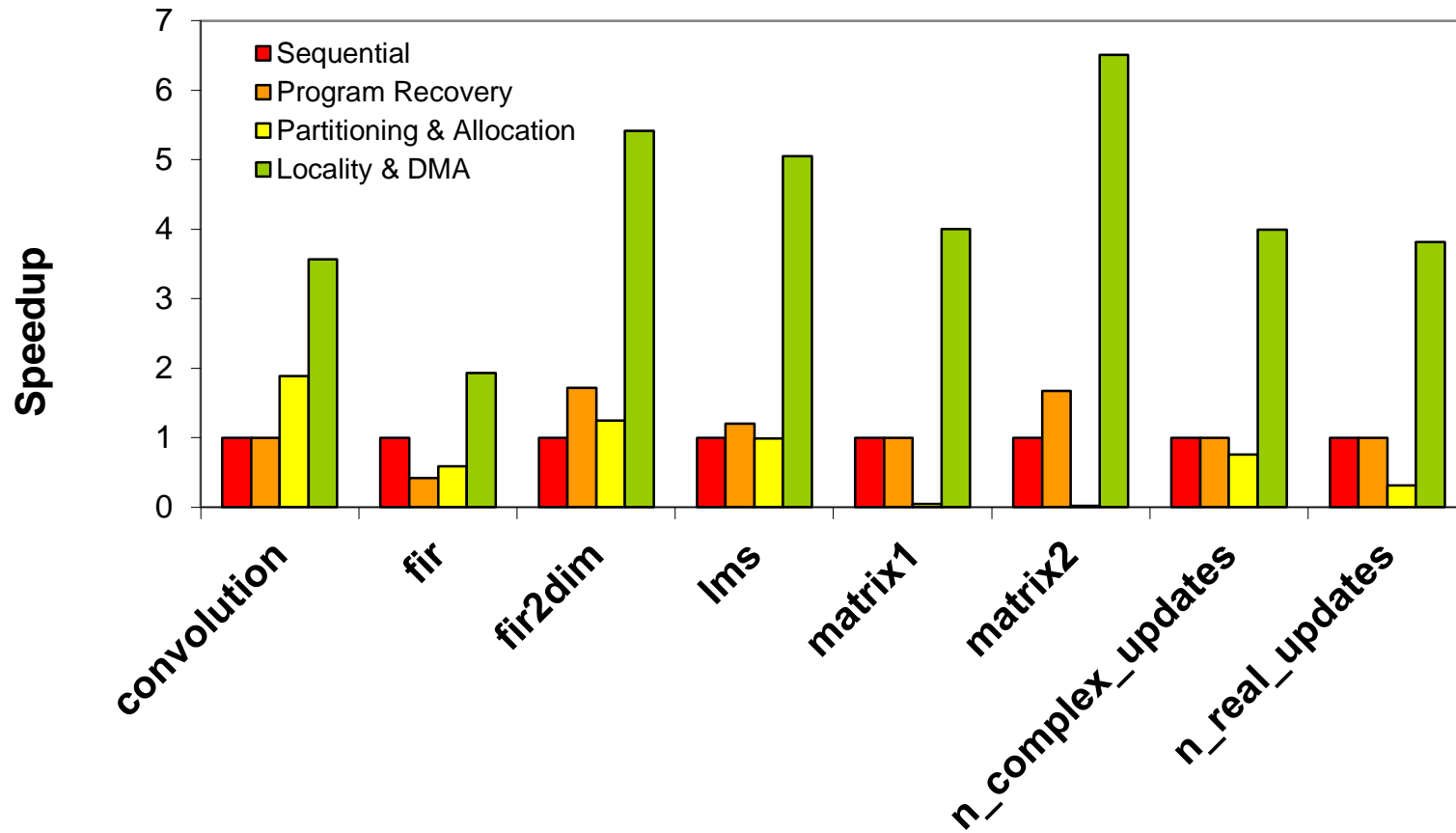
Parallelized Benchmark Programs

- *DSPstone*: small DSP kernels, low code complexity
- *UTDSP*: complex applications, compute-intensive code

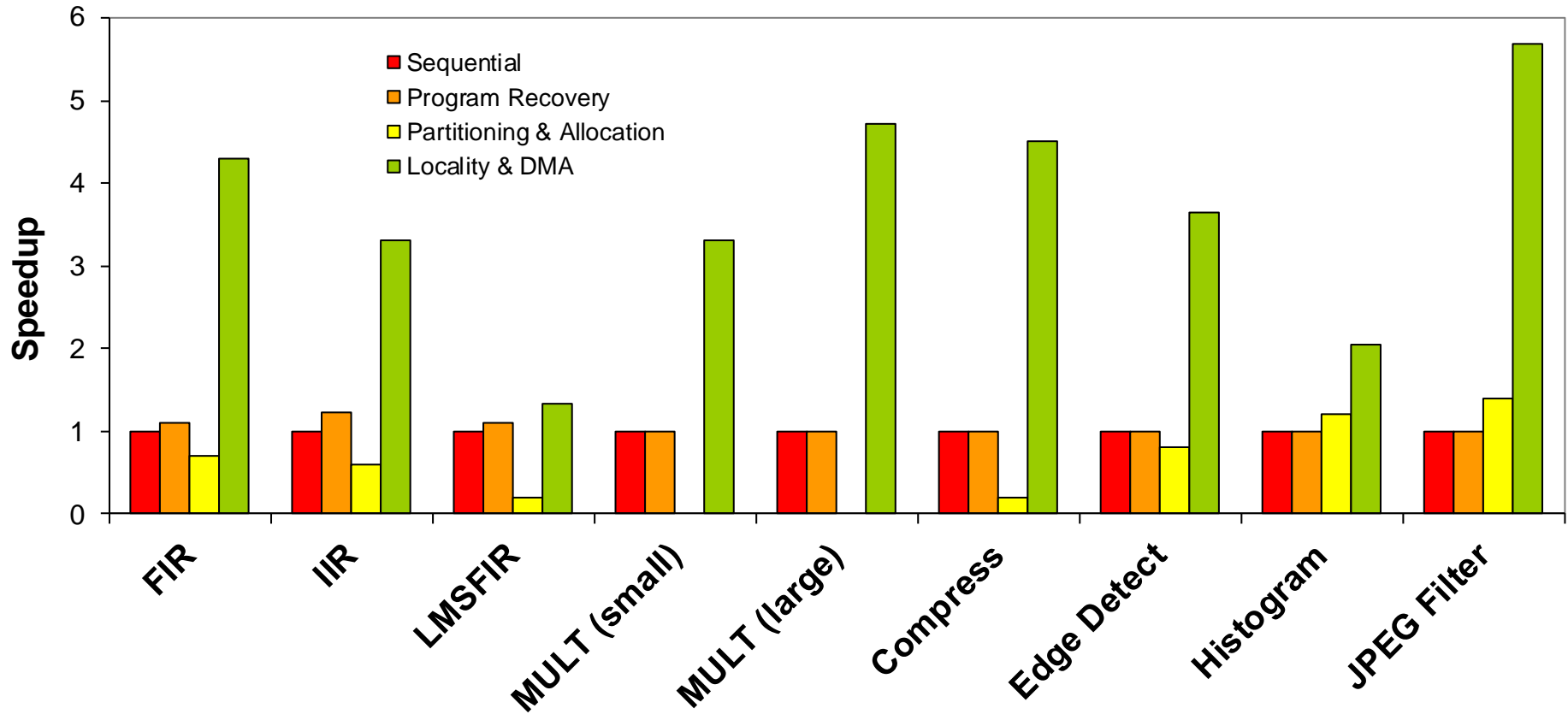
Results: Run-times

- for purely sequential code running on 1 DSP
- for code after program recovery
- for code after data partitioning and memory assignment
- for code after locality improvement & DMA

Results – DSPstone



Results – UTDSP



Discussion

Mean Overall Speedups

- DSPstone: 4.06x
- UTDSP: 3.38x
- All benchmarks: 3.68x

Very Astonishing

- *How is it possible to achieve (mean) speedups of a factor of 4 for DSPstone if parallelization is done for 4 DSPs?*

Reasons for Over-Proportional Speedups

Over-Proportional Speedups > 4 for 4 parallel DSPs

- Parallelized code is more amenable for subsequent compiler optimizations than original sequential code

- *Example:*

Sequential `i`-loop (☞ cf. [Slide 47](#)): 16 iterations

`i`-loop parallelized for 2 DSPs ([Slide 49](#)): 8 iterations

- ☞ Parallelized loops possibly candidates for *Loop Unrolling*:

| | | | |
|---|---------------------------------|---|----------------|
| <code>for (i = 0; i < 8; i++)</code> | <code><loop body>;</code> | } | <i>8 times</i> |
| <code><loop body>;</code> | <code><loop body>;</code> | | |
| | <code>...</code> | | |
| | <code><loop body>;</code> | | |

- Fully unrolled loop without any branches!

- ☞ No delay slots, no potential for branch misprediction

References

Function Specialization

- D. Bacon, S. Graham, O. Sharp. *Compiler Transformations for High-Performance Computing*. ACM Computing Surveys 26(4), 1994.
(Excellent survey paper on compiler optimizations in general!)
- P. Lokuciejewski, H. Falk, H. Theiling. *Influence of Procedure Cloning on WCET Prediction*. CODES+ISSS, Salzburg 2007.

Parallelization for Homogeneous Multi-DSPs

- B. Franke, M. O'Boyle. *A Complete Compiler Approach to Auto-Parallelizing C Programs for Multi-DSP Systems*. IEEE Transactions on Parallel and Distributed Systems 16(3), 2005.

Summary

HIR Optimizations

- Restructuring of Loops
- Restructuring of functions and their calling relationships

Function Specialization / Procedure Cloning

- Specialization of general-purpose functions
- Enable standard optimizations within specialized functions, simplify code for parameter passing
- Impact on ACET marginal, but huge for $WCET_{EST}$

Parallelization for Homogeneous Multi-DSPs

- Focus on exploitation of local memories and address ranges
- Speedups mostly linear in the number of available parallel DSPs