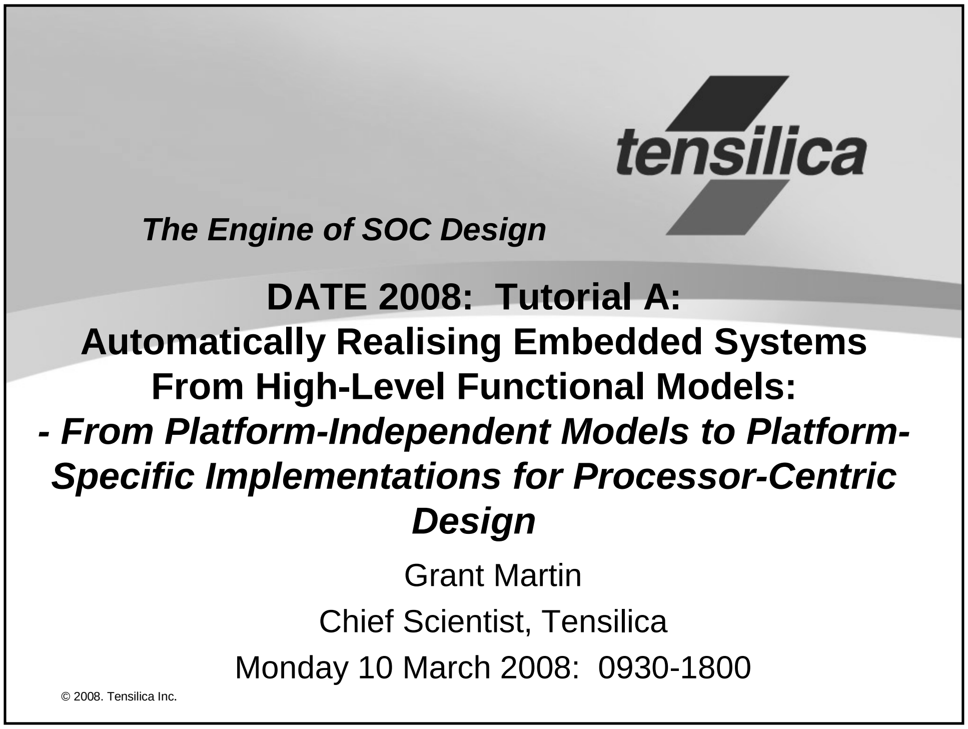


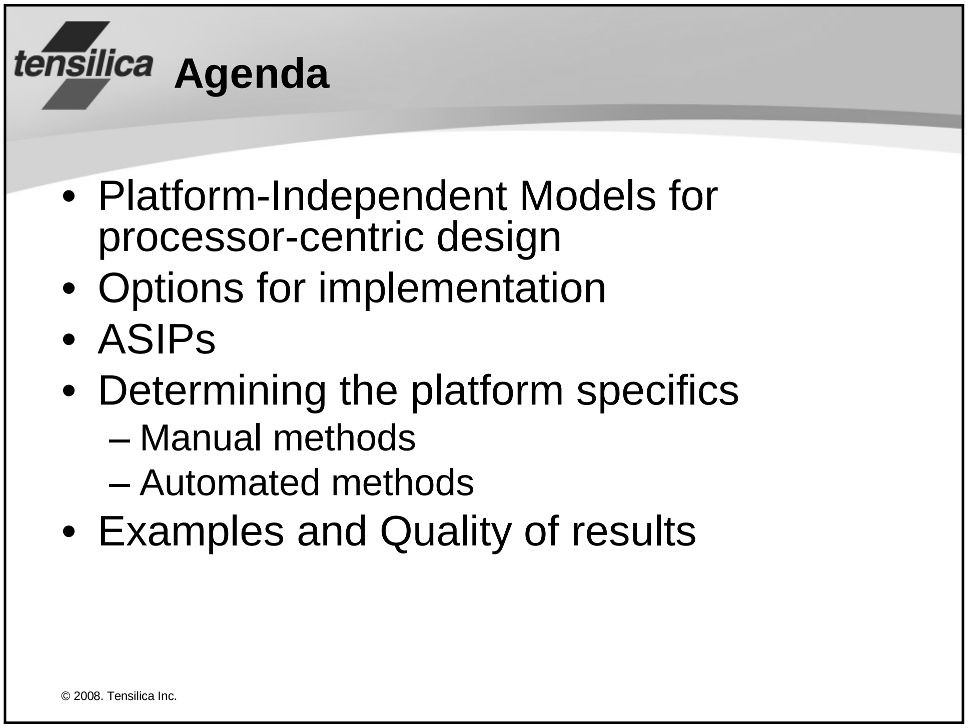
*The Engine of SOC Design*

# DATE 2008: Tutorial A: Automatically Realising Embedded Systems From High-Level Functional Models: *- From Platform-Independent Models to Platform-Specific Implementations for Processor-Centric Design*

Grant Martin

Chief Scientist, Tensilica

Monday 10 March 2008: 0930-1800



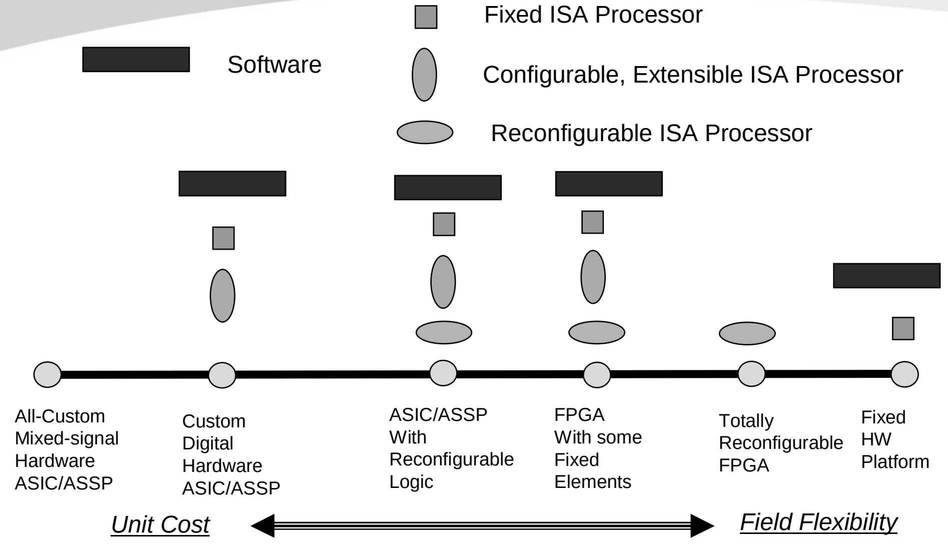
## Agenda

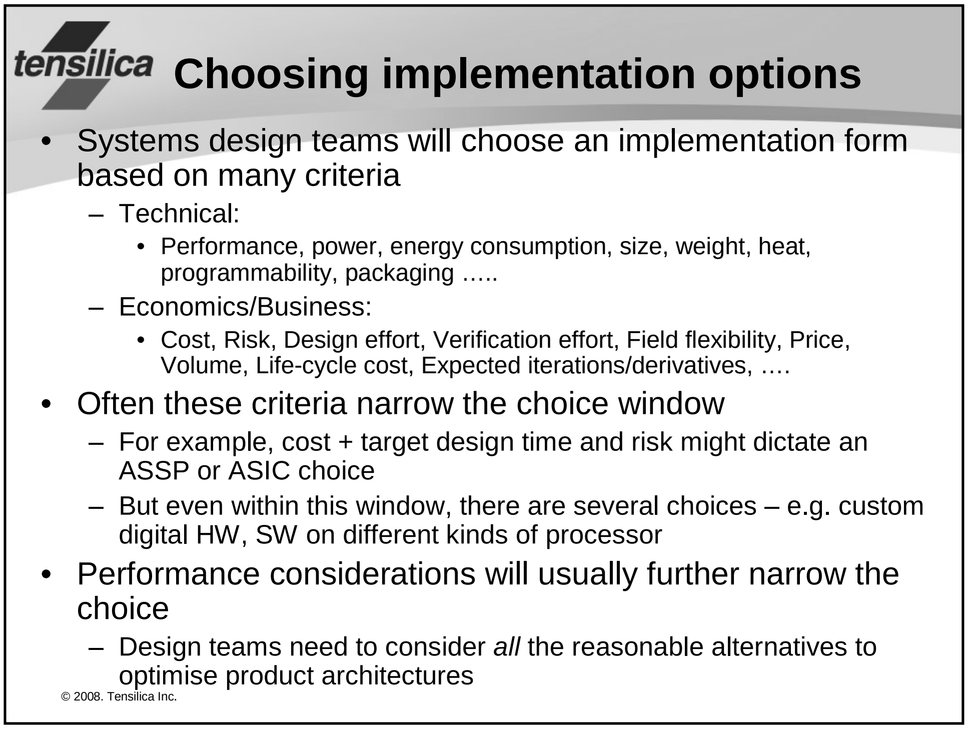
- Platform-Independent Models for processor-centric design
- Options for implementation
- ASIPs
- Determining the platform specifics
  - Manual methods
  - Automated methods
- Examples and Quality of results

## Platform-Independent Models for processor-centric design

- Usually algorithmic or control code in C, C++, possibly SystemC (e.g. C/C++ with SystemC interfaces) or other C dialect
- Might be generated from higher level specification: e.g. SDL, a UML profile, Mathworks Matlab/Simulink
- Such models are as “generic as they come”
  - Will compile and execute on just about any host platform
  - Usually will compile and execute (possibly poorly) on embedded processors of interest
- How does a design team decide what implementation platform to map this model to?

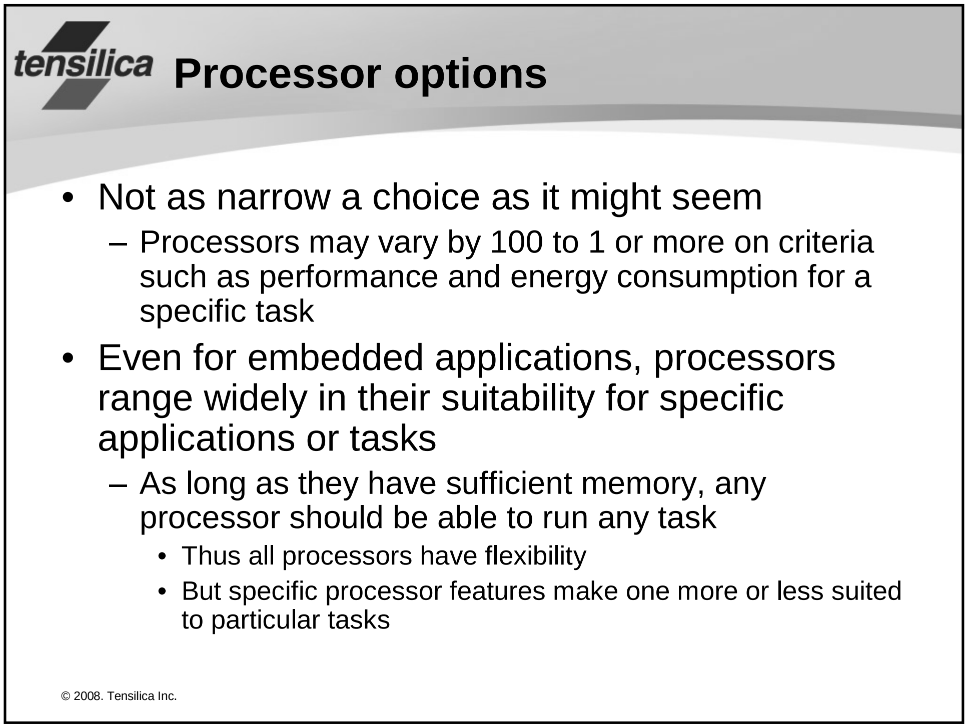
## The “lumpy continuum” for functional implementation


Software
Fixed ISA Processor
Configurable, Extensible ISA Processor
Reconfigurable ISA Processor
All-Custom Mixed-signal Hardware ASIC/ASSP
Custom Digital Hardware ASIC/ASSP
ASIC/ASSP With Reconfigurable Logic
FPGA With some Fixed Elements
Totally Reconfigurable FPGA
Fixed HW Platform
Unit Cost
Field Flexibility



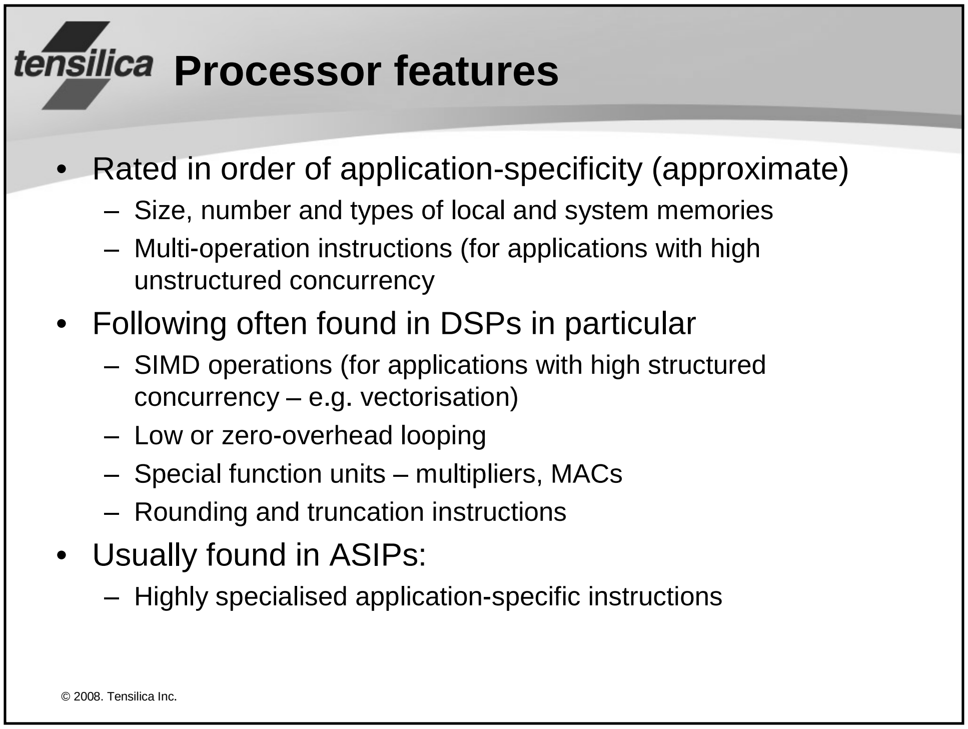
## Choosing implementation options

- Systems design teams will choose an implementation form based on many criteria
  - Technical:
    - Performance, power, energy consumption, size, weight, heat, programmability, packaging …..
  - Economics/Business:
    - Cost, Risk, Design effort, Verification effort, Field flexibility, Price, Volume, Life-cycle cost, Expected iterations/derivatives, ….
- Often these criteria narrow the choice window
  - For example, cost + target design time and risk might dictate an ASSP or ASIC choice
  - But even within this window, there are several choices – e.g. custom digital HW, SW on different kinds of processor
- Performance considerations will usually further narrow the choice
  - Design teams need to consider *all* the reasonable alternatives to optimise product architectures



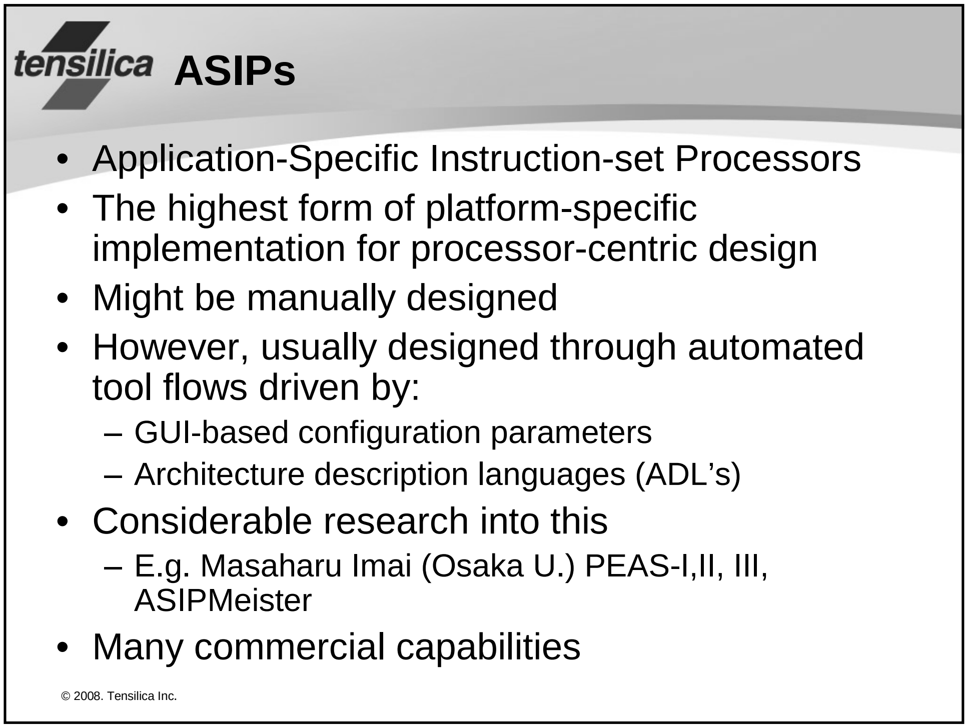
## Processor options

- Not as narrow a choice as it might seem
  - Processors may vary by 100 to 1 or more on criteria such as performance and energy consumption for a specific task
- Even for embedded applications, processors range widely in their suitability for specific applications or tasks
  - As long as they have sufficient memory, any processor should be able to run any task
    - Thus all processors have flexibility
    - But specific processor features make one more or less suited to particular tasks



## Processor features

- Rated in order of application-specificity (approximate)
  - Size, number and types of local and system memories
  - Multi-operation instructions (for applications with high unstructured concurrency
- Following often found in DSPs in particular
  - SIMD operations (for applications with high structured concurrency – e.g. vectorisation)
  - Low or zero-overhead looping
  - Special function units – multipliers, MACs
  - Rounding and truncation instructions
- Usually found in ASIPs:
  - Highly specialised application-specific instructions



## ASIPs

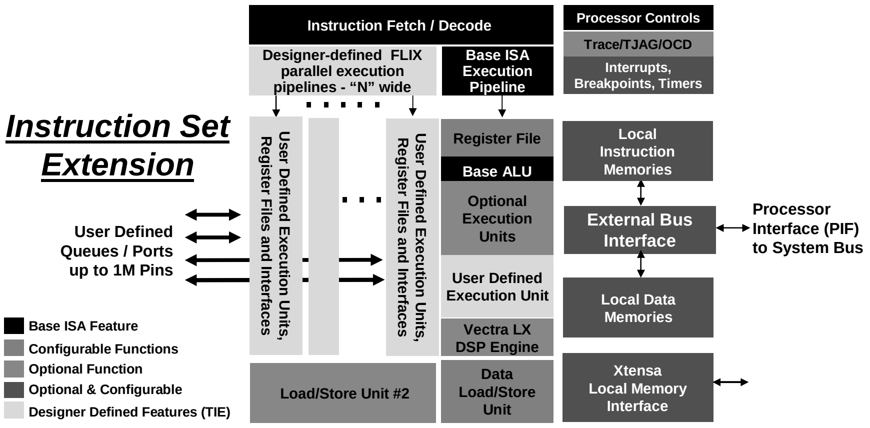
- Application-Specific Instruction-set Processors
- The highest form of platform-specific implementation for processor-centric design
- Might be manually designed
- However, usually designed through automated tool flows driven by:
  - GUI-based configuration parameters
  - Architecture description languages (ADL’s)
- Considerable research into this
  - E.g. Masaharu Imai (Osaka U.) PEAS-I,II, III, ASIPMeister
- Many commercial capabilities

## Architecture-Description Language ASIPs

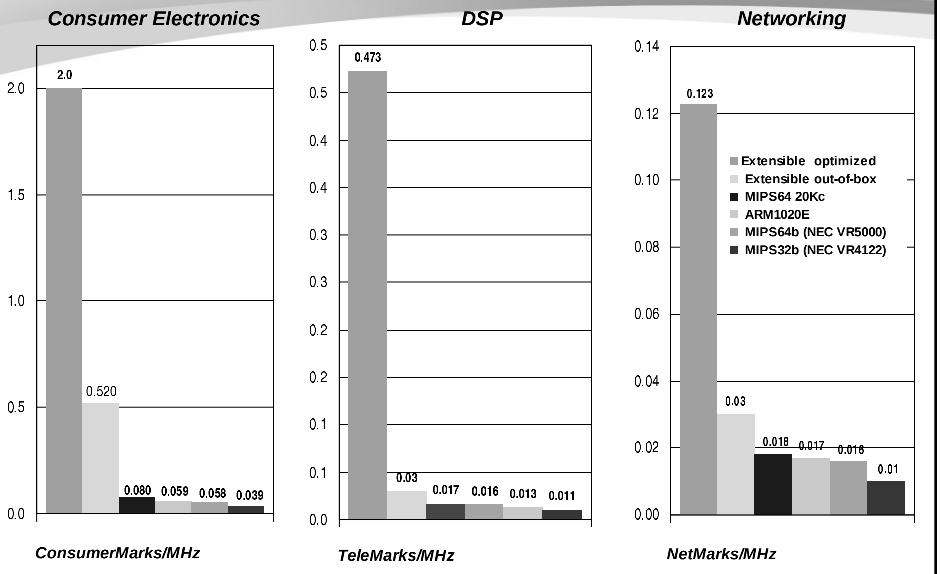
- Often resulted from research projects
  - E.g. nML – TU Berlin
  - Lisa – RWTH Aachen
- Research often used to generate software tools only
  - Eg Chess/Checkers – ISS and compilers– IMEC
  - Lisa – ISS
- Sometimes commercialised
  - Target compilers
  - Axys – now ARM; Lisatek – now CoWare
- Sometimes entirely commercial
  - E.g. Tensilica Instruction Extension Language (TIE)
    - Used for most of HW and SW toolchain but not for all parts of processor
    - Rest configured using GUI configurator

## Example of configuration/extension space for ASIP

***Configuration***

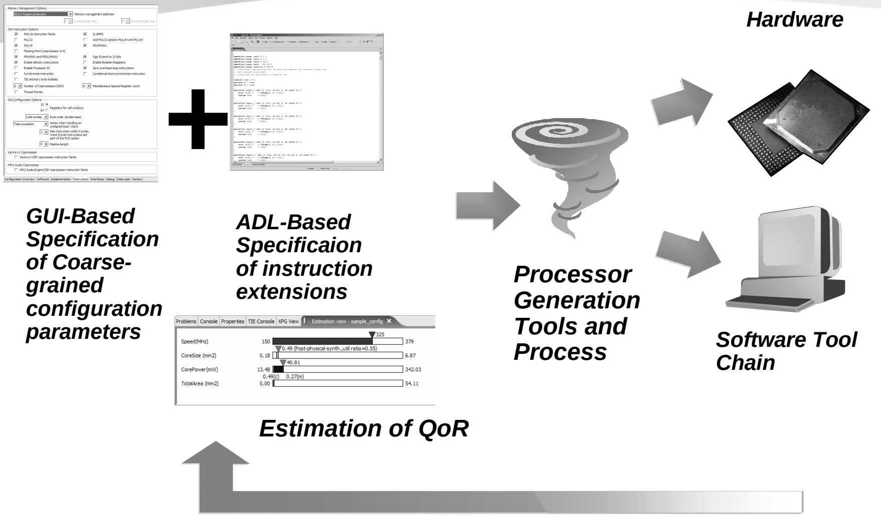

Instruction Set Extension
Instruction Fetch / Decode
Designer-defined FLIX parallel execution pipelines - "N" wide
Base ISA Execution Pipeline
Processor Controls
Trace/TJAG/OCD
Interrupts, Breakpoints, Timers
User Defined Execution Units, Register Files and Interfaces
User Defined Execution Units, Register Files and Interfaces
Register File
Base ALU
Optional Execution Units
User Defined Execution Unit
Vectra LX DSP Engine
Data Load/Store Unit
Load/Store Unit #2
Local Instruction Memories
External Bus Interface
Processor Interface (PIF) to System Bus
Local Data Memories
Xtensa Local Memory Interface
User Defined Queues / Ports up to 1M Pins
■ Base ISA Feature
■ Configurable Functions
■ Optional Function
■ Optional & Configurable
□ Designer Defined Features (TIE)

# Benefits of Configurability



Source: EEMBC

# An example ASIP generation flow



GUI-Based Specification of Coarse-grained configuration parameters

ADL-Based Specificaion of instruction extensions

Processor Generation Tools and Process

Software Tool Chain

Estimation of QoR

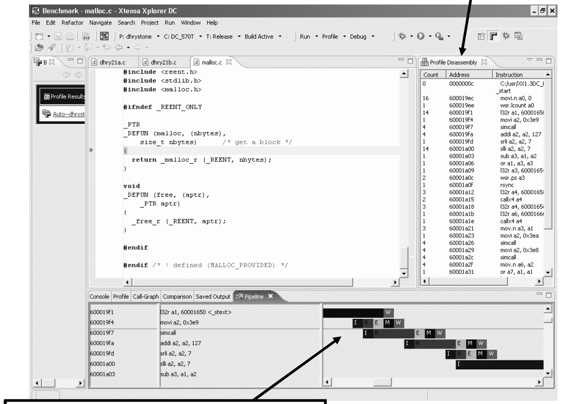
Manual iteration

# Determining the platform specifics for ASIPs

- Manual Methods
  - Profiling source code execution on target processor
  - Finding time consuming routines
  - Finding time consuming loop nests
  - Devising new instructions
  - Optimising memory accesses
  - Modifying source code, and
    - ………….. Round the loop again
- Automated Methods
  - Seek to move from source code to ASIP configuration in a single step

# Manual profiling for optimisation

**Profile Disassembly:**
**View disassembled instructions with cycle counts**



Benchmark - malloc.c - Xtensa Xplorer DC
File Edit Refactor Navigate Search Project Run Window Help
P: dhrystone C: DC_570T T: Release Build Active Run Profile Debug
B
Profile Results
Auto--dhryst
dhry21a.c
dhry21b.c
malloc.c
Profile Disassembly
Console Profile Call-Graph Comparison Saved Output Pipeline

```
#include <reent.h>
#include <stdlib.h>
#include <malloc.h>

#ifndef _REENT_ONLY

_PTR
_DEFUN (malloc, (nbytes),
    size_t nbytes)      /* get a block */
{
  return _malloc_r (_REENT, nbytes);
}

void
_DEFUN (free, (aptr),
    _PTR aptr)
{
  _free_r (_REENT, aptr);
}

#endif

#endif /* ! defined (MALLOC_PROVIDED) */
```

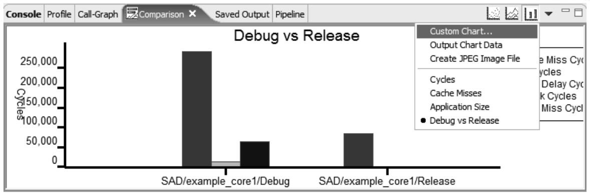
| Count | Address | Instruction |
|---|---|---|
| 0 | 0000000c | C:/usr/XX1.3DC_I<br>_start |
| 16 | 600019ec | movi.n a0, 0 |
| 1 | 600019ee | wsr.lcount a0 |
| 14 | 600019f1 | l32r a1, 6000165 |
| 1 | 600019f4 | movi a2, 0x3e9 |
| 4 | 600019f7 | simcall |
| 4 | 600019fa | addi a2, a2, 127 |
| 1 | 600019fd | srli a2, a2, 7 |
| 14 | 60001a00 | slli a2, a2, 7 |
| 1 | 60001a03 | sub a3, a1, a2 |
| 1 | 60001a06 | or a1, a3, a3 |
| 1 | 60001a09 | l32r a3, 6000165 |
| 2 | 60001a0c | wsr.ps a3 |
| 1 | 60001a0f | rsync |
| 3 | 60001a12 | l32r a4, 6000165 |
| 2 | 60001a15 | callx4 a4 |
| 3 | 60001a18 | l32r a4, 6000165 |
| 1 | 60001a1b | l32r a6, 6000166 |
| 1 | 60001a1e | callx4 a4 |
| 3 | 60001a21 | mov.n a3, a1 |
| 1 | 60001a23 | movi a2, 0x3ea |
| 4 | 60001a26 | simcall |
| 4 | 60001a29 | movi a2, 0x3e8 |
| 4 | 60001a2c | simcall |
| 4 | 60001a2f | mov.n a6, a2 |
| 1 | 60001a31 | or a7, a1, a1 |

```
600019f1   l32r a1, 60001650 <_stext>     W
600019f4   movi a2, 0x3e9                 I R E M W
600019f7   simcall                        I R E M W
600019fa   addi a2, a2, 127               I R E M W
600019fd   srli a2, a2, 7                 I R E M W
60001a00   slli a2, a2, 7                 I
60001a03   sub a3, a1, a2
```

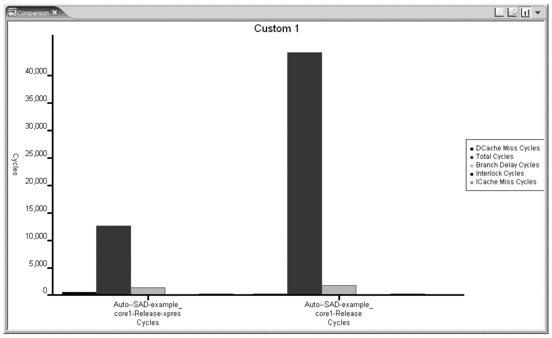
**Pipeline View:**
**See pipeline stalls due to cache misses, branch interlocks, etc**

# Manual profiling for optimisation

***Round the loop comparison of different configurations***

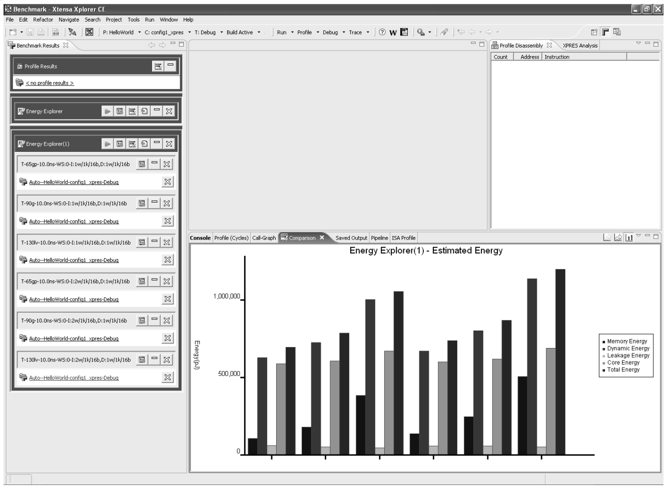


***Exploration of alternative cache configurations***



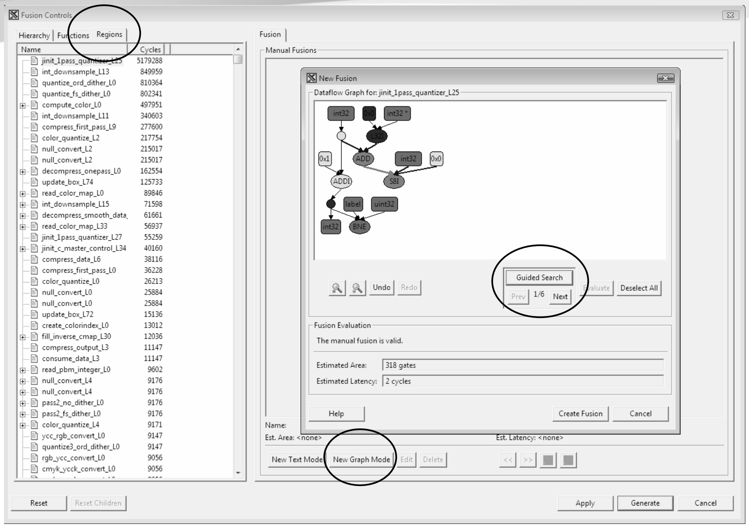
© 2008. Tensilica Inc.

# Energy Consumption Profiling and Optimisation


Benchmark - Xtensa Xplorer CE
File
Edit
Refactor
Navigate
Search
Project
Tools
Run
Window
Help
P: HelloWorld
C: config1_xpres
T: Debug
Build Active
Run
Profile
Debug
Trace
Benchmark Results
Profile Results
< no profile results >
Energy Explorer
Energy Explorer(1)
T-65gp-10.0ns-WS:0-I:1w/1k/16b,D:1w/1k/16b
Auto--HelloWorld-config1_xpres-Debug
T-90g-10.0ns-WS:0-I:1w/1k/16b,D:1w/1k/16b
Auto--HelloWorld-config1_xpres-Debug
T-130lv-10.0ns-WS:0-I:1w/1k/16b,D:1w/1k/16b
Auto--HelloWorld-config1_xpres-Debug
T-65gp-10.0ns-WS:0-I:2w/1k/16b,D:1w/1k/16b
Auto--HelloWorld-config1_xpres-Debug
T-90g-10.0ns-WS:0-I:2w/1k/16b,D:1w/1k/16b
Auto--HelloWorld-config1_xpres-Debug
T-130lv-10.0ns-WS:0-I:2w/1k/16b,D:1w/1k/16b
Auto--HelloWorld-config1_xpres-Debug
Profile Disassembly
XPRES Analysis
Count
Address
Instruction
Console
Profile (Cycles)
Call-Graph
Comparison
Saved Output
Pipeline
ISA Profile
Energy Explorer(1) - Estimated Energy
1,000,000
500,000
0
Energy(pJ)
■ Memory Energy
■ Dynamic Energy
■ Leakage Energy
■ Core Energy
■ Total Energy

© 2008. Tensilica Inc.

## GUI-based assistance for generating fused instructions manually



| Name | Cycles |
|---|---|
| jinit_1pass_quantizer_L25 | 5179288 |
| int_downsample_L13 | 849959 |
| quantize_ord_dither_L0 | 810364 |
| quantize_fs_dither_L0 | 802341 |
| compute_color_L0 | 497951 |
| int_downsample_L11 | 340603 |
| compress_first_pass_L9 | 277600 |
| color_quantize_L2 | 217754 |
| null_convert_L2 | 215017 |
| null_convert_L2 | 215017 |
| decompress_onepass_L0 | 162554 |
| update_box_L74 | 125733 |
| read_color_map_L0 | 89846 |
| int_downsample_L15 | 71598 |
| decompress_smooth_data_ | 61661 |
| read_color_map_L33 | 56937 |
| jinit_1pass_quantizer_L27 | 55259 |
| jinit_c_master_control_L34 | 40160 |
| compress_data_L6 | 38116 |
| compress_first_pass_L0 | 36228 |
| color_quantize_L0 | 26213 |
| null_convert_L0 | 25884 |
| null_convert_L0 | 25884 |
| update_box_L72 | 15136 |
| create_colorindex_L0 | 13012 |
| fill_inverse_cmap_L30 | 12036 |
| compress_output_L3 | 11147 |
| consume_data_L3 | 11147 |
| read_pbm_integer_L0 | 9602 |
| null_convert_L4 | 9176 |
| null_convert_L4 | 9176 |
| pass2_no_dither_L0 | 9176 |
| pass2_fs_dither_L0 | 9176 |
| color_quantize_L4 | 9171 |
| ycc_rgb_convert_L0 | 9147 |
| quantize3_ord_dither_L0 | 9147 |
| rgb_ycc_convert_L0 | 9056 |
| cmyk_ycck_convert_L0 | 9056 |

Dataflow Graph for: jinit_1pass_quantizer_L25

1/6

Fusion Evaluation

The manual fusion is valid.

Estimated Area: 318 gates

Estimated Latency: 2 cycles

Name:

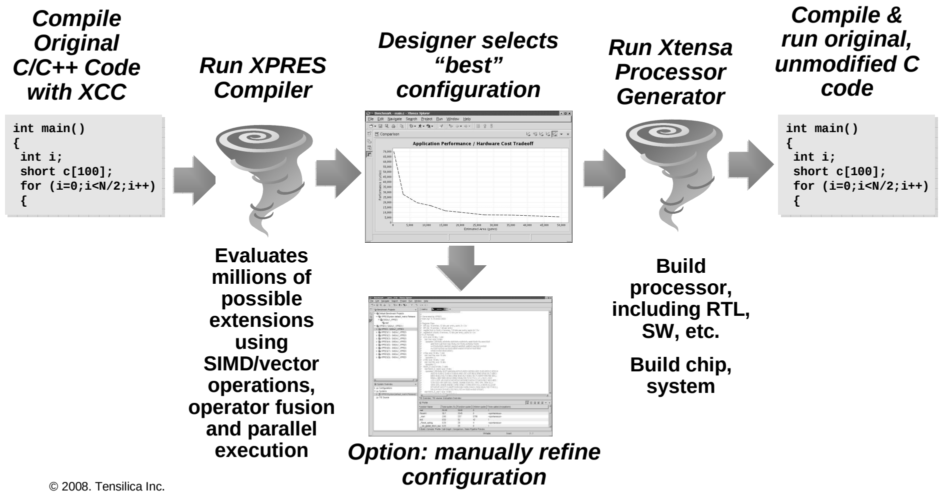
Est. Area: <none>

Est. Latency: <none>

## Automated methods for ASIP specification

- Long a research topic
  - For example, the work of Ienne and Pozzi at EPFL has been reported in many papers over the years
    - Concentrated on instruction fusions
- Also been available for some commercial extensible processors
  - For example, Tensilica’s XPRES tool

# Automated Methods of ASIP generation



*Compile Original C/C++ Code with XCC*

```
int main()
{
 int i;
 short c[100];
 for (i=0;i<N/2;i++)
 {
```

*Run XPRES Compiler*

**Evaluates millions of possible extensions using SIMD/vector operations, operator fusion and parallel execution**

*Designer selects “best” configuration*

*Option: manually refine configuration*

*Run Xtensa Processor Generator*

**Build processor, including RTL, SW, etc.**

**Build chip, system**

*Compile & run original, unmodified C code*

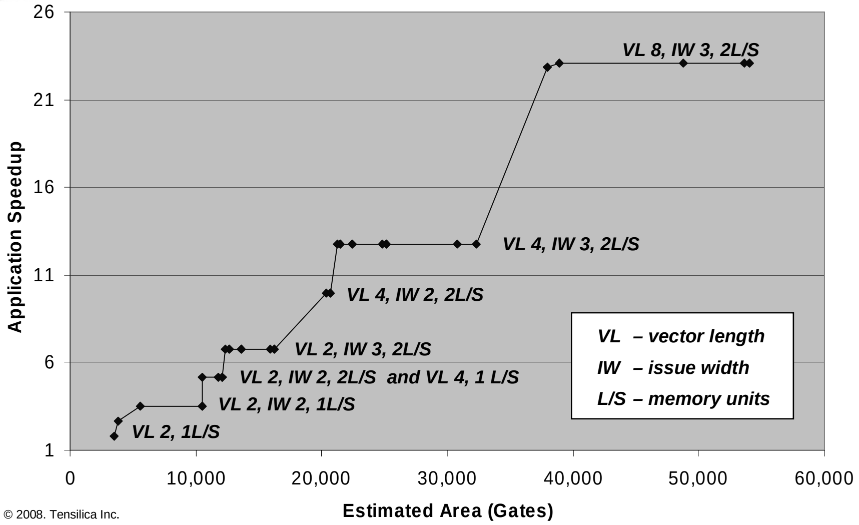
```
int main()
{
 int i;
 short c[100];
 for (i=0;i<N/2;i++)
 {
```

# XPRES Compiler Flow

- Profile / Analyze application
  - Identify performance critical functions / loops
  - Collect operation mix, dependence graphs
  - Provide feedback to user – code transformations to better target meta-ISA (esp. vectorization)
- Generate sets of ISA extensions
  - Each set implements some dimension of meta-ISA
  - Evaluate each set across all functions / loops
  - Performance and cost estimates allow exploration of large design space
- Collect ISA extensions that together provide maximum performance improvement
  - Each collection of ISA extensions forms ASIP
  - Generate family of ASIPs for varying hardware cost budget

# Example: Family of Architectures

`for (i=0; i<n; i++) c[i] = (a[i] * b[i]) >> 4;`



# TIE Techniques Employed by XPRES Compiler: 3 kinds of parallelism

- **Instruction-Level Parallelism: FLIX**
  - Bundling independent operations
- **Data-Level Parallelism: SIMD / Vector**
  - Single instruction, multiple-data operations
- **Pipeline Parallelism: Fusion**
  - Merging of compound, dependent operations
  - E.g. a multiply-add or a multiply feeding a shift

- **All three techniques can be combined**
  - E.g. Parallel FLIX execution of Fused-SIMD operations

# Instruction-Level Parallelism (ILP)

- XPRES supports ILP with multiple-issue
  - 32-bit or 64-bit instructions
  - Each instruction can contain 1-15 slots
  - Each slot can contain arbitrary mix of operations
- Implemented in Xtensa using FLIX
  - Similar to VLIW without code size increase
- Xtensa C/C++ compiler exploits FLIX automatically
  - Software pipelining for loops
  - List scheduler for other code
  - Code scheduling and packing in both compiler and assembler

# XPRES: Instruction Parallelism (exploited using FLIX)

Original C Code

```
  for (i=0; i<n; i++)
      c[i] = (a[i] * b[i]) >> 4
```

64-bit 3-issue FLIX

```
format f  64 { s0, s1, s2 }
slot_opcodes s0 { L32I, S32I }
slot_opcodes s1 { SRAI, MULL }
slot_opcodes s2 { ADDI }
```

64-bit 2-issue FLIX

```
format f  64 { s0, s1 }
slot_opcodes s0 { L32I, S32I }
slot_opcodes s1 { ADDI, SRAI, MULL }
```

- **Performance**
  - **Increased IPC**
- **Hardware cost**
  - **Replicated function units**
  - **Additional register file ports**

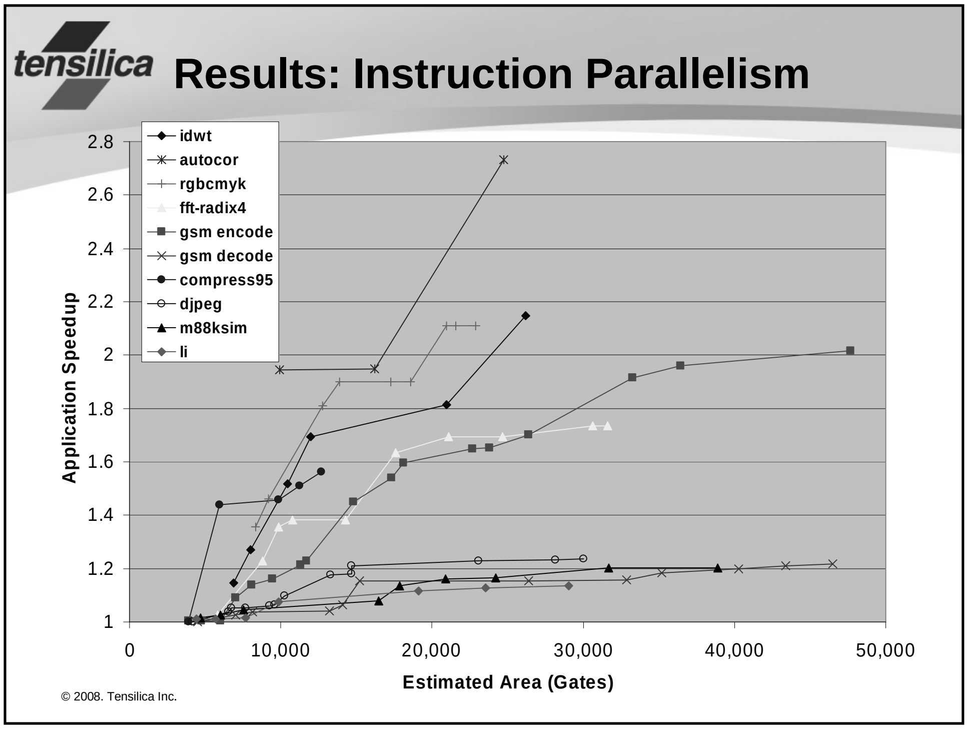
Generated Assembly
(1 iteration in 3 cycles, 2.6x)

```
loop:
  { l8ui a8,a9,0;     mul16u a12,a10,a8;  addi  a9,a9,1 }
  { l8ui a10,a11,0;   nop;                addi  a11,a11,1 }
  { s8i  a13,a14,508; srai   a13,a12,4;   addi  a14,a14,1 }
```

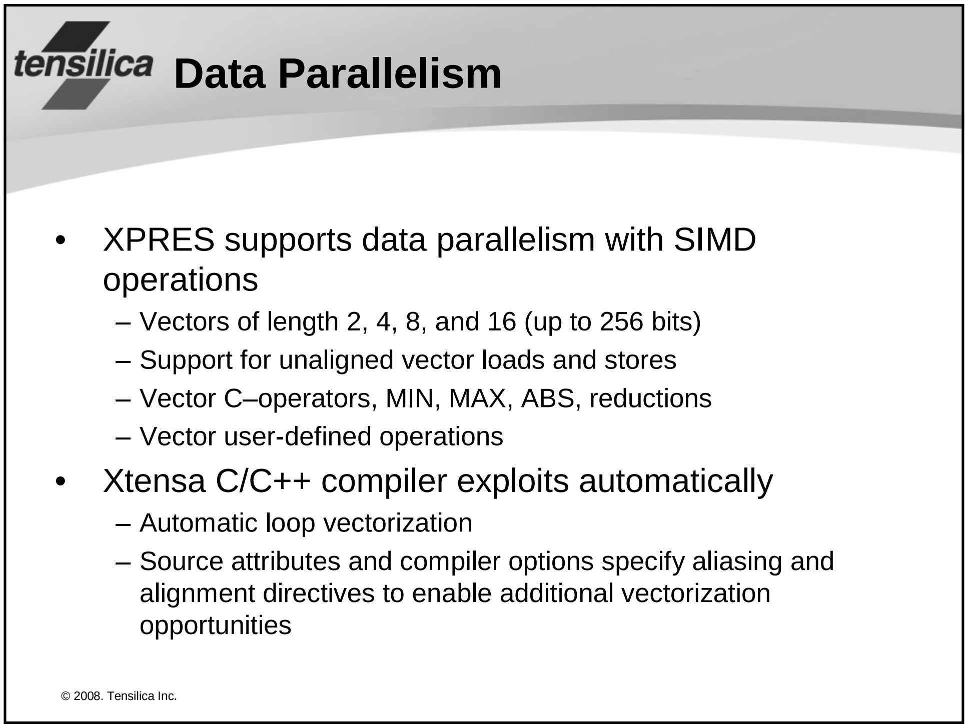
(2 iterations in 8 cycles, 2x)

```
loop:
  { l8ui   a4,a11,12;  addi    a13,a13,4 }
  { l8ui   a2,a9,12;   mul16u  a3,a3,a2 }
  { l8ui   a14,a9,4;   srai    a12,a14,4}

   …
```

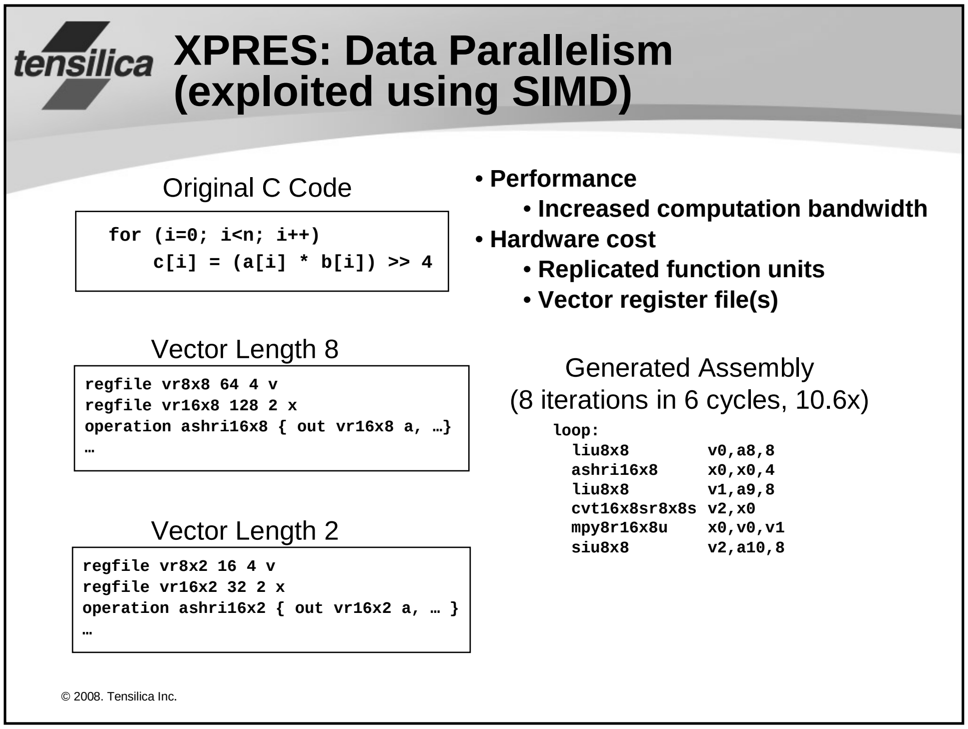


## Results: Instruction Parallelism



## Data Parallelism

- XPRES supports data parallelism with SIMD operations
  - Vectors of length 2, 4, 8, and 16 (up to 256 bits)
  - Support for unaligned vector loads and stores
  - Vector C–operators, MIN, MAX, ABS, reductions
  - Vector user-defined operations
- Xtensa C/C++ compiler exploits automatically
  - Automatic loop vectorization
  - Source attributes and compiler options specify aliasing and alignment directives to enable additional vectorization opportunities



## XPRES: Data Parallelism (exploited using SIMD)

Original C Code

```
  for (i=0; i<n; i++)
      c[i] = (a[i] * b[i]) >> 4
```

Vector Length 8

```
regfile vr8x8 64 4 v
regfile vr16x8 128 2 x
operation ashri16x8 { out vr16x8 a, …}
…
```

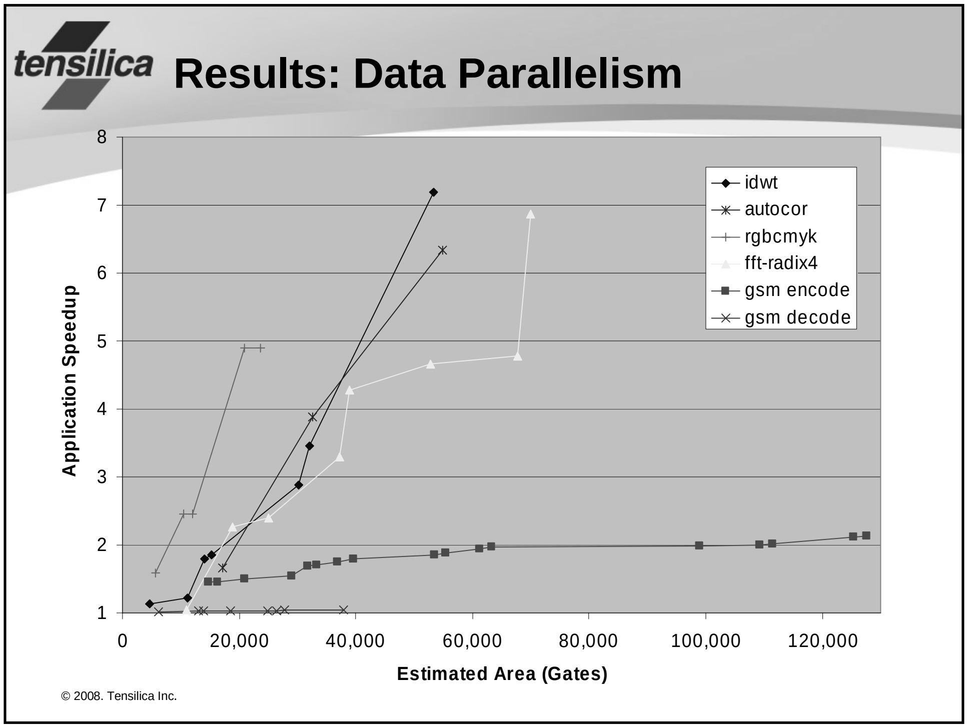
Vector Length 2

```
regfile vr8x2 16 4 v
regfile vr16x2 32 2 x
operation ashri16x2 { out vr16x2 a, … }
…
```

- **Performance**
  - **Increased computation bandwidth**
- **Hardware cost**
  - **Replicated function units**
  - **Vector register file(s)**

Generated Assembly
(8 iterations in 6 cycles, 10.6x)

```
loop:
  liu8x8         v0,a8,8
  ashri16x8      x0,x0,4
  liu8x8         v1,a9,8
  cvt16x8sr8x8s  v2,x0
  mpy8r16x8u     x0,v0,v1
  siu8x8         v2,a10,8
```



## Results: Data Parallelism

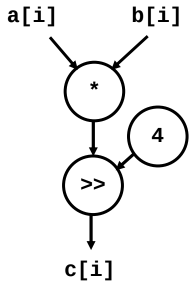
# Pipeline Parallelism

- XPRES supports pipeline parallelism with fused operations
  - Fused operation composed of two or more other operations, plus possibly constant values
  - Latency of fused operation usually less then combined latency of composing operations
  - Limits on input/output operands, number of composing operations, hardware cost, max latency, etc.
  - Graphical support for manual fused operation generation
- Tradeoff performance and generality
  - Performance: large fusions w/ fixed constants
  - Generality: smaller fusions, fewer fixed constants
- Xtensa C/C++ compiler exploits automatically
  - Fused operations automatically replace sequences of composing operations

# XPRES: Pipeline Parallelism (exploited using Fusion)

Original C Code

```
for (i=0; i<n; i++)
    c[i] = (a[i] * b[i]) >> 4
```

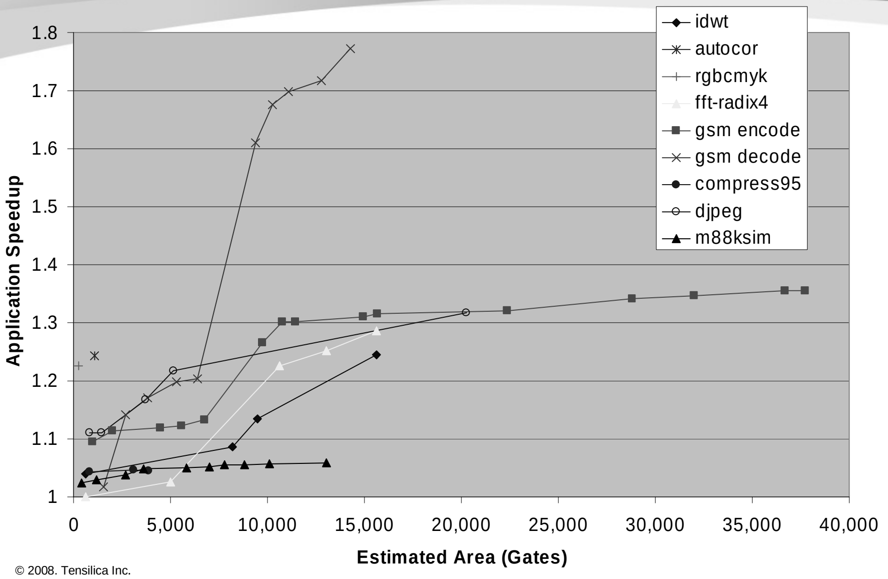


- **Performance**
  - **Decrease instruction count**
  - **Decrease computation latency**
- **Hardware cost**
  - **Logic to share function units**
  - **Register file ports**

Generated Assembly
(1 iteration in 5 cycles, 1.6x)

```
loop:
   l8ui    a12,a11,0
   l8ui    a13,a10,0
   addi.n  a10,a10,1
   addi.n  a11,a11,1
   fusion.mul16u.srai.s8i.addi a9,a12,a13
```

# Results: Pipeline Parallelism



# Exploiting Multiple Types of Parallelism

- XPRES combines FLIX, SIMD, and Fusion

Original C Code

```
for (i=0; i<n; i++)
    c[i] = (a[i] * b[i]) >> 4
```

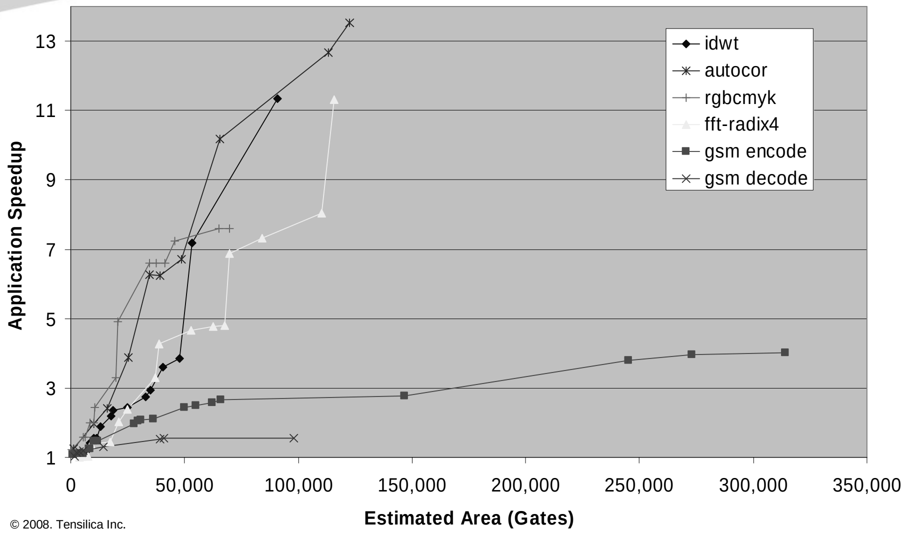
FLIX and Vectorization  (16 iterations in 4 cycles, 32x)

```
{ cvt16x8sr8x8s  v2,x1;       liu8x8  v0,a8,8;     mpy8r16x8u  x2,v1,v2 }
{ si8x8          v2,a10,8;    liu8x8  v1,a9,8;     ashri16x8   v1,v0,4 }
{ siu8x8         v3,a10,16;   liu8x8  v2,a9,8;     ashri16x8   x3,x2,4  }
{ cvt16x8sr8x8s  v3,x3;       liu8x8  v1,a8,8;     mpy8r16x8   x0,v0,v1 }
```

FLIX, Vectorization, and Fusion  (16 iterations in 3 cycles, 42.6x)

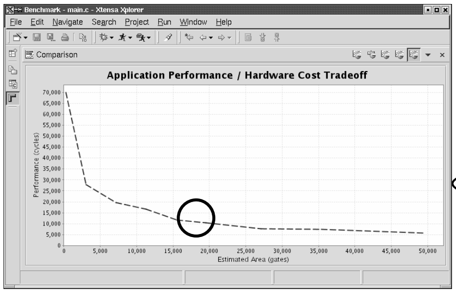
```
{ si8x8   v2,a10,8;           liu8x8  v0,a8,8;     fusion.mpy.ashri.cvtx8  v2,v0,v1 }
{ siu8x8  v5,a10,16;          liu8x8  v1,a9,8;     fusion.mpy.ashri.cvtx8  v5,v3,v4 }
{ liu8x8  v3,a8,8;            liu8x8  v4,a9,8;     nop }
```

# Instruction + Data + Pipeline Parallelism



# *Example:* SAD (Sum of Absolute Differences) Automatic Solution Search

***Visually Presented***

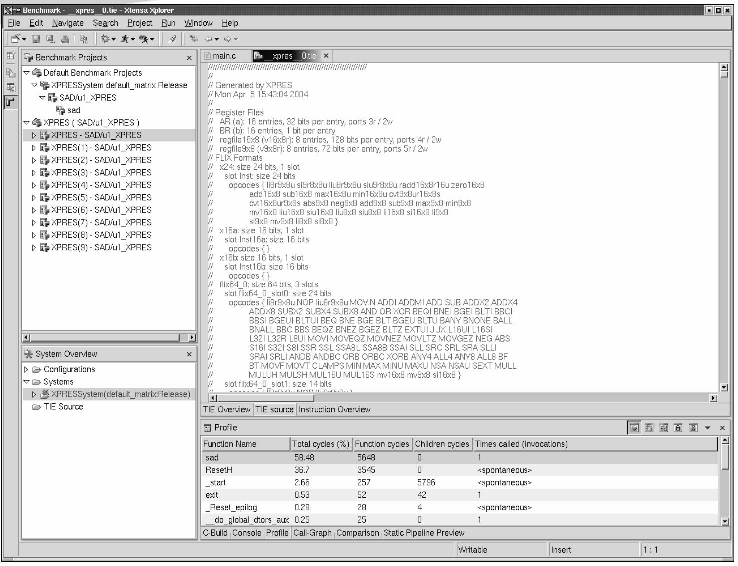


***Wide Range of Choices of Performance Increase -v- Hardware Cost***

***Generated Xtensa LX Configuration Parameters***

| *i* | Speedup -v- Base processor | Gates Added | SIMD Factor | FLIX Width (Slots) | Load / Store Units | Fusion |
|---|---|---|---|---|---|---|
| 1 | 18.6x | 49,681 | 8 | 3 | 2 | Yes |
| 2 | 14.1x | 35,439 | 8 | 2 | 2 | Yes |
| 3 | 13.9x | 27,082 | 4 | 3 | 2 | Yes |
| 4 | 10.1x | 19,547 | 4 | 2 | 2 | Yes |
| 5 | 9.0x | 15,527 | 2 | 3 | 2 | Yes |
| 6 | 6.8x | 8994 | 4 | 1 | 1 | Yes |
| 7 | 5.3x | 7158 | 2 | 2 | 1 | Yes |
| 8 | 1.5x | 301 | 1 | 1 | 1 | Yes |

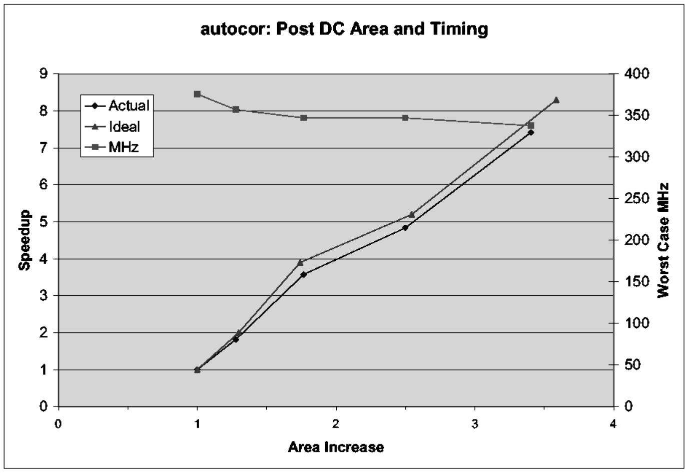
# *Example:* SAD Configuration #4 – Generated TIE

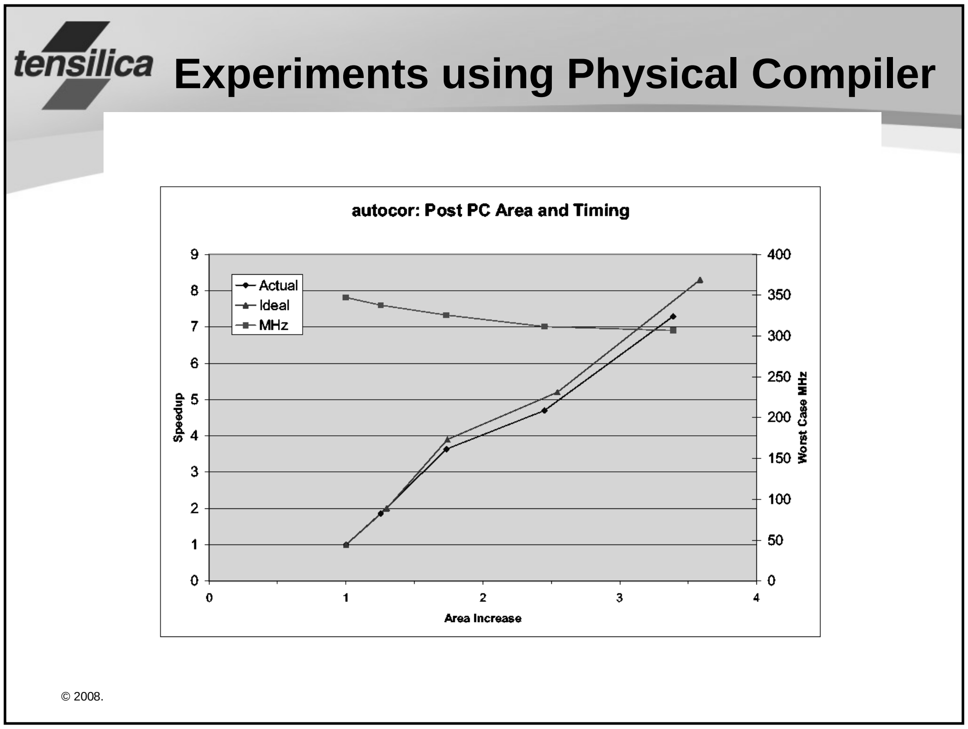


```
////////////////////////////////////////////////////////////////////
//
// Generated by XPRES
// Mon Apr  5 15:43:04 2004
//
// Register Files
//   AR (a): 16 entries, 32 bits per entry, ports 3r / 2w
//   BR (b): 16 entries, 1 bit per entry
//   regfile16x8 (v16x8r): 8 entries, 128 bits per entry, ports 4r / 2w
//   regfile9x8 (v9x8r): 8 entries, 72 bits per entry, ports 5r / 2w
// FLIX Formats
//   x24: size 24 bits, 1 slot
//     slot Inst: size 24 bits
//       opcodes { li8r9x8u si9r8x8u liu8r9x8u siu9r8x8u radd16x8r16u zero16x8
//                 add16x8 sub16x8 max16x8u min16x8u cvt9x8ur16x8s
//                 cvt16x8ur9x8s abs9x8 neg9x8 add9x8 sub9x8 max9x8 min9x8
//                 mv16x8 liu16x8 siu16x8 liu8x8 siu8x8 li16x8 si16x8 li9x8
//                 si9x8 mv9x8 li8x8 si8x8 }
//   x16a: size 16 bits, 1 slot
//     slot Inst16a: size 16 bits
//       opcodes { }
//   x16b: size 16 bits, 1 slot
//     slot Inst16b: size 16 bits
//       opcodes { }
//   flix64_0: size 64 bits, 3 slots
//     slot flix64_0_slot0: size 24 bits
//       opcodes { li8r9x8u NOP liu8r9x8u MOV.N ADDI ADDMI ADD SUB ADDX2 ADDX4
//                 ADDX8 SUBX2 SUBX4 SUBX8 AND OR XOR BEQI BNEI BGEI BLTI BBCI
//                 BBSI BGEUI BLTUI BEQ BNE BGE BLT BGEU BLTU BANY BNONE BALL
//                 BNALL BBC BBS BEQZ BNEZ BGEZ BLTZ EXTUI J JX L16UI L16SI
//                 L32I L32R L8UI MOVI MOVEQZ MOVNEZ MOVLTZ MOVGEZ NEG ABS
//                 S16I S32I S8I SSR SSL SSA8L SSA8B SSAI SLL SRC SRL SRA SLLI
//                 SRAI SRLI ANDB ANDBC ORB ORBC XORB ANY4 ALL4 ANY8 ALL8 BF
//                 BT MOVF MOVT CLAMPS MIN MAX MINU MAXU NSA NSAU SEXT MULL
//                 MULUH MULSH MUL16U MUL16S mv16x8 mv9x8 si16x8 }
//     slot flix64_0_slot1: size 14 bits
```

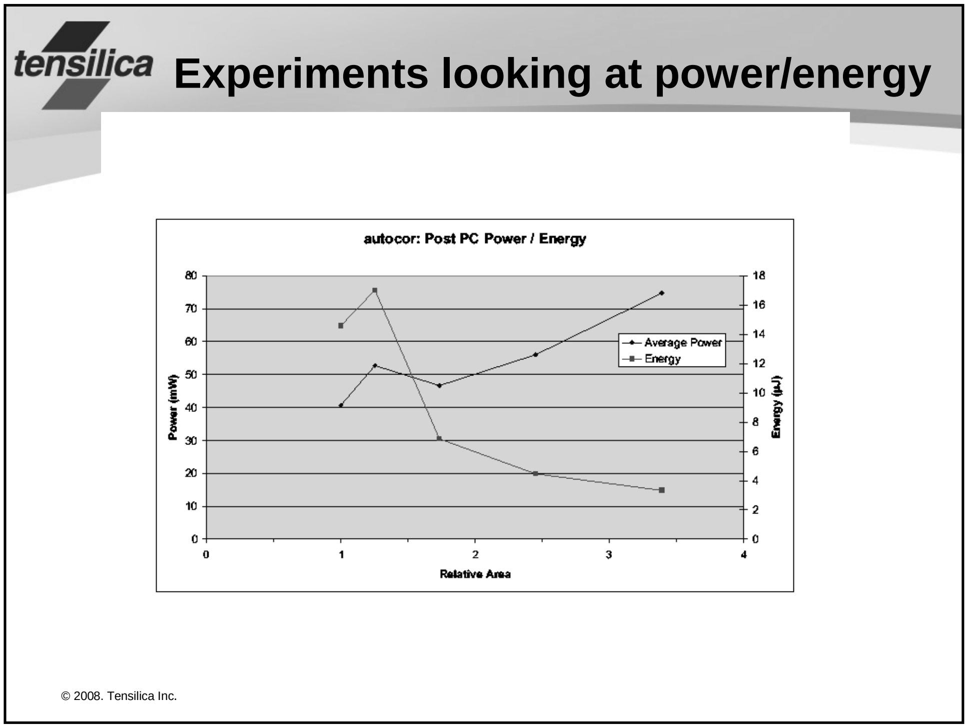
| Function Name | Total cycles (%) | Function cycles | Children cycles | Times called (invocations) |
|---|---|---|---|---|
| sad | 58.48 | 5648 | 0 | 1 |
| ResetH | 36.7 | 3545 | 0 | <spontaneous> |
| _start | 2.66 | 257 | 5796 | <spontaneous> |
| exit | 0.53 | 52 | 42 | 1 |
| _Reset_epilog | 0.28 | 28 | 4 | <spontaneous> |
| __do_global_dtors_aux | 0.25 | 25 | 0 | 1 |

## Exploration of Estimation vs. Actuals – Accuracy of design tradeoffs – monotonicity and correlation

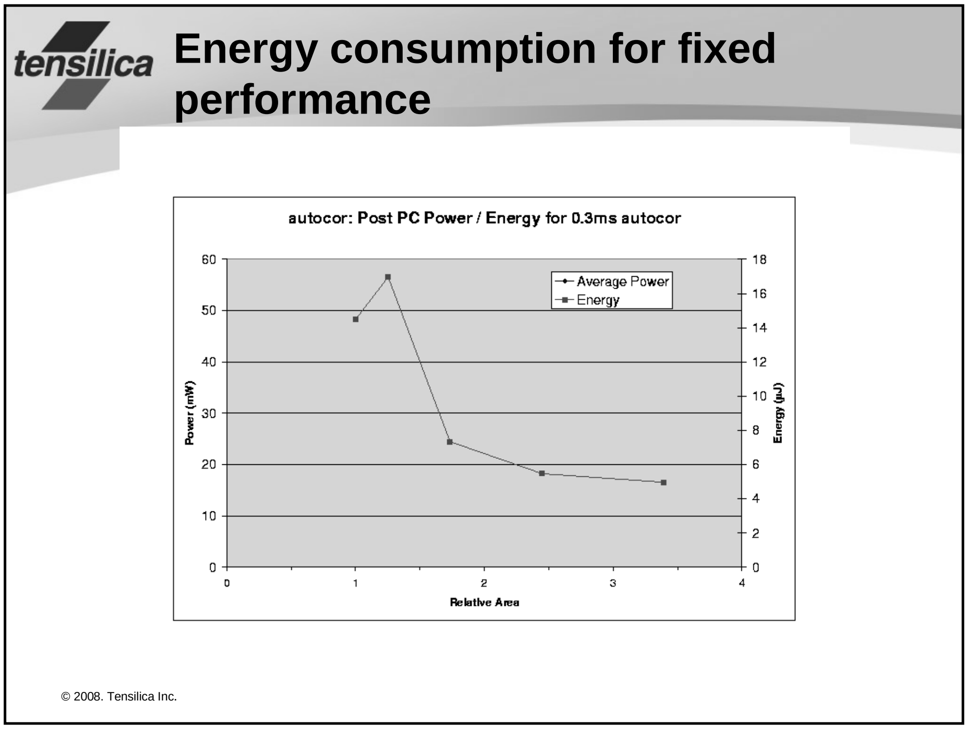




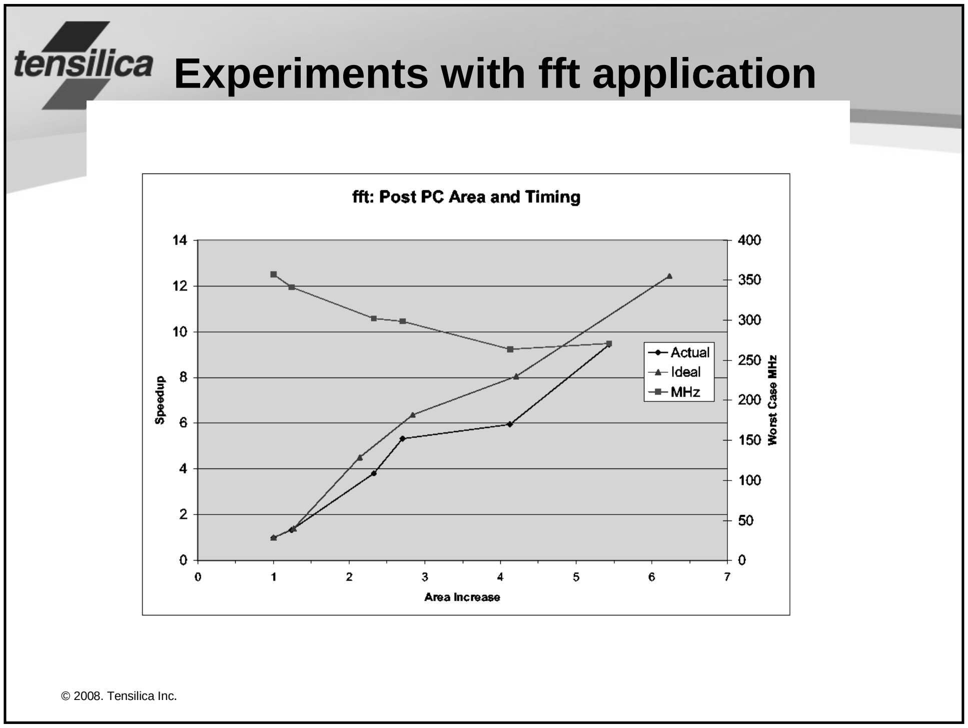
## Experiments using Physical Compiler



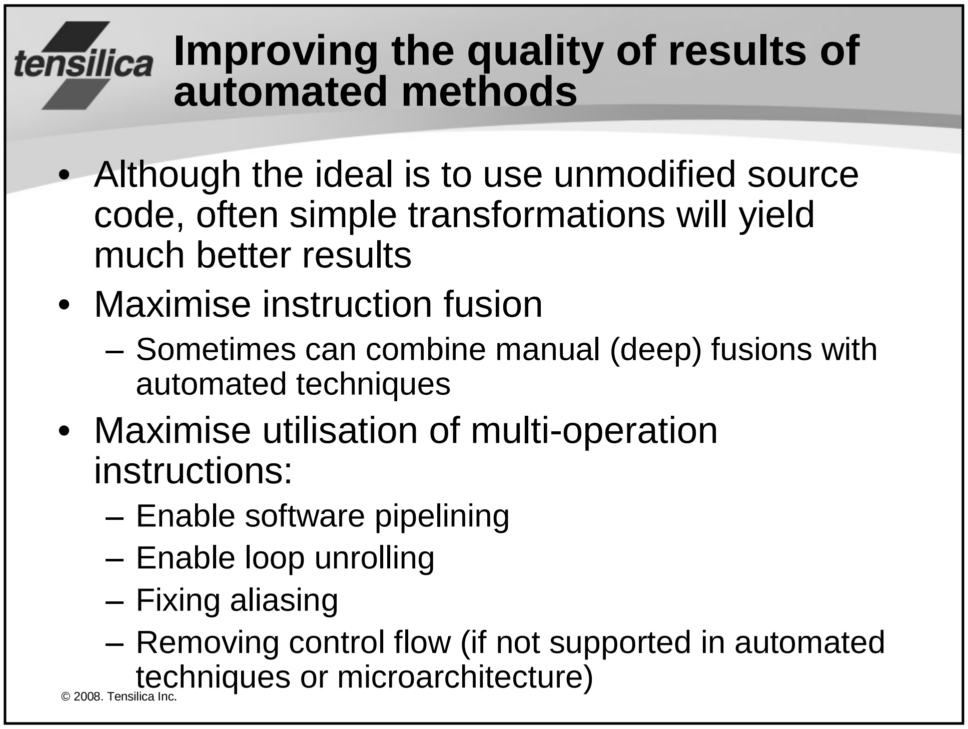
## Experiments looking at power/energy



# Energy consumption for fixed performance

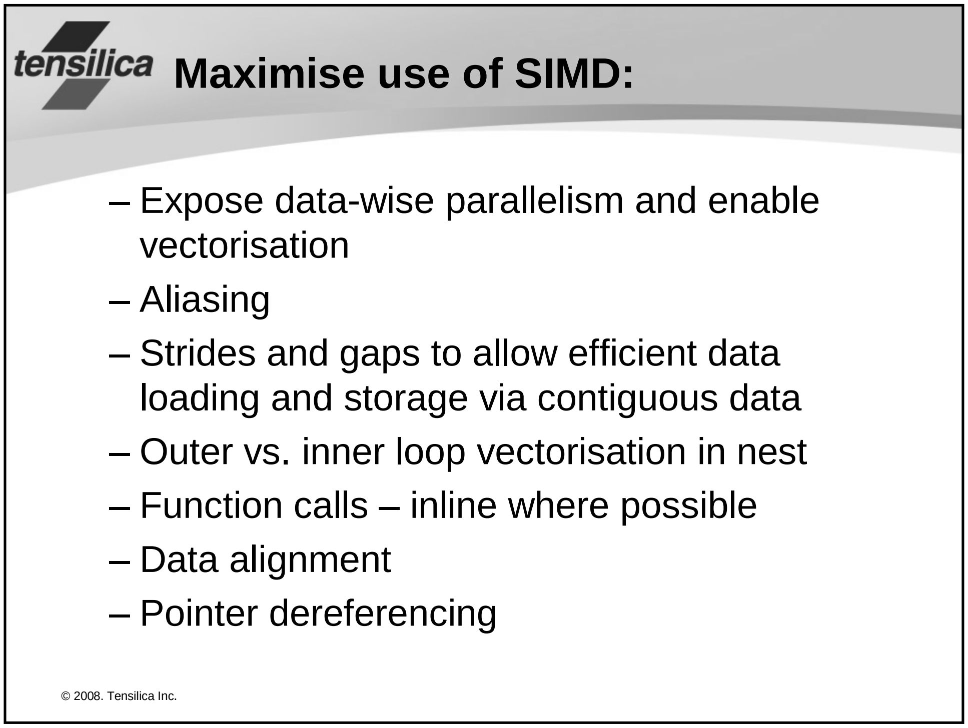


# Experiments with fft application



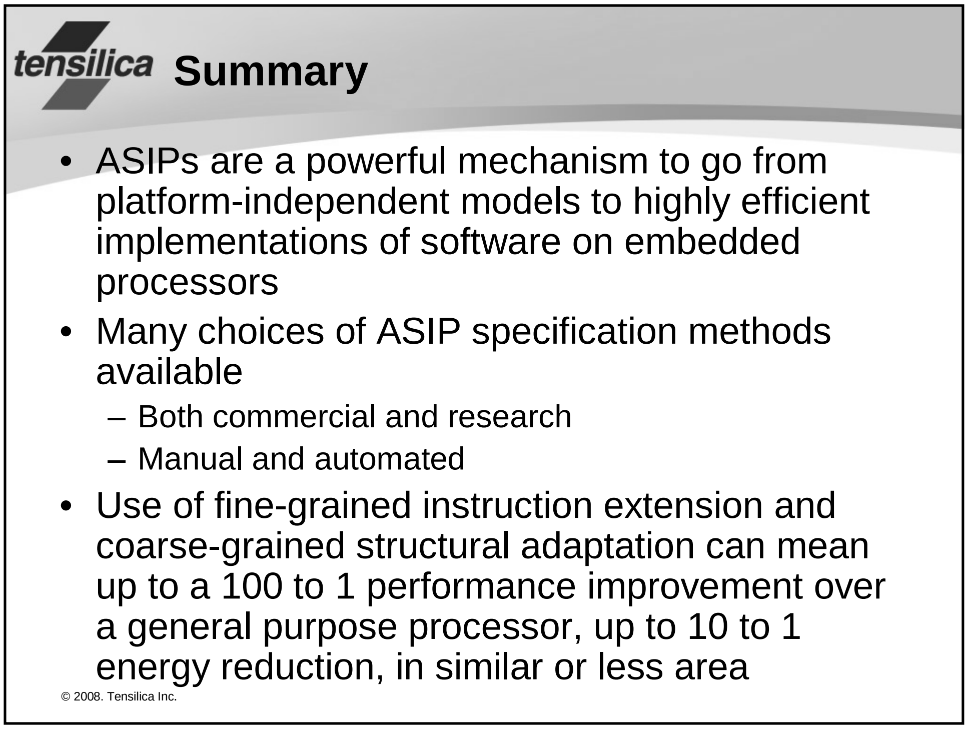
## Improving the quality of results of automated methods

- Although the ideal is to use unmodified source code, often simple transformations will yield much better results
- Maximise instruction fusion
  - Sometimes can combine manual (deep) fusions with automated techniques
- Maximise utilisation of multi-operation instructions:
  - Enable software pipelining
  - Enable loop unrolling
  - Fixing aliasing
  - Removing control flow (if not supported in automated techniques or microarchitecture)



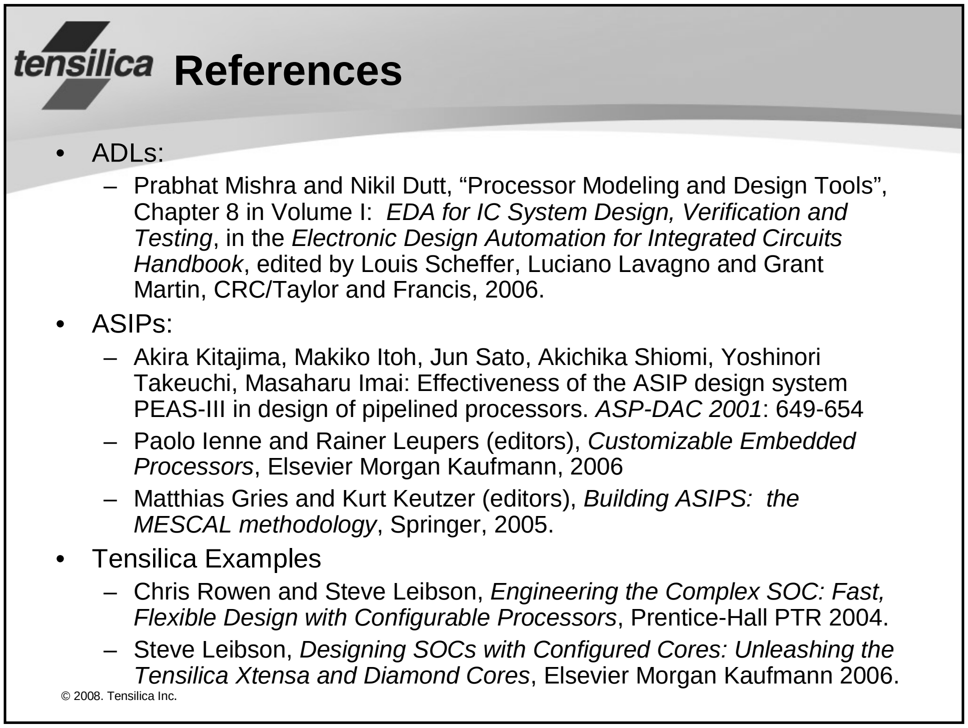
## Maximise use of SIMD:

- Expose data-wise parallelism and enable vectorisation
- Aliasing
- Strides and gaps to allow efficient data loading and storage via contiguous data
- Outer vs. inner loop vectorisation in nest
- Function calls – inline where possible
- Data alignment
- Pointer dereferencing



## Summary

- ASIPs are a powerful mechanism to go from platform-independent models to highly efficient implementations of software on embedded processors
- Many choices of ASIP specification methods available
  - Both commercial and research
  - Manual and automated
- Use of fine-grained instruction extension and coarse-grained structural adaptation can mean up to a 100 to 1 performance improvement over a general purpose processor, up to 10 to 1 energy reduction, in similar or less area



## References

- ADLs:
  - Prabhat Mishra and Nikil Dutt, “Processor Modeling and Design Tools”, Chapter 8 in Volume I: *EDA for IC System Design, Verification and Testing*, in the *Electronic Design Automation for Integrated Circuits Handbook*, edited by Louis Scheffer, Luciano Lavagno and Grant Martin, CRC/Taylor and Francis, 2006.
- ASIPs:
  - Akira Kitajima, Makiko Itoh, Jun Sato, Akichika Shiomi, Yoshinori Takeuchi, Masaharu Imai: Effectiveness of the ASIP design system PEAS-III in design of pipelined processors. *ASP-DAC 2001*: 649-654
  - Paolo Ienne and Rainer Leupers (editors), *Customizable Embedded Processors*, Elsevier Morgan Kaufmann, 2006
  - Matthias Gries and Kurt Keutzer (editors), *Building ASIPS: the MESCAL methodology*, Springer, 2005.
- Tensilica Examples
  - Chris Rowen and Steve Leibson, *Engineering the Complex SOC: Fast, Flexible Design with Configurable Processors*, Prentice-Hall PTR 2004.
  - Steve Leibson, *Designing SOCs with Configured Cores: Unleashing the Tensilica Xtensa and Diamond Cores*, Elsevier Morgan Kaufmann 2006.