Support of C++ Compiler for Embedded Multicore DSP Systems

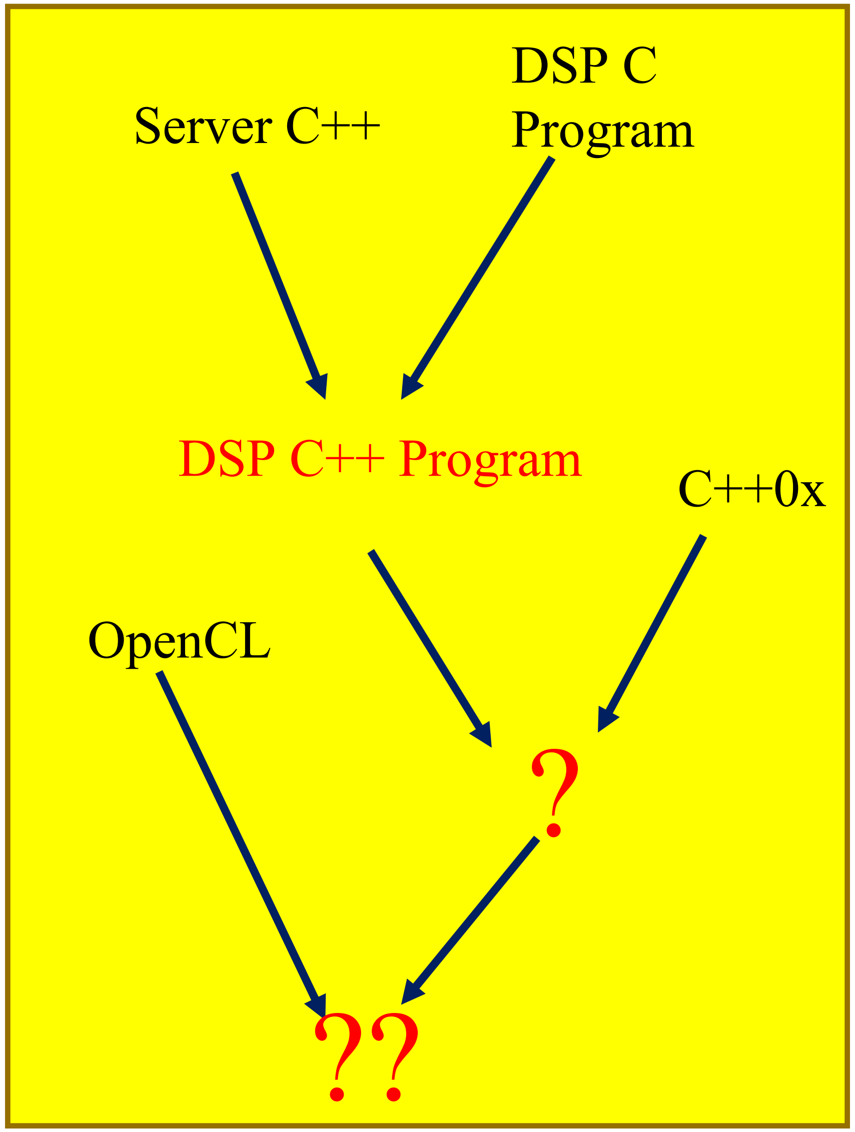
Jenq Kuen Lee Chi-Bang Kuan
Department of Computer Science
National Tsing Hua University
HsinChu, Taiwan

Outline

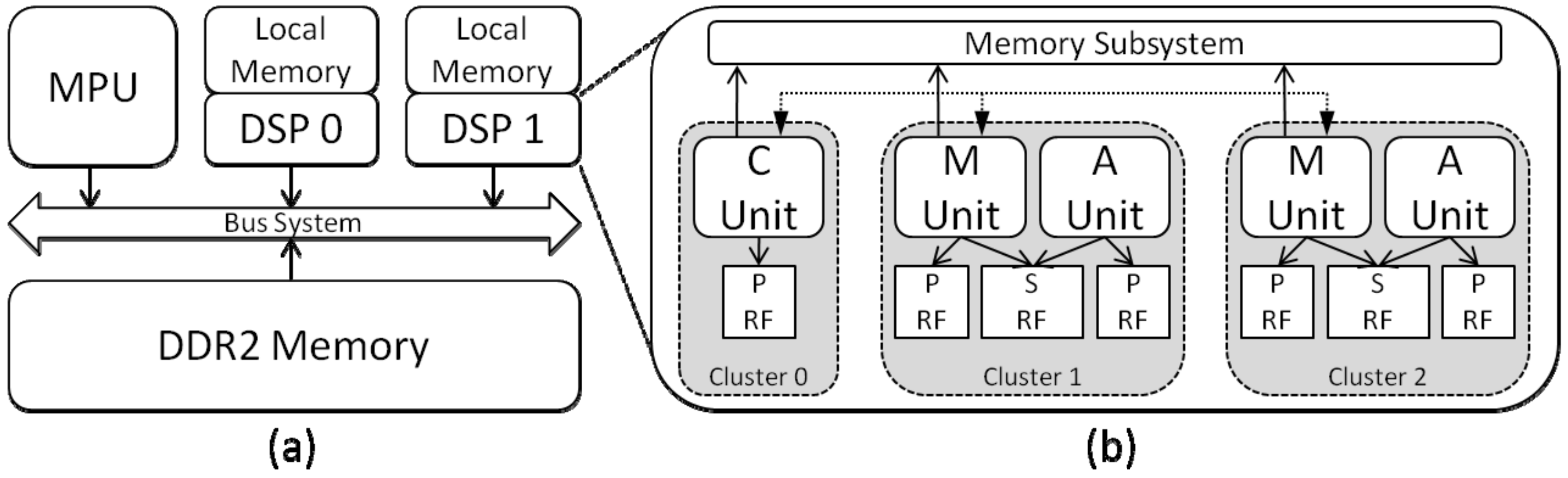
- Motivations
- Case Study: DSP C++ compiler
  - SIMD classes to hide DSP intrinsic functions
  - Code size issues with C++ libraries for DSP systems
  - Design patterns
- Experimental results
- Discussions

Why Supporting C++ for Embedded DSPs?

- Increasing numbers of standards provide C++ interfaces.
  - C++ binding of OpenCV 2.0, some Android codes, C++ legacy codes
- To enhance **programming models** by providing high-level abstraction, such as
  - Overloading vectors for SIMD programming.
- To build **design patterns** for specific applications.
  - Parallel design patterns.
  - Client-server pattern/Pipe-filter pattern/ Map-N-reduce patterns
- C++ in Junction



Case Study for DSP C++ Compiler: Platform

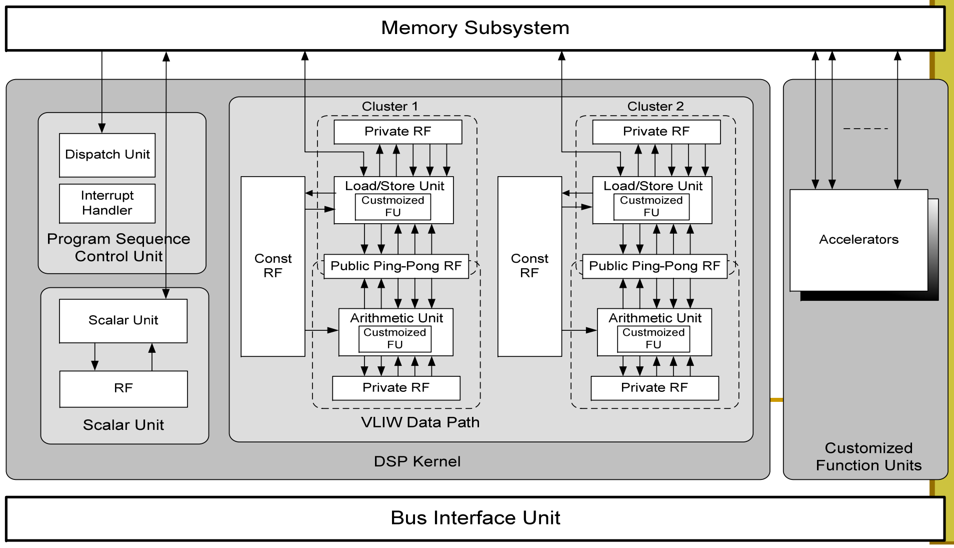


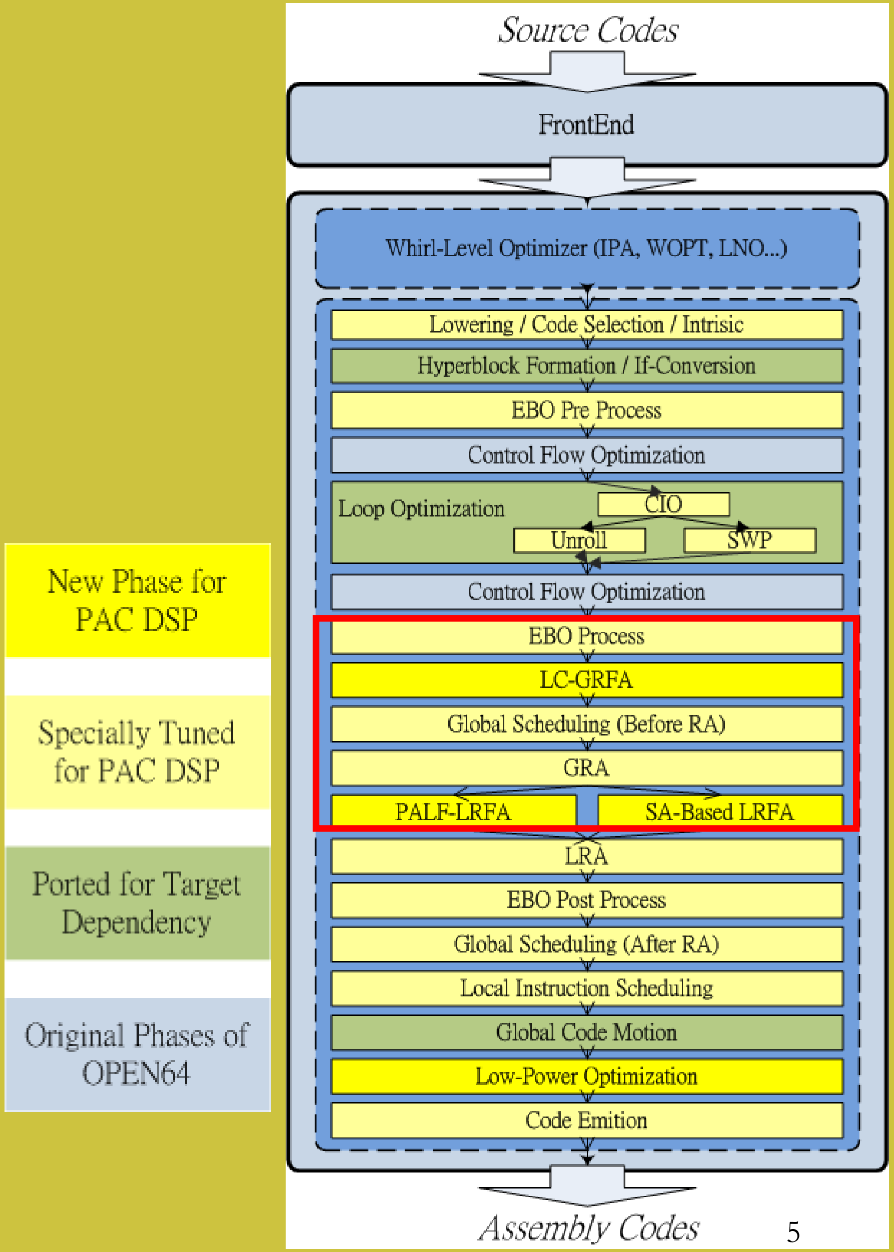
- 3 Cores SoC
  - ARM926EJS x 1
  - PAC DSP x 2
- Shared memory architecture

- ITRI PAC DSP
  - 5-way issue VLIW
  - 3 clustered processing units
  - Distributed register file
  - SIMD instruction set

Case Study: PACDSP Compiler

- Based on Open64 compiler
- VLIW DSP compilers for distributed register files
- PALF scheduling policies for ILP (CPC 2006)
- GRA scheme for distributed register files (CPC 2007)
- SIMD compiler optimizations + intrinsics/extrinsics
- Copy propagations for distributed register architectures (LCPC 2006)
- Register spills among distributed register banks (CPC 2009)

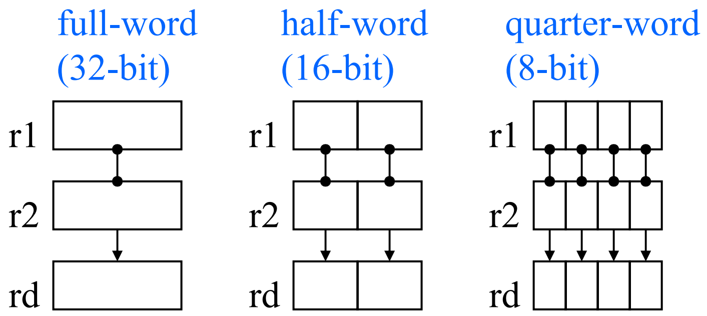



Source Codes
FrontEnd
Whirl-Level Optimizer (IPA, WOPT, LNO...)
Lowering / Code Selection / Intrisic
Hyperblock Formation / If-Conversion
EBO Pre Process
Control Flow Optimization
Loop Optimization
CIO
Unroll
SWP
Control Flow Optimization
EBO Process
LC-GRFA
Global Scheduling (Before RA)
GRA
PALF-LRFA
SA-Based LRFA
LRA
EBO Post Process
Global Scheduling (After RA)
Local Instruction Scheduling
Global Code Motion
Low-Power Optimization
Code Emition
Assembly Codes
New Phase for PAC DSP
Specially Tuned for PAC DSP
Ported for Target Dependency
Original Phases of OPEN64

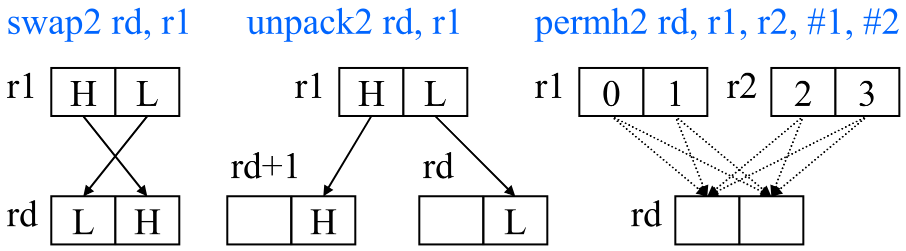
SIMD Features of PACDSP

- Subword instructions
  - short vector instructions that operate data in separated parts

add	→	one 32-bit add
add.d	→	two 16-bit add
add.q	→	four 8-bit add



- Permutation instructions
  - diverse instructions for reordering data in registers



- Parallel instruction issuing
  - issue up to two-multiply or five-other operations per cycle

{ Scalar , LSU , ALU , LSU , ALU }
{ nop , nop , mpy , nop , mpy }
{ add , add , add , add , add }

SIMD Intrinsics for PACDSP

- Summary of SIMD intrinsics

intrinsics	***subdivisions***	***instances in prototypes***
subword intrinsics	full-word	int __builtin_add (int, int) …
	half-word	int __builtin_add_d (int, int) …
	quarter-word	int __builtin_add_q (int, int) …
cluster intrinsics	cluster 1	int __builtin_c1_add (int, int) int __builtin_c1_add_d (int, int) int __builtin_c1_add_q (int, int) …
	cluster 2	int __builtin_c2_add (int, int) …

- All intrinsics are mapped to DSP instructions
  - subword intrinsics

    __builtin_add_d → add.d (a dual 16-bit add)
    __builtin_add_q → add.q (a quad 8-bit add) …
  - cluster intrinsics (*also have subword semantics*)

    __builtin_c1_add_d → add.d (a dual 16-bit add at cluster 1)
    __builtin_c2_add_d → add.d (a dual 16-bit add at cluster 2) …

SIMD Intrinsic Programming: The LMS Filter in DSPstone

1.1 identify parallel regions or parallelize programs

```

    short H[N], X[N];
    short *p_H, *p_X, *p_X2;
    int f, y=0, d=7, delta=1, error;

    /* state update and dot-product */
    for(f = 1; f < N; f++)
      y += H[N-f] * (X[N-f] = X[N-f-1]);
    y += H[0] * (X[0] = x);

    /* coefficient update */
    error = (d - y) * delta;
    for (f = 0; f < N; f++)
      H[f] += error * X[f];

```

Quick summary:

1. The data type of input arrays is 16-bit integer
2. There are two loops, one for dot-product and one for coefficient-update
3. The dot-product is not parallel, while the coefficient-update is parallel.

SIMD Intrinsic Programming: The LMS Filter with SIMD intrinsics

1. find the best way to process data

```
short __attribute__ ((aligned(4))) H[N];
short __attribute__ ((aligned(4))) X[N];
int *vX = (int *)X, *vH = (int *)H;
int v_err=0, sum1=0, sum2=0, sum3=0, sum4=0;

for(f = 1, X[-1] = x; f < (N/2)+1; f+=4)
{
  vX[(N/2)-f] =  __builtin_c1_permh2(vX[(N/2)-f-1], vX[(N/2)-f], 1, 2);
  sum1 = __builtin_c1_mac_d(sum1, vH[(N/2)-f], vX[(N/2)-f]);
  vX[(N/2)-f-1] = __builtin_c1_permh2(vX[(N/2)-f-2], vX[(N/2)-f-1], 1, 2);
  sum2 = __builtin_c1_mac_d(sum2, vH[(N/2)-f-1], vX[(N/2)-f-1]);

  vX[(N/2)-f-2] = __builtin_c2_permh2(vX[(N/2)-f-3], vX[(N/2)-f-2], 1, 2);
  sum3 = __builtin_c2_mac_d(sum3, vH[(N/2)-f-2], vX[(N/2)-f-2]);
  vX[(N/2)-f-3] = __builtin_c2_permh2(vX[(N/2)-f-4], vX[(N/2)-f-3], 1, 2);
  sum4 = __builtin_c2_mac_d(sum4, vH[(N/2)-f-3], vX[(N/2)-f-3]);
}
y = __builtin_c1_mergea(__builtin_c1_add_d(sum1, sum2)) +
    __builtin_c2_mergea(__builtin_c2_add_d(sum3, sum4));

error = (d - y) * delta;
v_err= __builtin_permh2(error, error, 0, 0);
for (f = 0; f < (N/2); f+=2)
{
  vH[f] = __builtin_c1_mac_d(vH[f], vX[f], v_err);
  vH[f+1] = __builtin_c2_mac_d(vH[f+1], vX[f+1], v_err);
}
```

SIMD Intrinsic Programming: The LMS Filter with SIMD intrinsics

2. prepare packed data for subword processing

```
short __attribute__ ((aligned(4))) H[N];
short __attribute__ ((aligned(4))) X[N];
int *vX = (int *)X, *vH = (int *)H;
int v_err=0, sum1=0, sum2=0, sum3=0, sum4=0;

for(f = 1, X[-1] = x; f < (N/2)+1; f+=4)
{
  vX[(N/2)-f] =  __builtin_c1_permh2(vX[(N/2)-f-1], vX[(N/2)-f], 1, 2);
  sum1 = __builtin_c1_mac_d(sum1, vH[(N/2)-f], vX[(N/2)-f]);
  vX[(N/2)-f-1] = __builtin_c1_permh2(vX[(N/2)-f-2], vX[(N/2)-f-1], 1, 2);
  sum2 = __builtin_c1_mac_d(sum2, vH[(N/2)-f-1], vX[(N/2)-f-1]);

  vX[(N/2)-f-2] = __builtin_c2_permh2(vX[(N/2)-f-3], vX[(N/2)-f-2], 1, 2);
  sum3 = __builtin_c2_mac_d(sum3, vH[(N/2)-f-2], vX[(N/2)-f-2]);
  vX[(N/2)-f-3] = __builtin_c2_permh2(vX[(N/2)-f-4], vX[(N/2)-f-3], 1, 2);
  sum4 = __builtin_c2_mac_d(sum4, vH[(N/2)-f-3], vX[(N/2)-f-3]);
}
y = __builtin_c1_mergea(__builtin_c1_add_d(sum1, sum2)) +
    __builtin_c2_mergea(__builtin_c2_add_d(sum3, sum4));

error = (d - y) * delta;
v_err= __builtin_permh2(error, error, 0, 0);
for (f = 0; f < (N/2); f+=2)
{
  vH[f] = __builtin_c1_mac_d(vH[f], vX[f], v_err);
  vH[f+1] = __builtin_c2_mac_d(vH[f+1], vX[f+1], v_err);
}
```

SIMD Intrinsic Programming: The LMS Filter with SIMD intrinsics

3. utilize subword instructions and distribute computation

```
short __attribute__ ((aligned(4))) H[N];
short __attribute__ ((aligned(4))) X[N];
int *vX = (int *)X, *vH = (int *)H;
int v_err=0, sum1=0, sum2=0, sum3=0, sum4=0;

for(f = 1, X[-1] = x; f < (N/2)+1; f+=4)
{
  vX[(N/2)-f] =  __builtin_c1_permh2(vX[(N/2)-f-1], vX[(N/2)-f], 1, 2);
  sum1 = __builtin_c1_mac_d(sum1, vH[(N/2)-f], vX[(N/2)-f]);
  vX[(N/2)-f-1] = __builtin_c1_permh2(vX[(N/2)-f-2], vX[(N/2)-f-1], 1, 2);
  sum2 = __builtin_c1_mac_d(sum2, vH[(N/2)-f-1], vX[(N/2)-f-1]);

  vX[(N/2)-f-2] = __builtin_c2_permh2(vX[(N/2)-f-3], vX[(N/2)-f-2], 1, 2);
  sum3 = __builtin_c2_mac_d(sum3, vH[(N/2)-f-2], vX[(N/2)-f-2]);
  vX[(N/2)-f-3] = __builtin_c2_permh2(vX[(N/2)-f-4], vX[(N/2)-f-3], 1, 2);
  sum4 = __builtin_c2_mac_d(sum4, vH[(N/2)-f-3], vX[(N/2)-f-3]);
}
y = __builtin_c1_mergea(__builtin_c1_add_d(sum1, sum2)) +
    __builtin_c2_mergea(__builtin_c2_add_d(sum3, sum4));

error = (d - y) * delta;
v_err= __builtin_permh2(error, error, 0, 0);
for (f = 0; f < (N/2); f+=2)
{
  vH[f] = __builtin_c1_mac_d(vH[f], vX[f], v_err);
  vH[f+1] = __builtin_c2_mac_d(vH[f+1], vX[f+1], v_err);
}
```

SIMD Programming with C++

Intrinsic programming is effective, though it makes code not portable and hard to read. Any solution?

- Use C++ classes to encapsulate SIMD details in vector data types.

```
class short4
{
   int subword1;
   int subword2;

public:
   ...
};
```

```
inline short4 operator+(
     const short4& opnd1, const short4& opnd2)
{
   short4 result(
     _dsp_c1_add_d(opnd1.subword1, opnd2.subword1),
     _dsp_c2_add_d(opnd1.subword2, opnd2.subword2));
   return result;
}                        /* C++ operator overloading */
```

- Seamless integration with SIMD intrinsics to provide vector abstraction.
- Better portability and readability with similar performance (see examples and performance in next slides).

More C++ Classes for DSPs

- **Emulated long vectors** with cluster intrinsics to encourage programmers to utilize VLIW data paths.
  - E.g. emulated **short4** types for 32-bit DSP processors
  - Implicit computation distribution over VLIW clusters.
- **Miscellaneous data types** to facilitate signal processing
  - Integers with saturation arithmetic.
  - Fractional numbers with different rounding modes.
  - Complex numbers.

	Vector Types	Generic Vector Operations
Native subword vectors	[u]char4, [u]short2	arithmetic: +, -, *, >>, <<, … reduction: sum, dot_product, min, max, … load/store: lw, sw access: [ ] permutation: perm, pack, unpack
Emulated full-word vectors	[u]char8, [u]short4, [u]int2, [u]int4, cmplx, cmplx2	

Convolution Kernels with SIMD

intrinsics vs. vector classes

```
/* Convolution with SIMD Intrinsics */

ushort coef [BLK_SZ];

int convolve ( uchar src[BLK_SZ*4] ) {
    int ycbcr, y, cbcr, COEF, Y, CBCR;
    __m64 tmp64;

    for( int i=0; i<BLK_SZ; i++ ) {
        ycbcr = ((int*)src)[i];

        tmp64 = _dsp_unpack4u(ycbcr);
        y      = _dsp_get64l (tmp64);
        cbcr   = _dsp_get64h(tmp64);

        COEF = _dsp_permh2(coef[i], 0, 0, 0);
        Y       = _dsp_mac_d(Y, y, COEF);
        CBCR = _dsp_mac_d(CBCR, cbcr, COEF);
    }
    return a pixel result with Y and CBCR;
}
```

- direct access to DSP instructions
- platform-dependent
- expose users to hardware details

- compact representation
- with similar performance

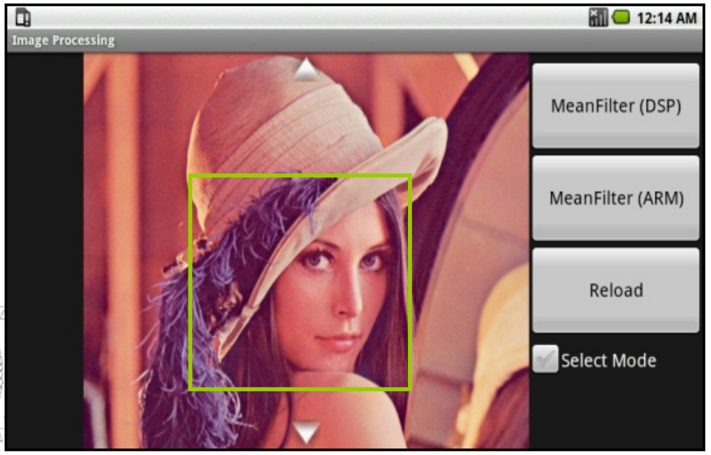
```
/* Convolution with SIMD Vectors */

ushort coef [BLK_SZ];

int convolve ( uchar src[BLK_SZ*4] ) {
    uchar4 ycbcr;
    ushort2 y, cbcr, CBCR, Y;

    for( int i=0; i<BLK_SZ; i++ ) {
        // load a char4 vector from &src[i]
        ycbcr.lw(&src[i*4]);

        // unpack a uchar4 into two ushort2
        ycbcr.unpack(y, cbcr);

        // vector operations on short2 vectors
        Y      += y * coef[i];
        CBCR += cbcr * coef[i];
    }
    return a pixel result with Y and CBCR;
}
```

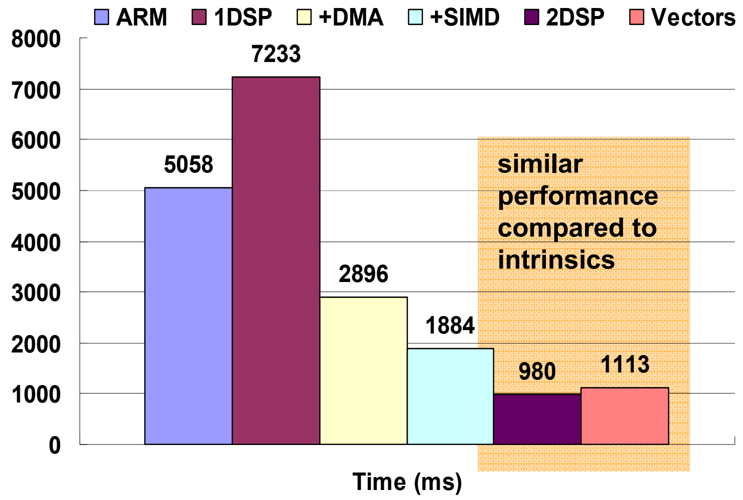
- Green functions are subword intrinsics for accessing DSP instructions.
- Blue data types are SIMD vectors; red operators are vector operations.

Applications with SIMD Vectors

- **Image blurring**
  - Blurring an arbitrarily selected block and displaying results via Android UI.
- Major kernels:
  - Color space transformations (rgb2yuv, yuv2rgb).
  - 2D convolution with a 8x8 coefficient matrix (prev page).
- Performance footnotes:
  - ARM: single-core program with ARM GCC -O3.
  - +DMA: DMAs are used to facilitate data transmission (2D data) between shared memory and DSP locals.
  - +SIMD: DSP subword instructions are used via SIMD intrinsics.
  - Vectors: SIMD intrinsics are replaced with C++ vector operations.

Fig. Blurred results.



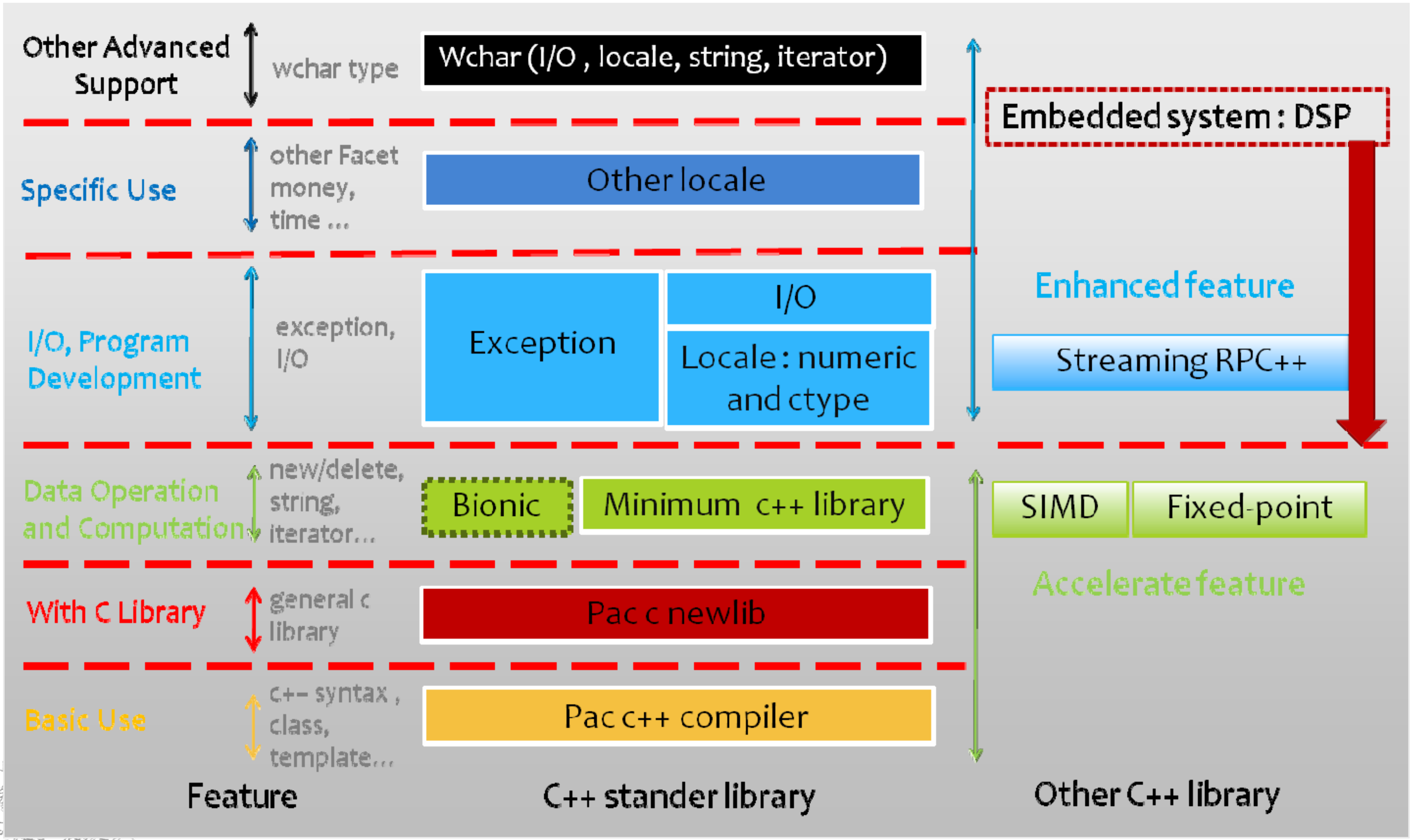


Code Size Issues of C++ Library

- Embedded processors typically have small instruction memory
  - E.g. PACDSP with 16KB~32KB i-cache in cache or SPM mode.
- Size analysis of C++ library
  - Apache C++ library compiled with -O2 and -Os.
  - Apparently, some of the C++ components are inappropriate for embedded.
- C++ also depends on partial C library for basic functions.
  - The overall lib stack is heavy.
  - Doesn't fit instruction cache, or cache miss penalty hurts.

Component	Code Size	Percentage
Language	4 KB	0.3%
Diagnostics	10 KB	1.0%
Utilities	3 KB	0.3%
Strings	63 KB	6.0%
Localizations	654 KB	62.4%
Containers	4 KB	0.3%
Iterators	48 KB	4.6%
Algorithms	0 KB	0.0%
Numerics	2 KB	0.2%
Input/Output	259 KB	24.7%
Total	1047 KB	100.0%

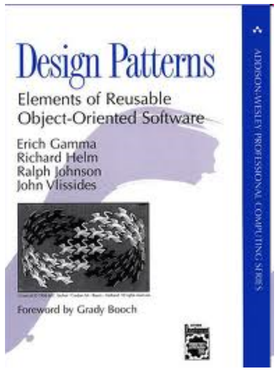
DSP C/C++ Library Stack and Layers



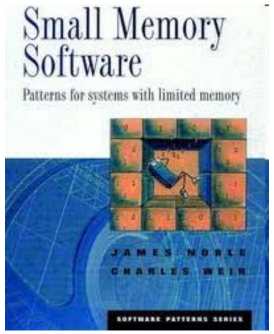
Trade-off between programmability, performance and code size.

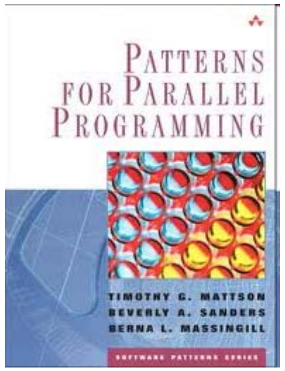
Design Patterns

- Brief introduction to design patterns
  - Proposed by C. Alexander for city planning and architecture.
  - Introduced to software engineering by Beck and Cunningham.
  - Become prominent in object-oriented programming by GoF.


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- Design patterns describe “good solutions” to recurring problems in a particular context.
  - Patterns for **object-oriented programming**
    - Creational patterns, Structural patterns, Behavioral patterns, etc.
  - Patterns for **limited memory systems**
    - Compression, Small data structures, Memory allocation, etc.
  - Patterns for **parallel programming**
    - Finding concurrency, Algorithm structure, Supporting structures and Implementation mechanisms.





Parallel Design Patterns

Parallelization can be a process to transform problems to programs by selecting appropriate patterns.

- Finding Concurrency

  ⇩ decomposition of problems

- Algorithm Structure

  ⇩ appropriate algorithms

- Supporting Structures

  ⇩ appropriate program constructs

- Implementation Mechanisms

  ⇩ parallelized programs

Design Space of Parallel Patterns

- **Decomposition** patterns: {data, task}
- **Dependency analysis** patterns: {group tasks, order tasks, data sharing}
- **Design evaluation** pattern

How the given problem is organized?

- By **tasks**: {task parallelism, divide & conquer}
- By **data decomposition**: {geometric, recursive}
- By **flow of data**: {pipeline, event-based coord.}

Software constructs to express parallel algorithms

- **Program structures**: {SPMD, master/worker, loop parallelism, fork-join, client-server, SIMD}
- **Data structures**: {shared data, shared queue, distributed array}

- UE management: {thread/process creation/destr.}
- Synchronization: {barrier, mutex, mem fence}
- Communication: {msg passing, collective comm}

These patterns are summarized from the book, "Patterns for Parallel Programming" by Mattson et al

Multicore Programming Design Patterns with C++

- Client-server pattern and data stream abstraction for hiding remote task creation and channel manipulation details.
- Well-defined interface instead of APIs taking function pointers.

```
/* RPC-client program */
class task : public srpc::call_template
{
      CALL_INIT(task);
      void TX() {
         istrm << shape(len, b, s)
             << INPUT << flush;
      }
      void RX() {
         ostrm >> shape(len)
             >> RESULT >> pop;
      }
};

task task_call("task", DSP0);
srpc::srpc_call<task> call(task_call);
```

```
/* RPC-server program */
class task : public srpc::service_template
{
      SERVICE_INIT(task);
      void SERVE() {
         // compute data with istrm.
         // return results via ostrm.
      }
};

int main()
{
      // register a "task" service
      task task_service("task");
      srpc::srpc_server<task>
               server(task_service);
}
```

Multicore Programming Design Patterns with C++

- Client-server pattern and data stream abstraction for hiding remote task creation and channel manipulation details.
  Well-defined interface i ng function pointers.

User-defined functor which implements remote call interface

Client interface which defines essential routines & data structures for RPC

```
/* RPC-client program */
class task : public srpc::call_template
{
    CALL_INIT(task);
    void TX() {
      istrm << shape(len, b, s)
         << INPUT << flush;
    }
    void RX() {
      ostrm >> shape(len)
         >> RESULT >> pop;
    }
};

task task_call("task", DSP0);
srpc::srpc_call<task> call(task_call);
```

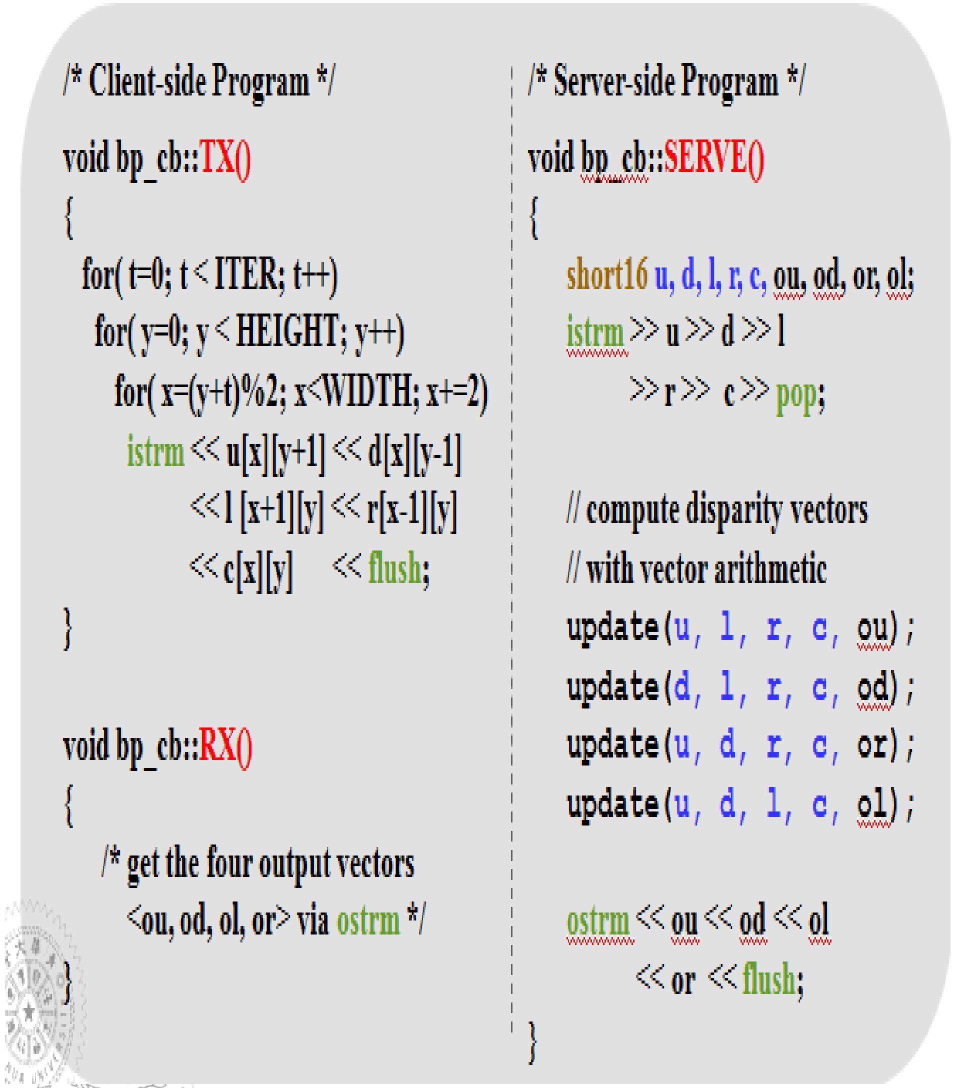
```
/* RPC-server program */
class task : public srpc::service_template
{
    SERVICE_INIT(task);
    void SERVE() {
      // compute data with istrm.
      // return results via ostrm.

int main()
{
    // register a "task" service
    task task_service("task");
    srpc::srpc_server<task>
          server(task_service);
}
```

Encapsulated I/O streams with operators and manipulators to accommodate various data formats

All RPC invocation details are hidden in srpc::srpc_call

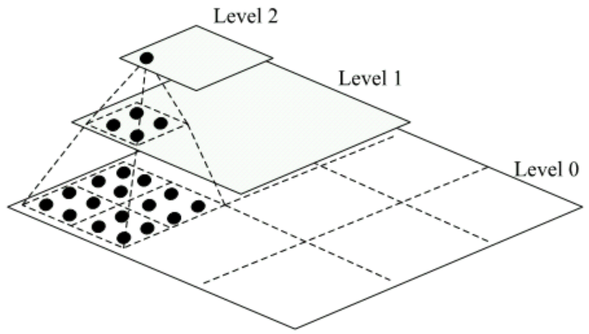
Program Parallelization

- Belief Propagation (BP) method for Stereo Vision

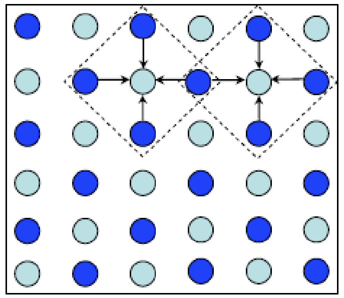


```
/* Client-side Program */

void bp_cb::TX()
{
  for( t=0; t < ITER; t++)
   for( y=0; y < HEIGHT; y++)
    for( x=(y+t)%2; x<WIDTH; x+=2)
     istrm << u[x][y+1] << d[x][y-1]
         << l [x+1][y] << r[x-1][y]
         << c[x][y]    << flush;
}

void bp_cb::RX()
{
   /* get the four output vectors
     <ou, od, ol, or> via ostrm */
}
```

```
/* Server-side Program */

void bp_cb::SERVE()
{
    short16 u, d, l, r, c, ou, od, or, ol;
    istrm >> u >> d >> l
         >> r >> c >> pop;

    // compute disparity vectors
    // with vector arithmetic
    update(u, l, r, c, ou);
    update(d, l, r, c, od);
    update(u, d, r, c, or);
    update(u, d, l, c, ol);

    ostrm << ou << od << ol
         << or << flush;
}
```



belief propagation over multi-scale planes

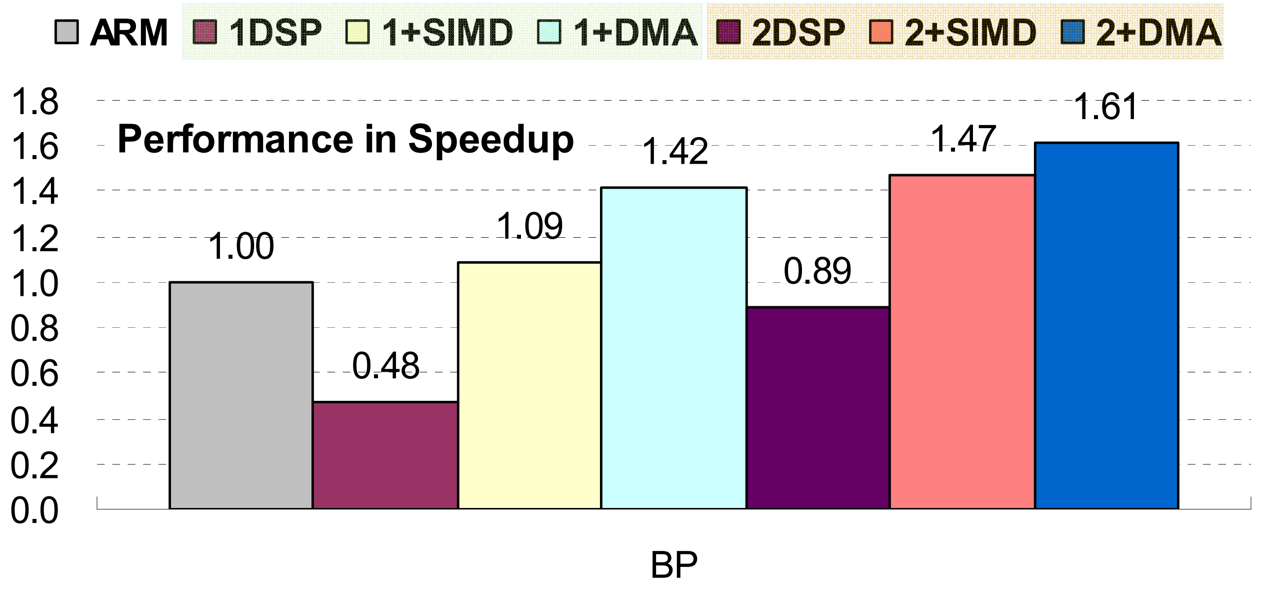


message update by neighboring pixels

Finding Concurrency	■ Data decomposition ■ Data sharing (read-only)
Algorithm Structures	■ Geometric decomposition (chunk: tuple<u,d,l,r,c>)
Supporting Structures	■ Client-server (msg-update) ■ SIMD (disparity vectors)
Implementation Mechanisms	■ RPC ■ Data streaming

Experimental Results on BP

- About the performance with +SIMD
  - SIMD vectors are employed to update disparity vectors.
  - Besides SIMD, few branches are eliminated by MIN intrinsics.
- About the performance with +DMA
  - DMAs on the multi-DSP system are incorporated to facilitate data streaming, implicitly in stream implementation.
- About the performance degradation (0.48 and 0.89)
  - MPU waiting for results from DSPs (especially serious for 1DSP).
  - Data communication and RPC overhead.



OpenCL Compilation Flow for PAC Duo

***Work-item Coalescing**

Serialize work-items to avoid context switching.

```
gid = get_global_id(0);
C[gid] = A[gid] + B[gid];
```

Cluster-aware Work-item Dispatching

Transform each statement with **cluster intrinsic** to explicitly dispatch work-items to clusters.

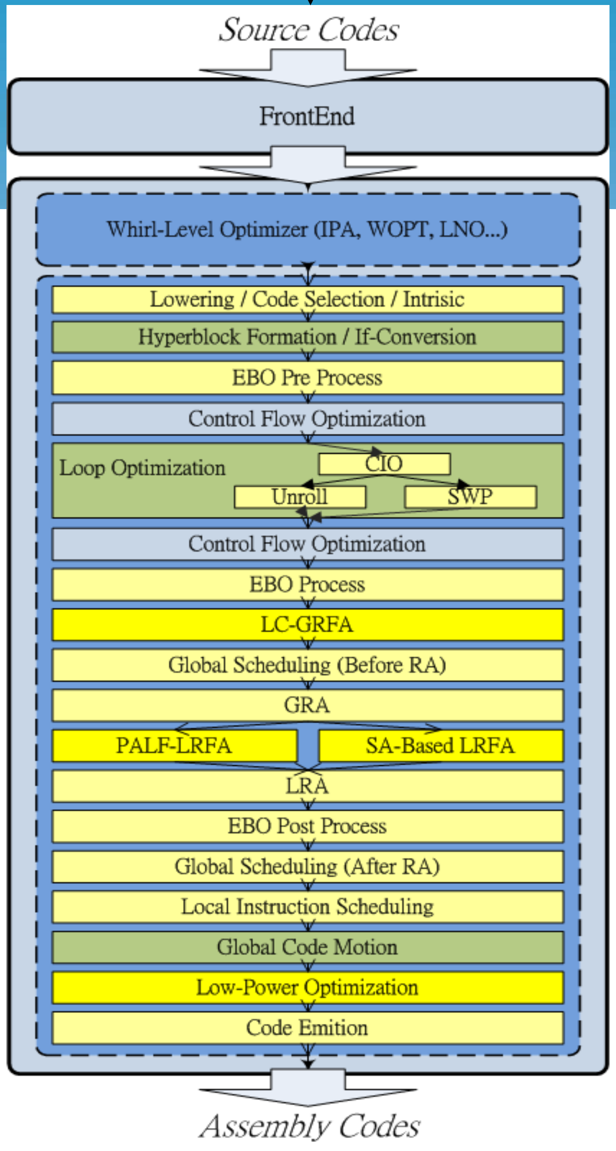
```
for (i=0; i<4; i++) {
  gid = get_global_id(0);
  C[gid] = A[gid] + B[gid];
}
```

```
for (i=0; i<4; i+=2) {
  gid  = get_global_id(0);
  gid2 = get_global_id(0) + 1;
  C[gid]  = __builtin_c1_add(A[gid],  B[gid]);
  C[gid2] = __builtin_c2_add(A[gid2], B[gid2]);
}
```

Source-to-source Transformation:

*Work-item Coalescing: Clang

Cluster-aware Work-item Dispatching: *Cetus


Source Codes
FrontEnd
Whirl-Level Optimizer (IPA, WOPT, LNO...)
Lowering / Code Selection / Intrisic
Hyperblock Formation / If-Conversion
EBO Pre Process
Control Flow Optimization
Loop Optimization
CIO
Unroll
SWP
Control Flow Optimization
EBO Process
LC-GRFA
Global Scheduling (Before RA)
GRA
PALF-LRFA
SA-Based LRFA
LRA
EBO Post Process
Global Scheduling (After RA)
Local Instruction Scheduling
Global Code Motion
Low-Power Optimization
Code Emition
Assembly Codes

"OpenCL Update", Yu-Te Lin, Shao-Ching Wang, Jia-Jer Li, Jenq-Kuen Lee, Open64 Developers Forum, Cupertino, USA, June 2011.

* J. Lee, et al, "An OpenCL Framework for Heterogeneous Multicores with Local Memory," PACT'10

*S. Lee, et al, "Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation," LCPC'03

Discussions: Emerging Issues C++0x and OpenCL/C++

- C++0x: Lambda functions allow programmers to define functions “directly” in places where their expressions are used.
  - Similar to **anonymous classes** in Java.
  - Better ways to provide a syntax sugar for parallel design patterns

```
typedef pair IntPair      /* A sorting example with lambdas */
std::vector v;

// sort v by the second element of the pair
std:sort(v.begin(), v.end(),
        [](const IntPair &a, const IntPair &b) // call by reference
        { return a.second < b.second; }       // return value
);
```

//Source: Sorting example is referenced with Intel Software Blogs. Author: Stefanus Du Toit (Intel)

- C++0x vs. OpenCL:
  - C++0x thread model needs compilers for cares not to introduce data racing in optimizations.
  - C++0x thread model is suitable for OpenCL program with host, but might be a competing linguistic with kernel programs.

Summary

- We present techniques in supporting C++ compiler and programming models for multi-core DSP systems.
- C++ SIMD vector class and overloading can be used to hide intrinsic functions of C programs for better abstractions.
- C++ program can help build parallel design patterns.
- Program parallelization with parallel design patterns on stereo-vision and image-processing applications are presented with a multi-DSP platform as a case study.
- Issues ahead when the embedded system meets C++0x and embedded OpenCL meets C++0x.
- Related references can be seen in http://www.cs.nthu.edu.tw/~jklee