

Extracting Parallelism for MPSoC's

David Andrews
Computer Engineering Group
University of Paderborn

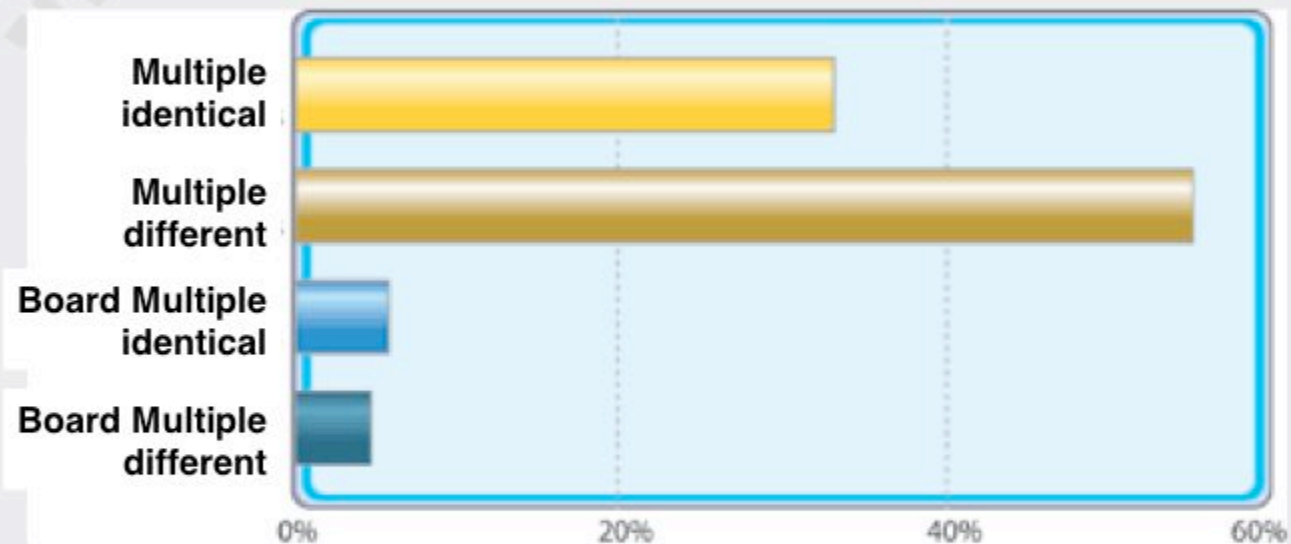
`dandrews@ittc.ku.edu`



UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Heterogeneity

Processor Heterogeneity



□ Nearly 2/3 of SoC's are heterogeneous MP

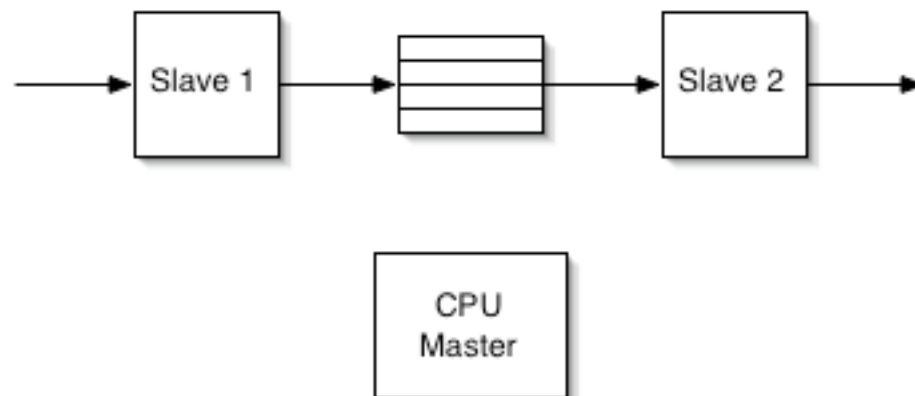
Source: Embedded Systems Programming Magazine, 2005



5

System Architecture

- Most Common Organization of Multiprocessors:
 - ◆ GP CPU: Controller
 - ◆ Special Purpose Processors: Slaves
- Overall Application Partitioning Part of Larger Picture
 - ◆ Assume MPSoC Is One Application Program
 - Application Broken into Threads and Tasks
 - RTOS For Providing Services Inter-Processor
 - ◆ We'll Talk About This Level of Abstraction Later
- Assume A Task Per Processor



Parallelism and Speedups

- Extensible Processors Allow Exploitation of Parallelism
- Where Does Parallelism Come From ?
 - ◆ Remember Amdahls Law

$$\text{Speedup} = \frac{1}{(1 - f) + \frac{Sp}{f}}$$

Parallelism Granularity

Hockney & Jessope

Job Level

Between jobs

Between phases of jobs

Program Level

-Between parts of program

-Within loops

Instruction Level

-Between phases of
instruction execution

-VLIW

Arithmetic/Bit level

-Between elements of a

-Vector operation

-Within ALU circuits

Gottlieb

Procedures and functions

I/O

Overlap Disk, DMA

Loops

Unrolling

Conditional Statements

Both Sides

Basic Blocks

Parallel Blocks

Circuit Levels

Arithmetic/Bit Level

Tensilica Extensible Core Provides:

■ Fusion

- ◆ Identifies Instructions that can be combined

Add R1,R2,R3

Sll R1, R1, ##4

Create: Add_sll R1, R2, R3, #4 /* 1 clock cycle instruction

■ Vector/SIMD

- ◆ Best Bet for Parallelization Using this Method

- Attacks Loops: Unroll and Create New Wider Register File + ALU's of Depth 2, 4, 8

■ VLIW: Called "Flix" (Flexible Length Instruction Xtensions)

- ◆ 32 or 64 bit VLIW Instruction:

- Can be multicycle

Big Win Areas

- Amdahls Law
 - ◆ Look For Where Program Spends Most Time
 - Straight Line Code Not Particularly Ideal
 - Look For Loops
- Classic Compiler Optimizations All Come Into Play
 - ◆ Code Migration
 - ◆ Loop Fusion
 - ◆ Loop Unrolling:
 - Create Parallel Instantitions of the Loop Body
 - Repackage As SIMD/Array Processing Operations

(Imperative) Language Level Representations

Two Approaches to Getting Parallelism Out of a Single Thread

1) Automatic Extraction

- Compiler. Easy for programmer, but doesn't work well

- -parallelism is generally within loops
 - ○ superscalar's do this automatically
 - ○ out of order execution, completion taps instruction level parallelism
 - ○ Studies show approximately 2-3 instructions can be executed in parallel

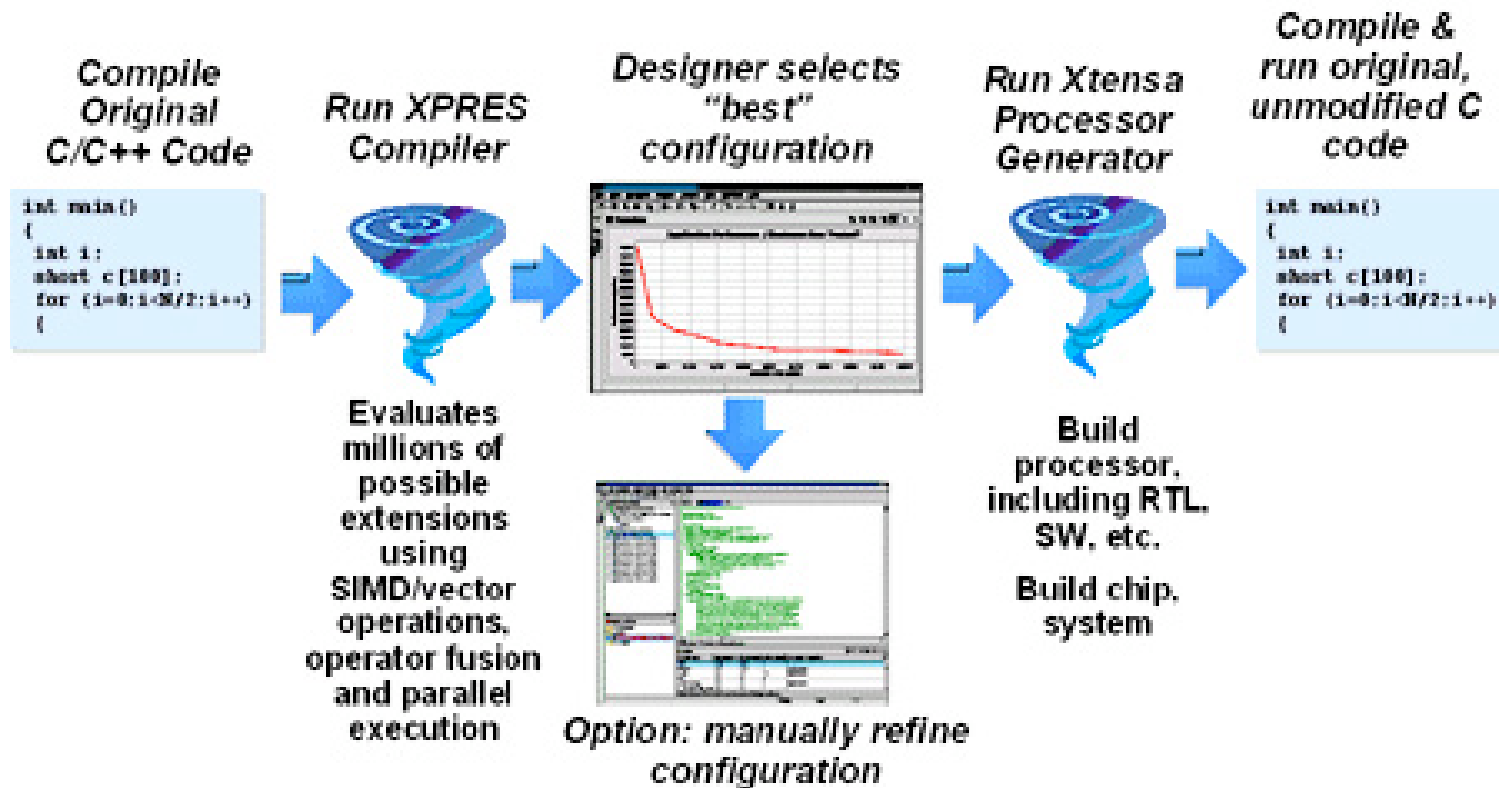
2) User Directed

-Parallel extensions to imperative languages

- - low level (parbegin/parend)
- - SIMD/Systolic approaches

Tensilica Starts With Automatic Extraction, and Allows Users to Craft New Instructions for Extensions

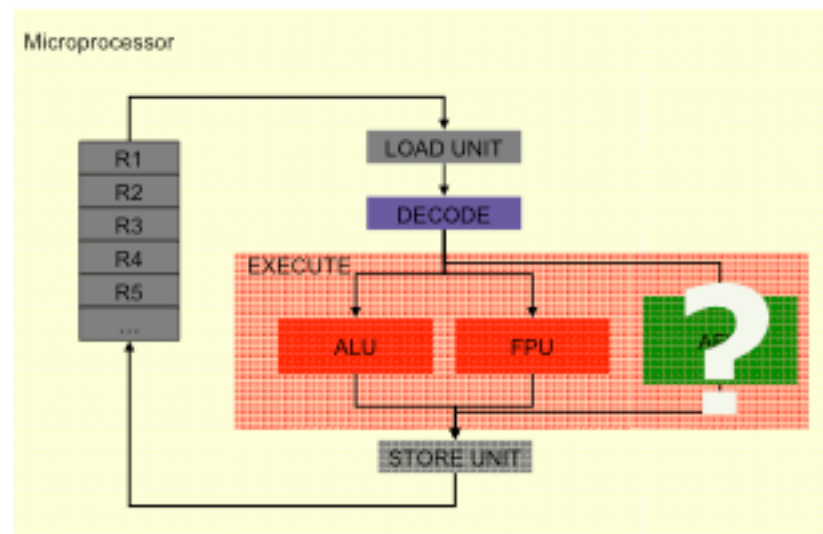
Tensilica Automatic Processor Generation



Automatic Instruction Set Extensions (Next Set from P. lenne's Slides)

Instruction Set Extensions

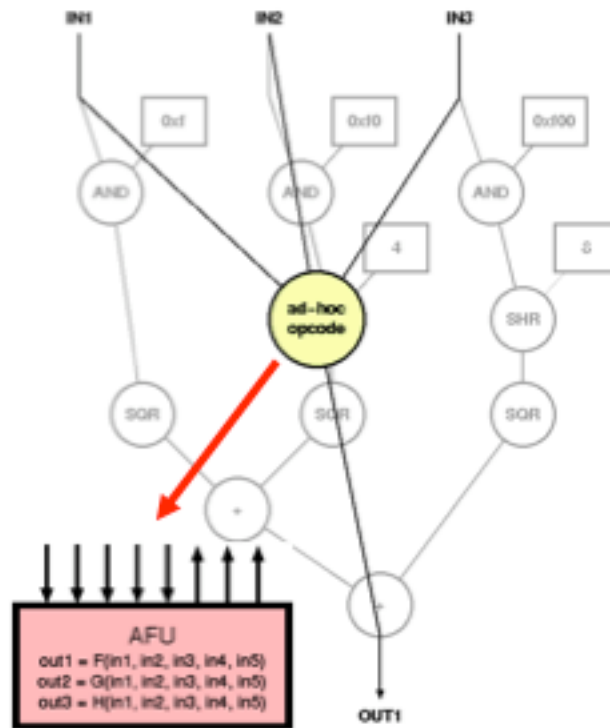
- A "safe" technique for customization



- Available in many commercial processors (from MIPS, STM, IFX, Tensilica, ARC, Xilinx, Altera,...)

Creation of Co-Processor Instruction

Instruction Set Extensions (ISEs)

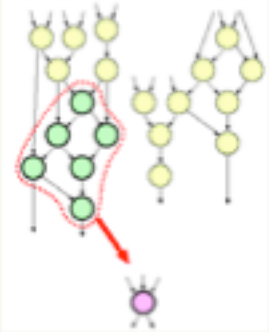


- Collapse a subset of the Direct Acyclic Graph nodes into a single **Application-Specific Functional Unit (AFU)**
 - Exploit cheaply the parallelism within the basic block
 - Simplify operations with constant operands
 - Optimise sequences of instructions (logic, arithmetic, etc.)
 - Exploit limited precision

Tool Focus

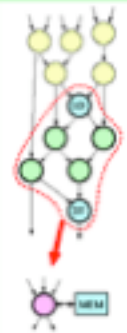
Many Related Problems

Automatic Identification of Instruction-Set Extensions



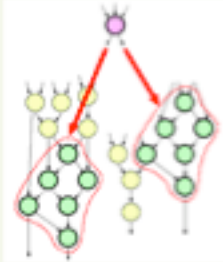
Atasu, Pozzi, Jenne (DAC 2003, BPA, and TCAD 2006)
Biswas, Pozzi, Jenne, Dutt, et al. (DATE 2005 and TVLSI)

Inclusion of Architecturally Visible Registers and Memory



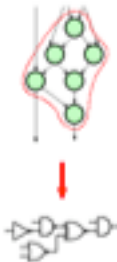
Biswas, Pozzi, Jenne, Dutt, et al. (DAC 2004)
Biswas, Pozzi, Jenne, Dutt, et al. (DATE 2006, BPA Nominee, and TCAD)

Symbolic Algebra for Instruction Selection




Peymandoust, Pozzi, Jenne, and De Micheli (ASAP 2003)

Arithmetic Optimisations



Verma and Jenne (ICCAD 2004)

Pipelining to Relax Port Constraints

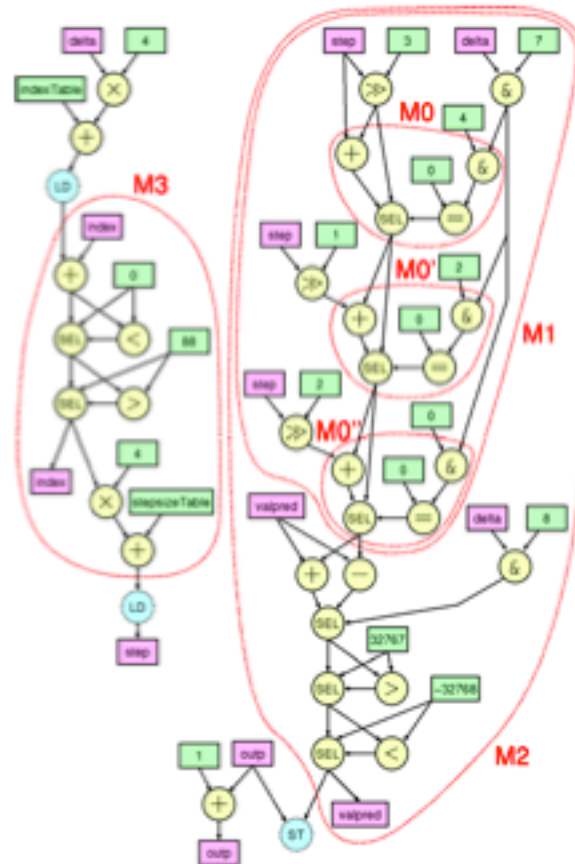


Pozzi and Jenne (CASES 2005)

Data Flow Analysis

- Represent Program As Data Flow Graph

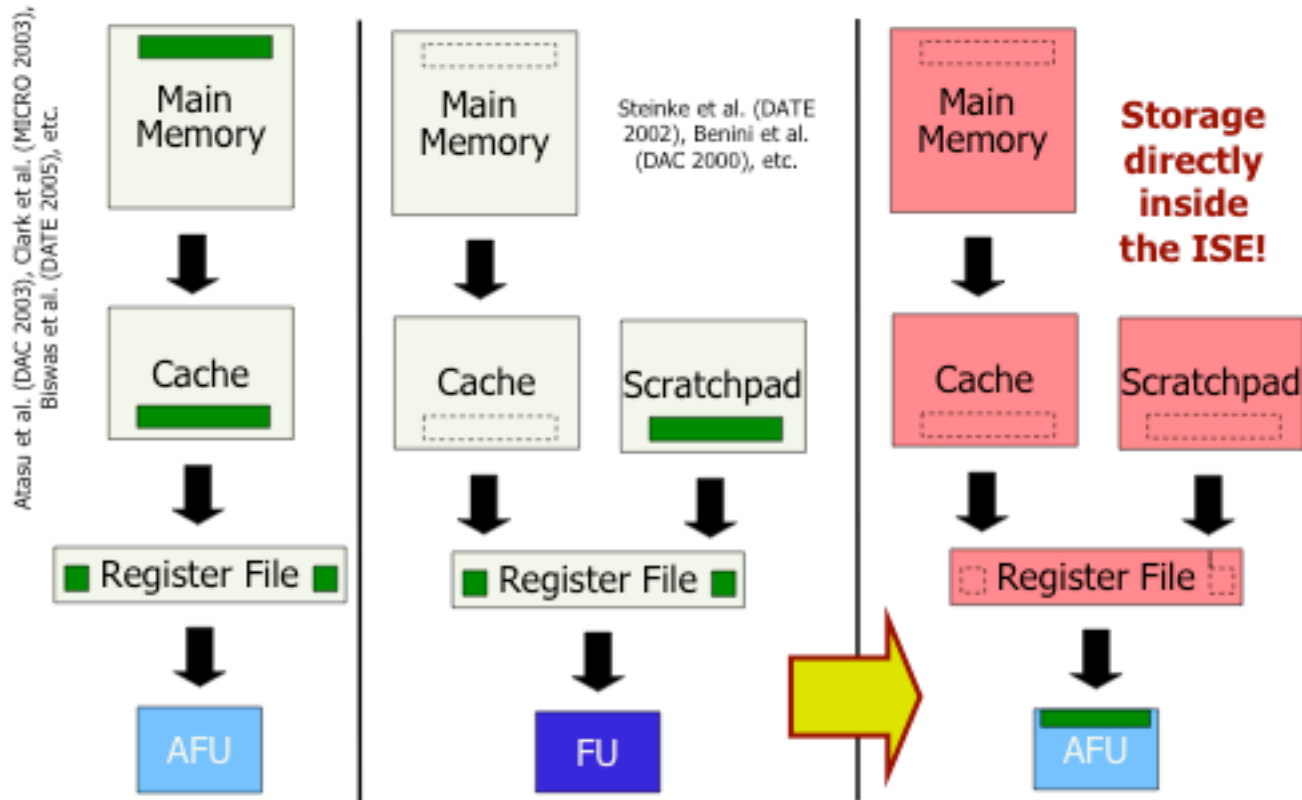
The Basic Problem



- **Goal:** Find subgraphs
 - having a user defined maximum number of inputs and outputs,
 - including disconnected components, and
 - that maximize the overall speedup

Keeping Data Local

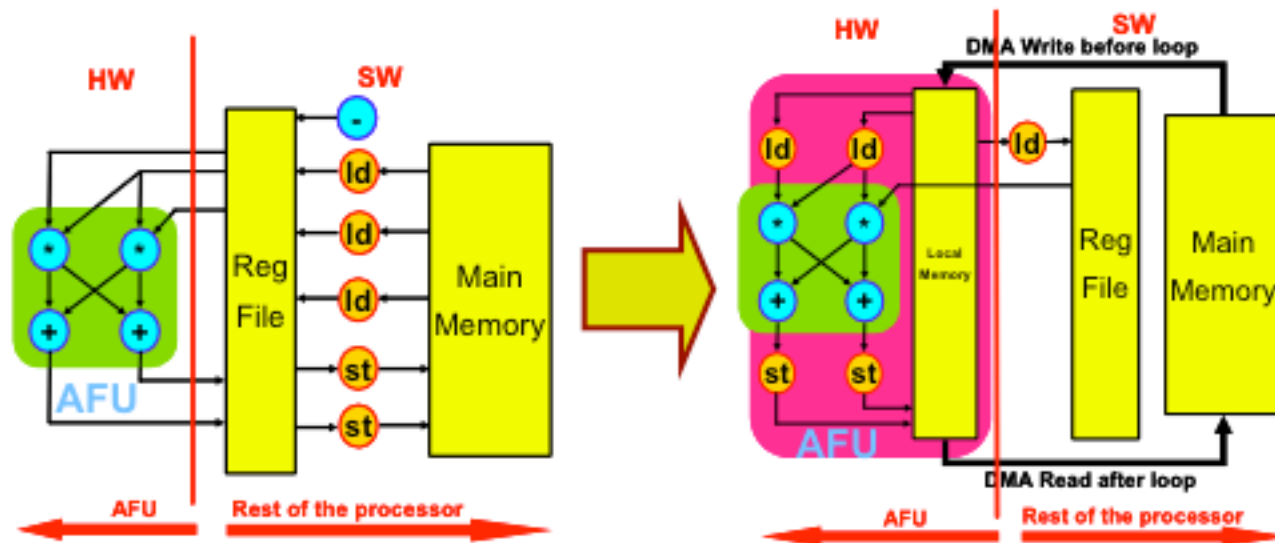
Bringing Data Closer to the Consumer



Fast Transfer Between Memory/AFU

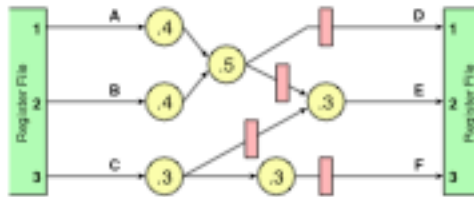
Adding Local Memory to ISEs

- Include selected **LD/ST operations in subgraphs** for instruction set-extensions
- Decide which pieces of data are best **stored locally** in the AFU
- Preload/unload the AFU memories with **DMA transfers** placed in the most economical spots in the Control Flow Graph



Balancing I/O

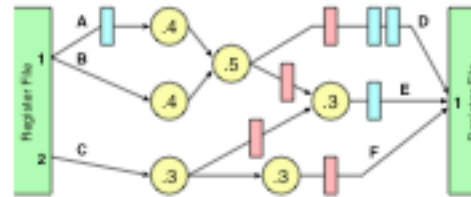
Not Enough RF Ports?



	Rr		Wr
1	A	B	C
2			
3			D
			E
			F

One wants to add **pipeline registers** to the application-specific functional unit...

...and one wants to add **delay registers** to transfer sequentially more values than the register file can in a cycle



	Rr	Wr
1	A	B
2	B	C
		F
		E
		D

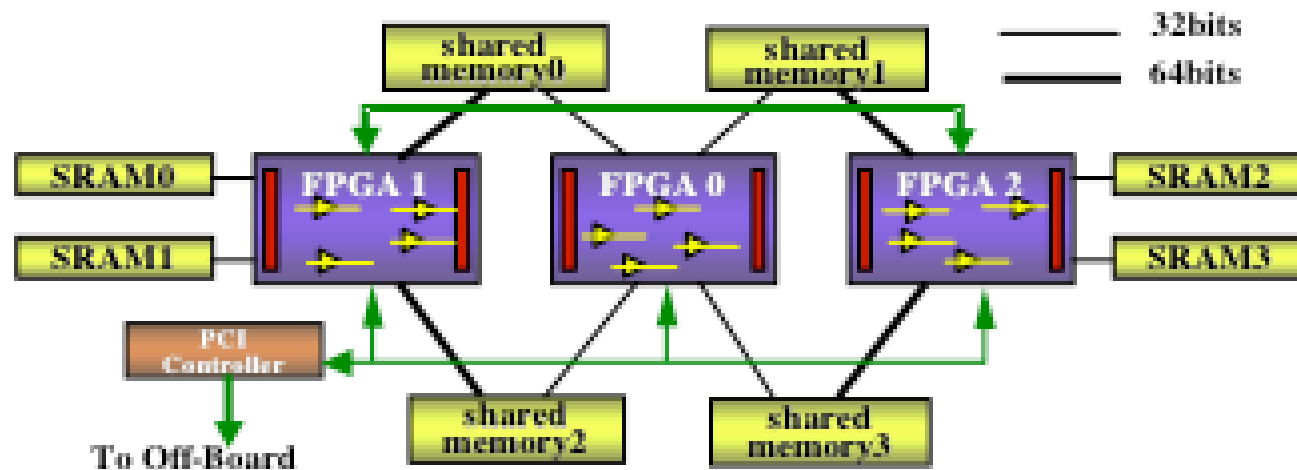


Rd	Wr
1	A
2	B
1	C
	D
	E
	F

Doing both at once minimizes the cost (cycles and registers)

Loop Optimizations

- DEFACTO
 - ◆ Design Environment for Adaptive Computing Technology
- Automated Approach For Co-Processors in FPGA's



Loop Optimizations

- Data Reuse Analysis and Transforms
 - ◆ Reuse Analysis: Tells us How Data Is Reused Between Loop Iterations
 - Input Dependencies: Re-use Data Input from Memory
 - True Dependencies: Re-use a Computed Value
 - Output Dependencies: Update Same Memory Location Several Times
 - ◆ Reuse Transforms
 - Scalar Replacement: Creates On Chip Register for Temp Storage
 - Tapped Delay Lines: Shift Register Structures for Regular Accesses

- Loop Unrolling
 - ◆ Expose Parallelism Within Loop Body

- Tiling
 - ◆ Within Nested Loops, Can Create Spatial “Blocks” That Can Be Unrolled

Create Local Registers

```

int th[60][60];
char mask[4][4];
char image[63][63];
for(m=0; m<60; m++){
  for(n=0; n<60; n++){
    sum = 0;
    for(i=0; i<4; i++){
      for(j=0; j<4; j++){
        if(mask[i][j] != 0)
          sum += image[m+i][n+j];
      }
    }
    th[m][n] = sum;
  }
}

```



(a) Source Code & Reuse Graph

```

for(m = 0; m < 60; m++){
  for(n = 0; n < 60; n++){
    sum = 0;
    for(i = 0; i < 4; i++){
      for(j = 0; j < 4; j++){
        if (m == 0 && n == 0)
          mask_0 = mask[i][j];
        if (mask_0 != 0)
          sum += image[m+i][n+j];
        rotate_register(mask_0,
          mask_1,mask_2,mask_3,
          mask_4,mask_5,mask_6,
          mask_7,mask_8,mask_9,
          mask_10,mask_11,mask_12,
          mask_13,mask_14,mask_15);
      }
    }
    th[m][n] = sum;
  }
}

```

(b) Scalar Replacement

Loop Unrolling

- Inner Loop Body “Expanded”

```
for(m=0; m<60; m++) {
  for(n=0; n<60; n++) {
    sum = 0;
    for(i=0; i<4; i++) {
      for(j=0; j<4; j += 2) {
        if(mask[i][j] != 0) sum += image[m+i][n+j];
        if (mask[i][j+1] != 0) sum += image[m+i][n+j+1];
      }
    }
    th[m][n] = sum;
  }
}
```

(c) Loop Unrolling

Tiling

- Can Be Used to Create Coarse Grained Processing Tiles

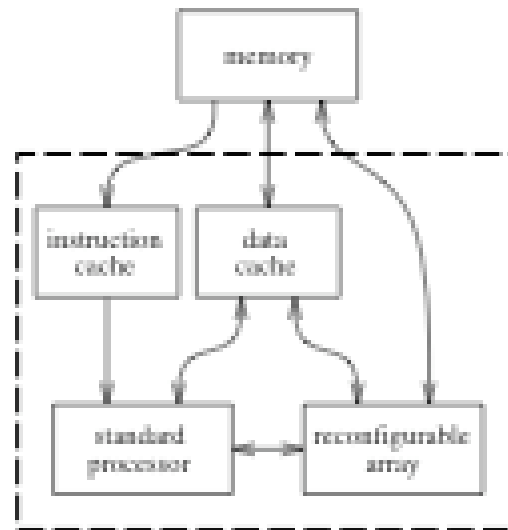
```
for (m = 0; m < 60; m++) {
  for (n = 0; n < 60; n++) {
    sum = 0;
    for (i_tile = 0; i_tile < 2; i_tile++) {
      if (m == 0 && n == 0) {
        mask_0_0 = mask[2*i_tile][0]; mask_1_0 = mask[2*i_tile][1];
        mask_2_0 = mask[2*i_tile][2]; mask_3_0 = mask[2*i_tile][3];
        mask_4_0 = mask[2*i_tile+1][0]; mask_5_0 = mask[2*i_tile+1][1];
        mask_6_0 = mask[2*i_tile+1][2]; mask_7_0 = mask[2*i_tile+1][3];
      }
      if (mask_0_0 != 0) sum += image[m+2*i_tile][n];
      if (mask_1_0 != 0) sum += image[m+2*i_tile][n+1];
      if (mask_2_0 != 0) sum += image[m+2*i_tile][n+2];
      if (mask_3_0 != 0) sum += image[m+2*i_tile][n+3];
      if (mask_4_0 != 0) sum += image[m+2*i_tile+1][n];
      if (mask_5_0 != 0) sum += image[m+2*i_tile+1][n+1];
      if (mask_6_0 != 0) sum += image[m+2*i_tile+1][n+2];
      if (mask_7_0 != 0) sum += image[m+2*i_tile+1][n+3];
      swap_reg(mask_0_0, mask_0_1); swap_reg(mask_1_0, mask_1_1);
      swap_reg(mask_2_0, mask_2_1); swap_reg(mask_3_0, mask_3_1);
      swap_reg(mask_4_0, mask_4_1); swap_reg(mask_5_0, mask_5_1);
      swap_reg(mask_6_0, mask_6_1); swap_reg(mask_7_0, mask_7_1);
    }
    th[m][n] = sum;
  }
}
```

(d) Tiling

Figure 3. ATR Kernel

GARP

- Re-Programmable Application Specific Functional Unit (ASFU)
 - ◆ Allows “different” Custom Instructions
 - ◆ Uses Reconfigurable Array
 - ◆ Exploits Loop Bodies For Highest Return



GARP

■ Based on Single Issue MIPS Core

- ◆ Reconfigurable Array For Exploiting Loop Level Parallelism
 - Few Cycles From Registers to Array
 - Direct Connection To Memory (Most Loops Operate on Memory Structures)
 - Array Rapidly Reconfigurable By Having Multiple Planes
- ◆ Based on Unaltered C Code For Compatibility

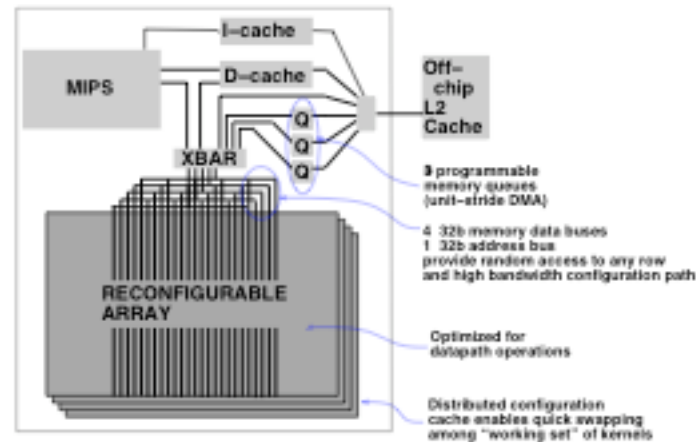


Figure 1.1: The Garp chip.

Compiler Flow

- Identify loops and map into hardware
 - ◆ Accelerate From Custom Loop Bodies
 - ◆ Cannot “unroll” loops due to size

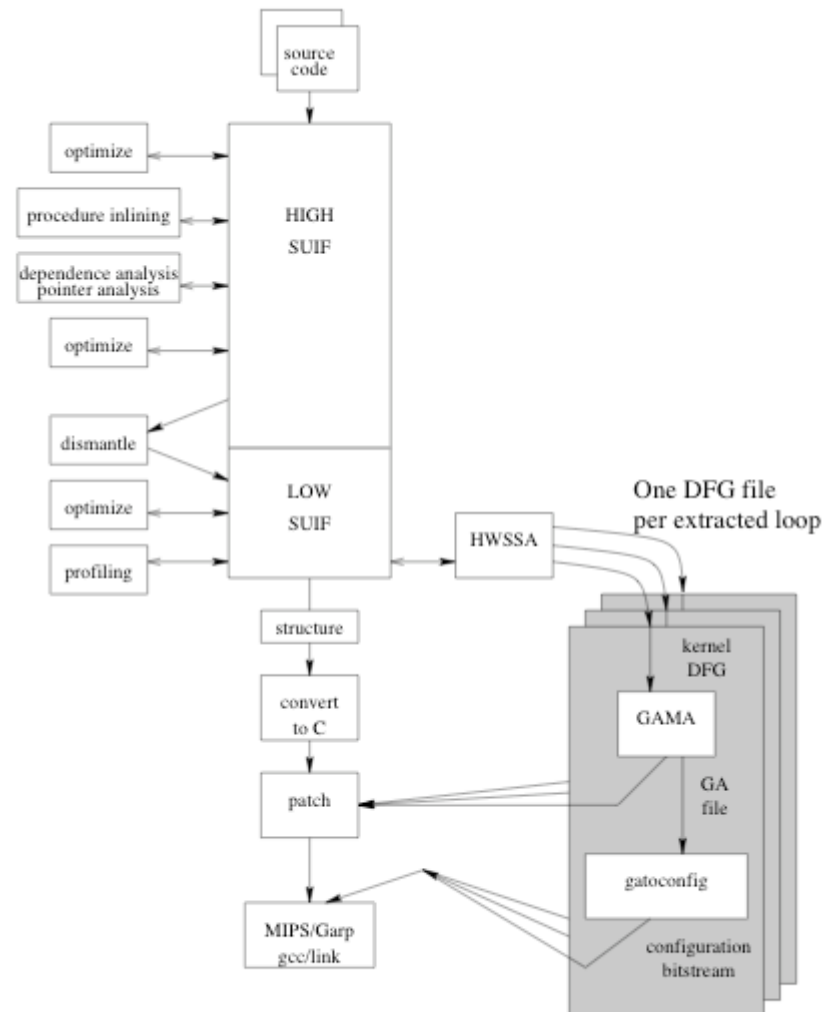


Figure 2.1: Garp Compiler Structure

Identifying Hyperblocks

- Technique from VLIW Architectures for Multiple Paths

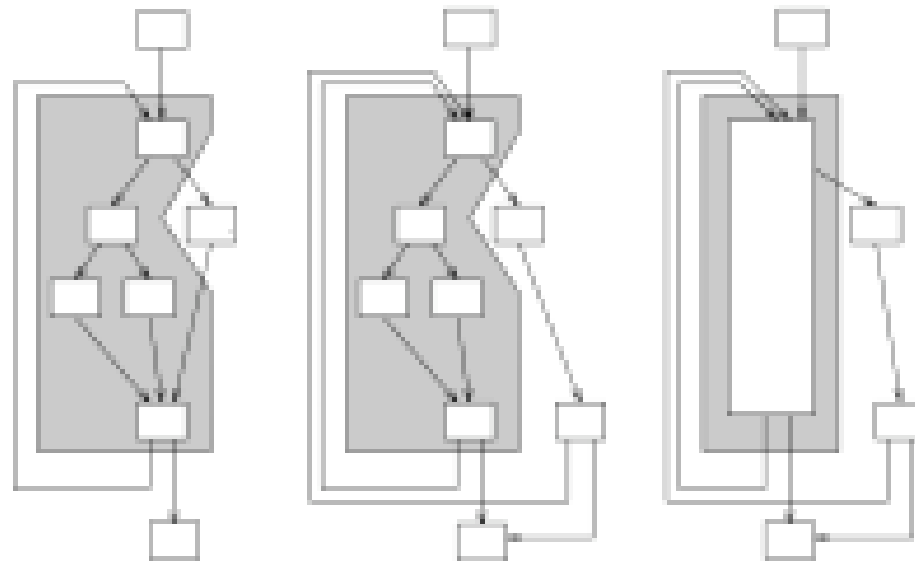
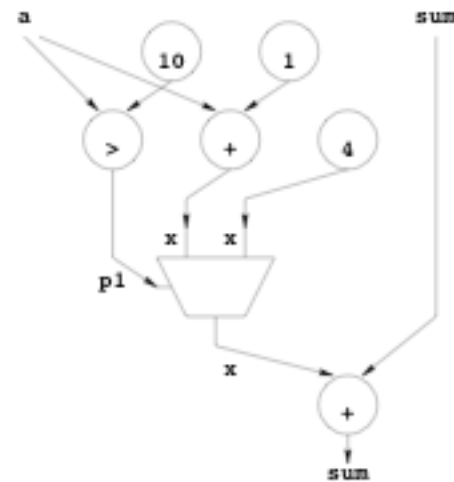
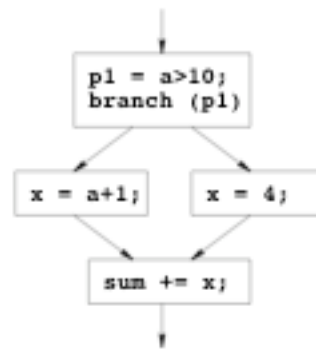


Fig. 1. Hyperblock formation for VLIW compilation.

Predicating Conditionals within Hyberblock

```
if (a>10) {  
  x = a+1;  
} else {  
  x = 4;  
}  
sum += x;
```



(a)

(b)

(c)

Figure 1.2: Predicated, speculative execution.

Loop Duplication for Hardware

- Creating Hardware Copies of a Block + Software Copy

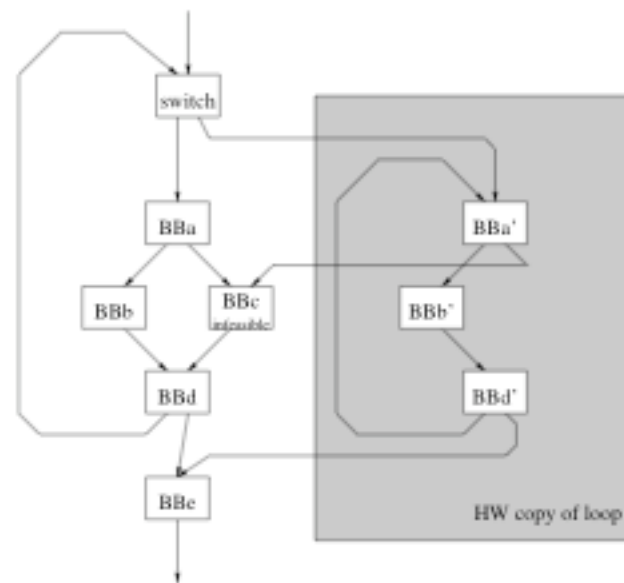


Figure 3.4: Loop Duplication

Reference Counts

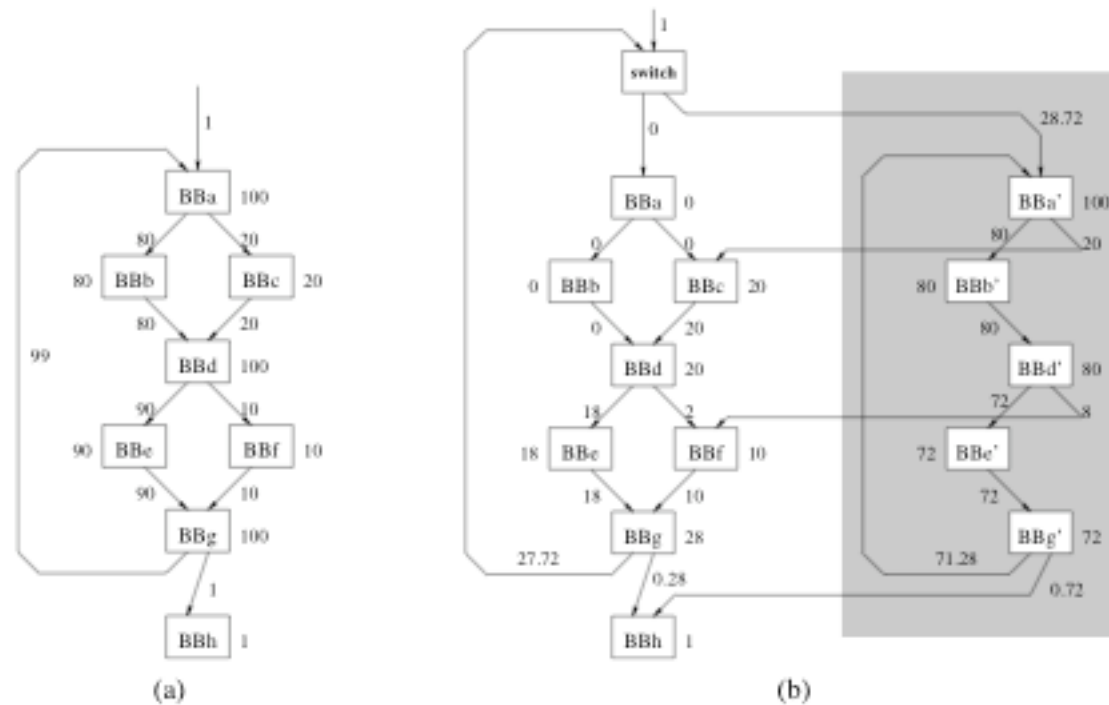


Figure 3.5: Profiling count adjustment. (a) original loop and profiling data, (b) adjusted hardware counts, and corresponding software tail execution counts.

Eliminating Operations From Hyperblocks

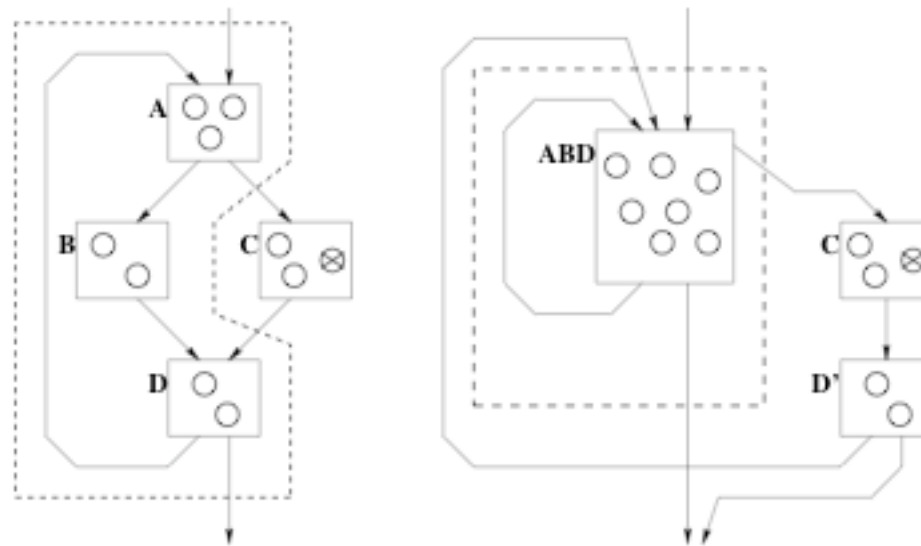


Figure 1.3: Hyperblock: exposing operation parallelism while excluding uncommon paths. When used by `garpcc`, only the hyperblock ABD is executed using reconfigurable hardware while other code is executed in software. Basic block D must be duplicated as D' since a hyperblock cannot be re-entered.

Reasons for Elimination

■ Hardware Infeasible Loops

- ◆ Subroutine Calls
 - Stack Operations and Control Flow
- ◆ Floating Point Arithmetic
 - FP Circuits Bigger Than Garp
- ◆ Operations On 64 bit Data Values
 - Again too large
- ◆ Generalized Division or Remainders (Powers of 2 can shift)
 - Again too large
- ◆ Compiler Built in Functions
 - Can't form circuits

■ Inner Loops

- ◆ Treated As A Main Loop And Start Over

Simulated Speedups

- Published In 1997 Paper

Benchmark	167 MHz SPARC	133 MHz Garp	ratio
DES encrypt of 1 MB	3.60 s	0.15 s	24
Dither of 640 × 480 image	160 ms	17 ms	9.4
Sort of 1 million records	1.44 s	0.67 s	2.1

Figure 14: Benchmark results. The times for Garp are obtained from program simulation.

Candidate Loops

Table 1. Execution time breakdown in cycles. Categories explained in text.

Test case	Single Exit Loops	Multi exit Loops	Hyperblock Loops	Unfruitful Loops	Other	Total
gnip C source	530149 33.1%	586173 36.6%	143449 9.0%	134213 8.4%	209459 13.1%	1603443 100.0%
gnip English text	601187 28.6%	662164 31.5%	218534 10.4%	179781 8.6%	439634 20.9%	2101290 100.0%
c++ input 1	2949104 20.3%	2158983 14.8%	8423327 57.6%	213459 1.5%	878407 6.0%	14633280 100.0%
c++ input 2	1092072 18.5%	894918 15.2%	2179589 37.0%	265824 4.5%	1463763 24.8%	5896166 100.0%

Final Performance Comparisons

Callahan's Thesis

Benchmark	Speedup vs. GCC	Speedup vs. SUIF	% HW Compute	% HW Overhead	# Kernels	# Kernel Executions
go	0.82	0.84	5.44%	15.06%	96	525491
m88ksim	1.05	1.06	18.15%	1.52%	15	152221
gcc	0.80	0.99	2.10%	2.17%	147	112796
compress	0.98	1.04	17.53%	7.48%	5	654763
li	0.94	1.01	0.01%	0.00%	3	185
jpeg	1.03	1.09	6.40%	3.39%	42	333691
perl	0.82	1.01	0.55%	0.18%	8	10659
vortex	0.94	0.99	0.16%	0.17%	20	10644
wavelet-image	2.60	2.80	66.74%	14.43%	11	8892
mpeg2decode	0.97	1.04	8.02%	0.88%	18	129908
pegwit	1.04	1.07	4.65%	0.94%	11	46498
gzip	1.19	1.51	38.87%	9.38%	15	30110
cpp	1.14	1.25	7.75%	9.81%	35	41128

Table 9.2: Benchmark execution on Garp.

Effect of Cache Depth Under Array

Benchmark	Miss rate percentage							
	1	2	4	8	16	64	128	perfect
go	60.728	38.448	19.176	7.457	2.957	0.020	0.018	0.018
m88ksim	73.447	4.032	0.011	0.010	0.010	0.010	0.010	0.010
gcc	37.263	17.262	4.431	1.503	1.301	0.604	0.130	0.130
compress	0.002	0.001	0.001	0.001	0.001	0.001	0.001	0.001
li	2.703	1.622	1.622	1.622	1.622	1.622	1.622	1.622
jpeg	2.433	1.144	0.953	0.890	0.331	0.013	0.013	0.013
perl	60.784	1.379	0.159	0.150	0.150	0.150	0.150	0.150
vortex	29.979	7.788	0.658	0.188	0.188	0.188	0.188	0.188
wavelet-image	20.434	0.247	0.135	0.124	0.124	0.124	0.124	0.124
mpeg2decode	16.562	3.623	1.049	0.023	0.014	0.014	0.014	0.014
pegwit	29.840	27.203	9.252	0.047	0.047	0.047	0.047	0.047
gzip	72.089	24.268	0.063	0.053	0.053	0.053	0.053	0.053
cpp	42.689	29.911	12.082	3.351	0.246	0.085	0.085	0.085

Table 9.3: Configuration miss rate percentages for different configuration cache sizes

Assuming Effectively Infinite Cache

Benchmark	Speedup vs. normal	Speedup vs. SUIF	% HW Compute	% HW Overhead	# Kernels	# Kernel Executions
go	1.14	0.96	6.13	5.45	96	525491
m88ksim	1.00	1.06	18.15	1.52	15	152221
gcc	1.01	1.00	2.11	1.25	147	112796
compress	1.00	1.04	17.53	7.48	5	654763
li	1.00	1.01	0.01	0.00	3	185
ijpeg	1.00	1.09	6.41	3.26	42	333691
perl	1.00	1.01	0.55	0.18	8	10659
vortex	1.00	0.99	0.16	0.14	20	10644
wavelet-image	1.00	2.80	66.74	14.43	11	8892
mpeg2decode	1.00	1.04	8.02	0.74	18	129908
pegwit	1.01	1.08	4.67	0.43	11	46498
gzip	1.00	1.51	38.87	9.38	15	30110
cpp	1.04	1.31	8.07	6.25	35	41128

Table 9.5: Benchmark execution on Garp with 128-level (effectively infinite) configuration cache.

Breakdown of Execution Time in Kernel

	hw	straight	outer	excluded	swloop	library	os-kernel
go	16.23	30.35	0.20	1.12	52.03	0.05	0.02
m88ksim	30.71	51.24	0.01	0.02	14.47	3.24	0.30
gcc	6.82	39.11	3.48	0.12	38.59	5.04	6.84
compress	28.27	29.21	8.33	0.00	33.94	0.17	0.08
li	0.02	36.96	1.07	0.00	58.94	0.24	2.77
jpeg	20.79	3.66	0.56	0.48	72.24	1.47	0.80
perl	2.35	25.63	0.34	0.00	12.21	14.68	44.79
vortex	0.70	84.98	0.06	0.01	4.49	9.47	0.29
wavelet-image	94.91	0.01	0.08	1.07	2.72	0.96	0.25
mpeg2decode	13.60	20.66	0.15	0.00	65.53	0.02	0.03
pegwit	4.71	87.59	0.00	0.00	5.57	1.78	0.35
gzip	67.94	8.06	4.38	0.97	17.61	0.86	0.18
cpp	39.52	5.13	1.41	1.20	39.29	12.95	0.50

Table 9.6: Breakdown of original software execution time by category.

For Loops that Could Not Be Accelerated

	total	low-iter	low-total	size	vmult	div-rem	float	call	misc
go	52.03	13.65	0.67	1.11	1.38	0.03	0.00	32.86	2.34
m88ksim	14.47	5.94	0.00	0.00	0.20	0.00	0.00	0.76	7.57
gcc	38.59	4.39	0.67	0.67	0.79	1.34	0.00	23.27	7.46
compress	33.94	0.00	0.00	0.39	0.00	0.00	0.00	33.55	0.00
li	58.94	12.10	0.00	6.20	0.00	0.00	0.00	14.20	26.44
jpeg	72.24	1.02	0.01	36.81	24.08	0.00	0.00	9.85	0.46
perl	12.21	0.09	0.02	0.00	0.35	0.00	0.00	8.25	3.50
vortex	4.49	0.25	0.01	0.00	0.00	0.89	0.00	2.90	0.44
wavelet-image	2.72	2.71	0.01	0.00	0.00	0.00	0.00	0.00	0.00
mpeg2decode	65.53	0.75	0.00	12.27	23.70	0.01	0.00	1.39	27.40
pegwit	5.57	0.22	0.09	3.38	0.00	0.00	0.00	1.88	0.00
gzip	17.61	0.41	0.04	0.92	0.17	0.00	0.00	16.06	0.00
cpp	39.29	1.75	0.02	31.81	0.04	0.00	0.00	5.03	0.63

Table 9.7: Percentage of execution time spent in loops that could not be accelerated using the array.

Instruction Level Parallelism Summary

- Automatic Parallelization Has Been Pursued Over 30 Years
- ISE Generation Getting Better: Can Create Better Hardware Support
- Performance Results Still Mixed
- Next, Look At Raising Level of Abstraction For to Programming Model
 - ◆ How to Partition Tasks/Threads
 - ◆ Operating System Support
 - ◆ Automatic Generation Of Hw/Sw Interfaces