Increasing the Number of Effective Registers in a Low-Power Processor Using a Windowed Register File

Rajiv A. Ravindran, Robert M. Senger, Eric D. Marsman,
Ganesh S. Dasika, Matthew R. Guthaus, Scott A. Mahlke, Richard B. Brown
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122
{rravindr,rsenger,emarsman,gdasika,mguthaus,mahlke,brown}@eecs.umich.edu

ABSTRACT

Low-power embedded processors utilize compact instruction encodings to achieve small code size. Instruction sizes of 8 to 16 bits are common. Such encodings place tight restrictions on the number of bits available to encode operand specifiers, and thus on the number of architected registers. The central problem with this approach is that performance and power are often sacrificed as the burden of operand supply is shifted from the register file to the memory due to the limited number of registers. In this paper, we investigate the use of a windowed register file to address this problem by providing more registers than allowed in the encoding. The registers are organized as a set of identical register windows where at each point in the execution there is a single active window. Special window management instructions are used to change the active window and to transfer values between windows. The goal of this design is to give the appearance of a large register file without compromising the instruction encoding. To support the windowed register file, we designed and implemented a novel graph partitioning based compiler algorithm that partitions virtual registers within a given procedure across multiple windows. On a 16-bit embedded processor with a parameterized register window, an average of 10% improvement in application performance and 7% reduction in system power was achieved as an eight-register design was scaled from one to four windows.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: [Real-time and Embedded Systems]; D.3.4 [**Programming Languages**]: Processors—*Code generation, Optimization, Retargetable compilers*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Embedded processor, graph partitioning, instruction encoding, low-power, register window, window assignment

1. INTRODUCTION

In the embedded processing domain, power consumption is the dominant design concern. Designers are being pushed to create processors that operate for long periods of time on a single battery. To this end, a common approach is to employ a 16-bit design, such as the Motorola-68HC12 [21]. These processors offer the advantage of extremely compact code and thus smaller memory requirements. Further, 8- and 16-bit data is common in embedded applications. Thus, these processors provide more efficient designs with datapaths optimized for narrow precision data.

While the efficiency of 16-bit processors is high, the performance of these systems can be problematic. Many embedded applications such as signal processing, encryption, and video/image processing, have significant computational demands. Low-power designs are often unable to meet the desired performance levels for these types of applications. In this paper, we focus on one particular aspect in the design of 16-bit processors, the architected registers. With a 16-bit instruction set, the number of bits to encode source and destination operand specifiers often limits the number of architected registers to a small number (e.g., eight or less). For example, TMS320C54x [26] has 8 address registers and ADSP-219x [2] has 16 data registers. Such a small number of registers often limits performance by forcing a large fraction of program variables/temporaries to be stored in memory. Spilling to memory is required when the number of simultaneously live program variables and temporaries exceeds the register file size. This has a negative effect on power consumption as more burden is placed on the memory system to supply operands each cycle.

Our approach is to provide a larger number of physical registers than allowed by the instruction set encoding. This approach has been designed and implemented within the low-power, 16-bit WIMS (Wireless Integrated Microsystems) microcontroller [22]. The registers are exposed as a set of identical register windows in the instruction set. At any point in the execution, only one of the windows is active, thus operand specifiers refer to the registers in the active window. Special instructions are included to activate windows and move data between windows. The goal is to provide the appearance of a large monolithic register file by

CASES'03, Oct. 30–Nov. 2, 2003, San Jose, California, USA.

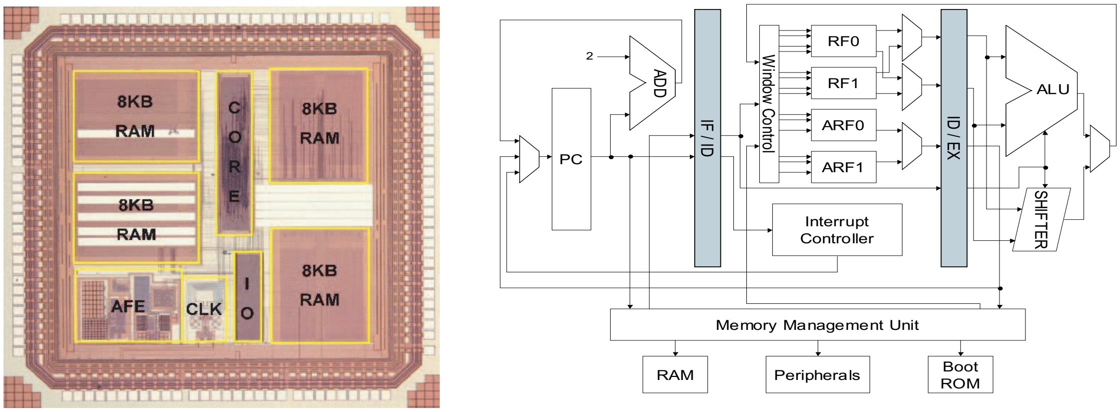


Figure 1: The WIMS microcontroller in TSMC 0.18μm CMOS and the WIMS pipeline.

judiciously employing the register window.

Traditionally, register windows have been used to reduce the register save and restore overhead at procedure calls or interrupts, such as in the SPARC architecture [24]. Our studies have shown that for the more loop-dominated applications found in the embedded domain, the use of register windows in this manner has limited impact on performance. The central problem is that programs spend most of their time in loop nests contained within a single procedure [9]. Thus, the overhead due to register spills dominates the save and restore code at procedure boundaries. Our approach is to make use of multiple register windows within a single procedure to effectively address the problem of spill code.

To support intra-procedural window assignment, the compiler employs a novel graph partitioning technique. A graph of virtual registers is created and partitioned into window groups. In the graph, each virtual register is a node and edges represent the affinity (the desire to be in the same window) between registers. Spill code is reduced by aggressively assigning virtual registers to different windows, hence exploiting the larger number of physical registers available. However, window maintenance overhead in the form of activating windows (also known as window swap) and moving data between windows (also referred to as inter-window moves) can become excessive. Thus, the register partitioning technique attempts to select a point of balance whereby spills are reduced by a large margin at a modest overhead of window maintenance.

2. WINDOWED ARCHITECTURE

2.1 WIMS Microcontroller Overview

The WIMS Microcontroller [22] was designed at the University of Michigan to control a variety of miniature, low-power embedded sensor systems. The microcontroller fabricated in TSMC 0.18μm CMOS is shown in Figure 1 and consists of three major sub-blocks: the digital core, the analog front-end (AFE) and the CMOS-MEMS clock reference. Power minimization was a key design constraint for each of the aforementioned sub-blocks. In addition to low-power circuit and processing techniques, system power can be significantly reduced through power-aware program compilation. The following section presents an overview of the digital core while focusing on architectural specifics relevant for power-aware compilation. Refer to Figure 1 for a block diagram of the digital core.

A 16-bit load/store architecture with dual-operand register-to-register instructions was chosen to satisfy the power and performance requirements of the microcontroller. The 16-bit datapath was selected to reduce the complexity and power consumption of the core while providing adequate precision in calculations, given that the sensors controlled by this chip require 12 bits of resolution. The datapath pipeline consists of three stages: fetch, decode, and execute. Typically in sensor applications, processing throughput requirements are minimal and power dissipation is a key design constraint; therefore clock frequencies should be kept as low as possible. A three-pipeline-stage architecture was chosen to obtain adequate performance without incurring the hardware overhead of more deeply pipelined machines. A unified 24-bit address space for data and instruction memory satisfies the potentially large storage requirements of remote sensor systems. The 16MB of supported memory is byte addressable and provides sufficient storage for program, data, and memory-mapped peripheral components. The current implementation of the core has four 8KB banks of on-chip SRAM with a memory management unit that disables inactive banks of memory. This memory topology permits simultaneous instruction fetch and data accesses to different banks of memory without stalling the pipeline.

A 16-bit WIMS instruction set was custom designed and includes seventy-seven instructions and eight addressing modes. The 16-bit instruction encoding supports a diverse assortment of instructions that would be unrealizable with just 8-bit encodings. In contrast, 32-bit instructions require twice as much power to fetch from memory and the additional 16-bits would not be efficiently utilized by the applications that typically run on low-power embedded processors. The 16-bit encoding represents an intelligent compromise between the power required to fetch an instruction from memory and the versatility of the instruction set. Some two-word instructions are necessary to support 24-bit absolute addressing modes with 16-bit instructions. Address update modes facilitate manipulation of the 24-bit addresses stored in the address registers by allowing both pre- and post-update operations. Load and store instructions are available with or without update and in word or byte mode.

The core contains sixteen 16-bit data registers that are split into two register windows each containing eight data

```
for ( i = 0;  i < 100;  i ++ ) {
   a [i] = b[i]  *  c[i] + d[i]
}
```

(a)

```
loop:
ADD    GPR1-3, GPR1-0, GPR1-6
LOAD   GPR1-2, [GPR1-3]
ADD    GPR1-3, GPR1-0, GPR1-7
LOAD   GPR1-4, [GPR1-3]
MPY    GPR1-3, GPR1-2, GPR1-4
ADD    GPR1-2, GPR1-0, GPR1-5
ADD    GPR1-1, GPR1-1, #1
LOAD   GPR1-4, [GPR1-2]
ADD    GPR1-2, GPR1-3, GPR1-4
STORE  [GPR1-0], GPR1-2
ADD    GPR1-0, GPR1-0, #4
CMP    GPR1-1, #100
BRCT   loop
```

(b)

```
loop:
LOAD     GPR1-1, [SP, #24]
ADD      GPR1-0, GPR1-3, GPR1-1
LOAD     GPR1-0, [GPR1-0]
LOAD     GPR1-1, [SP, #32]
STORE    [SP, #72], GPR1-0
ADD      GPR1-0, GPR1-3, GPR1-1
LOAD     GPR1-0, [GPR1-0]
LOAD     GPR1-1, [SP, #72]
MPY      GPR1-0, GPR1-1, GPR1-0
STORE    [SP,#40], GPR1-0
LOAD     GPR1-0, [SP, #16]
ADD      GPR1-1, GPR1-3, GPR1-0
LOAD     GPR1-0, [GPR1-1]
LOAD     GPR1-1, [SP, #40]
ADD      GPR1-0, GPR1-1, GPR1-0
LOAD     GPR1-1, [SP, #80]
STORE    [GPR1-3], GPR1-0
ADD      GPR1-0, GPR1-1, #1
ADD      GPR1-3, GPR1-3, #4
CMP      GPR1-0, #100
BRCT     loop
```

(c)

```
loop:
IW_MOV      GPR1-0, GPR2-1
WIN_SWAP    GPR, #1
ADD         GPR1-3, GPR1-2, GPR1-0
IW_MOV      GPR1-0, GPR2-2
LOAD        GPR1-1, [GPR1-3]
ADD         GPR1-3, GPR1-2, GPR1-0
LOAD        GPR1-0, [GPR1-3]
MPY         GPR1-3, GPR1-1, GPR1-0
IW_MOV      GPR1-1, GPR2-3
ADD         GPR1-1, GPR1-2, GPR1-1
LOAD        GPR1-1, [GPR1-0]
ADD         GPR1-0, GPR1-3, GPR1-1
STORE       [GPR1-2], GPR1-0
WIN_SWAP    GPR, #2
ADD         GPR2-0, GPR2-0, #1
WIN_SWAP    GPR, #1
ADD         GPR1-2, GPR1-2, #4
WIN_SWAP    GPR, #2
CMP         GPR2-0, #100
BRCT        loop
```

(d)

Figure 2: Register window example (a) C-source (b) 1-window of 8-registers (c) 1-window of 4-registers (d) 2-windows of 4-registers each. The number following the hyphen '-', denotes the allocated register number.

registers (RF0, RF1). Similarly, four 24-bit address registers are evenly split into two register windows (ARF0, ARF1). This windowing scheme permits instructions to be encoded in 16 bits by reducing the number of bits required to encode the sixteen register operands from 4 bits to 3 bits. In general, instructions can access only one register window at a time. The only exceptions are the non-windowed instructions which are used to copy data and addresses between the two windows. A window bit stored in the Machine Status Register (MSR) selects the active register window. Additional window bits can be added to the MSR to support extra register windows. A special instruction switches register windows in a single cycle by changing the MSR window bit setting. Three additional non-windowed address registers (a stack pointer, frame pointer, and link register) are used by the compiler for subroutine and stack support.

2.2 Windowed Register File Example

In order to demonstrate the benefits of register windowing for reducing spill code while incurring the overhead of the window management instructions, consider the example shown in Figure 2. The original C-source is shown in Figure 2(a). This has been mapped to 3 different register window configurations: Figure 2(b) shows 1-window of 8-registers, Figure 2(c) shows 1-window of 4-registers and Figure 2(d) shows 2-windows of 4-registers per window. For clarity, we use a generic RISC-like instruction set instead of the WIMS instruction set. The leftmost operand is the destination. *GPR* stands for the integer register file. *GPR1, GPR2* denotes the first and second window of the integer register file, respectively. *WIN_SWAP* is the window swap operation. The first operand specifies the register file (integer, floating, address, etc.) while the second argument specifies the target window number. The *WIN_SWAP* instruction sets the machine status register bit to activate the target window. All subsequent operations then access their operands from the new window. For example, the instruction *WIN_SWAP GPR, #1* sets the current active window to the 1^{st} window of the integer register file. The spill code is shown by all *SP* relative load/store instructions (shown in bold). *IW_MOV* denotes the inter-window move instruction which can move values between any two register windows.

In Figure 2(b), all program variables and temporaries can fit in registers and hence no spill is generated with 8-registers. Conversely with 4-registers, significant spill code is generated because there are insufficient registers to hold the necessary values as shown in Figure 2(c). In Figure 2(d), by partitioning the variables and temporaries into 2-windows of 4-registers, no spill is generated although there is an overhead of 4-window-swaps and 3-inter-window moves (all shown in bold). On the WIMS processor where every instruction shown takes a single cycle, Figure 2(c) has an 8-cycle overhead as compared to Figure 2(b) while Figure 2(d) has only 7 extra instructions. Also, (d) consumes less power as it has fewer loads and stores (0 spill operations) to memory as compared to (c) (8 spill operations).

3. REGISTER WINDOW PARTITIONING

3.1 Overview

The overall compilation system for register window partitioning is shown in Figure 3. Ignoring the gray boxes, the base compiler system consists of the frontend which does profiling, classical code optimizations (such as CSE, constant folding, induction variable elimination, etc.), loop unrolling and procedure inlining to produce a generic RISC-like assembly language code. The assembly code uses an infinite supply of virtual registers (VRs) to communicate values between operations. A machine description file (MDES) is used to describe the architecture of the target machine for generating machine-specific assembly code. The MDES contains a detailed description of the register files including the windows into which each file is partitioned, number of registers, width of the register files, connectivity of register files to functional units, instruction format, and a detailed resource usage model which is used by the instruction scheduler. The connectivity model helps the compiler's code generator conform to the architectural specifications

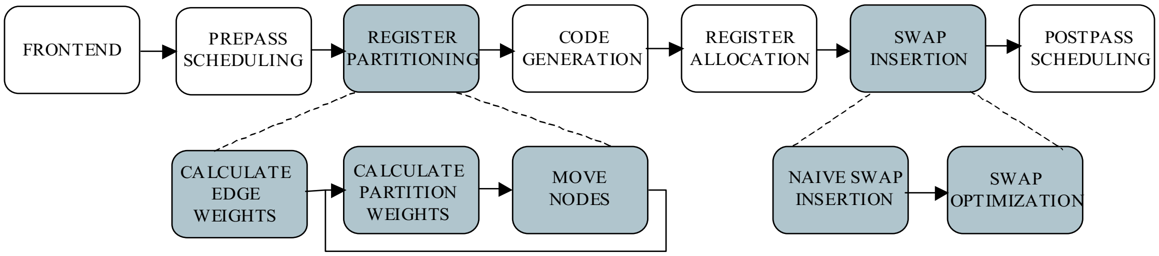


Figure 3: Overview of the compiler system.

of the target machine. For the WIMS architecture, a given operation at any instance can source its operands only from a single register window. After prepass scheduling, these VRs are partitioned into the available register windows. For each register file, the register allocator uses a graph coloring based algorithm [16] to assign physical registers to the VRs, generating spills if required. Finally, the resultant code is scheduled (postpass).

The new phases added to handle register window partitioning are shown in gray boxes in Figure 3. Register partitioning treats each window/partition as a separate register file and binds VRs allocated to a given partition to the corresponding register file. The partitioning algorithm could assign VRs referenced by a given operation to different windows. The code generator, using the register file connectivity information from the MDES, inserts appropriate inter-window moves to honor the architectural constraints. The swap insertion phase inserts window swaps in the code so that two operations that access different register windows are separated by a window swap. The swap optimizer then removes the redundant swaps and looks for opportunities to eliminate swaps by combining them with other operations. Prior to postpass scheduling, an edge drawing phase inserts control dependency edges between the swap operations and the operations that precede or follow it so that the postpass scheduler does not move operations across swaps.

The register window partitioning algorithm is modeled as a graph partitioning problem where the nodes in the graph correspond to virtual registers (VRs) used in the assembly code and the edges represent the affinity between VRs. The goal of the graph partitioning based approach is to partition the VRs into different register windows/partitions so as to minimize the overall spill, inter-window moves, and window swaps.

Partitioning consists of two distinct phases - weight calculation and node assignment. Each partition is assigned a weight which measures the cost of spilling the VRs assigned to that partition. The affinity between VRs is captured using edge weights. The edge affinity represents the penalty incurred if two nodes connected by the edge are assigned to different partitions. The penalty can be an inter-window move, window swap, or both. If two VRs referenced within a given operation are assigned to different partitions, the code generator is forced to insert an inter-window move. Similarly, if two VRs in two different operations are assigned to different windows, a window swap is required at some point between the two operations. Unlike traditional graph partitioning, which uses statically computed node weights, the partitioning algorithm uses partition weights that change dynamically during the partitioning process.

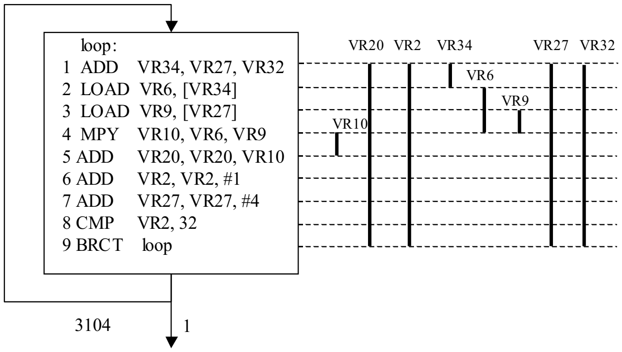

```
  loop:
1  ADD   VR34, VR27, VR32
2  LOAD  VR6, [VR34]
3  LOAD  VR9, [VR27]
4  MPY   VR10, VR6, VR9
5 ADD    VR20, VR20, VR10
6 ADD    VR2, VR2, #1
7 ADD    VR27, VR27, #4
8 CMP    VR2, 32
9 BRCT   loop
```

Figure 4: FIR loop with virtual register live-ranges.

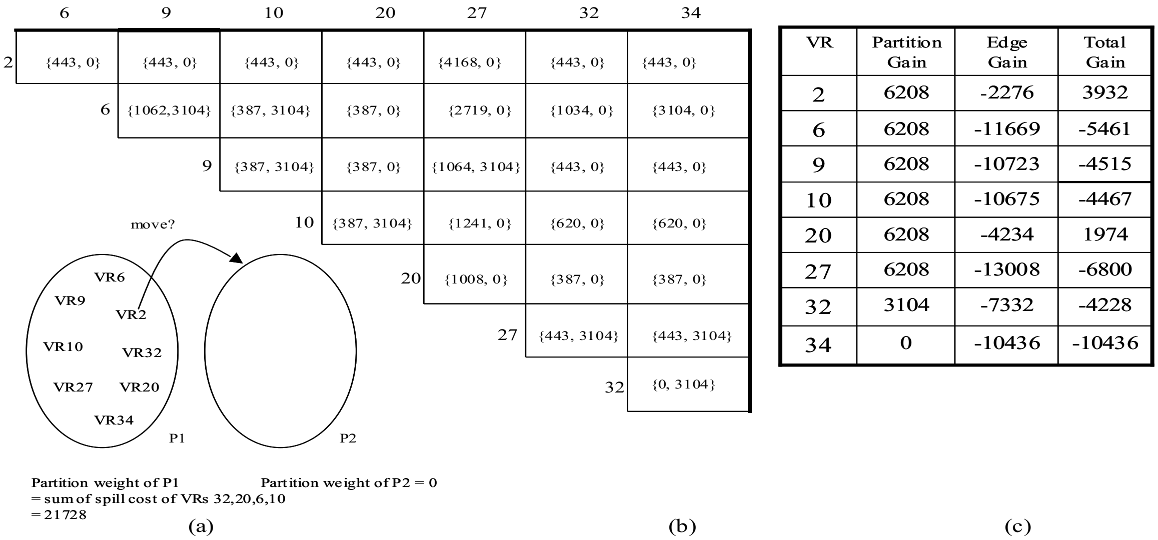
The node assignment phase uses the calculated weights to consider moving nodes between partitions so as to minimize the sum of all the partition weights and the edge weights between every partition. The register partitioning algorithm uses a modified version of the Kernighan-Lin [15] graph partitioning algorithm. The partitioning algorithm is region-based[1], i.e all the VRs in the most frequently executed region are partitioned, followed by the VRs in the next most frequently executed region and so on. The node assignment phase must ensure that the partitioning decisions are honored across all regions.

Figure 4 is a code segment from the inner-most loop of the FIR filter. The dynamic execution frequency, obtained from profiling the application on a sample input, is 3104. This example will be used throughout this section to illustrate the weight calculation and node assignment process. We try to partition this region into 2-windows of 4-registers each. In this work, profile information is used in the edge and partition weight calculations. Alternately, static weights based on the nesting depth of loops can also be used.

3.2 Weight Calculation

Edge Weight: An edge is associated with every pair of VRs. The edge weight is used by the partitioning algorithm to measure the degree of affinity between two VRs. Higher edge weights imply greater affinities. The algorithm tries to place two nodes with high affinity in the same partition while trying to minimize the sum of the edge weights between nodes placed in different partitions. An edge weight is an estimate of the number of dynamic moves and swaps required between two VRs. By placing two VRs with high affinity in a single partition, the algorithm reduces the num-

[1] A region is any block of code considered as a unit for scheduling like a basicblock, superblock [12] or hyperblock.



	6	9	10	20	27	32	34
2	{443, 0}	{443, 0}	{443, 0}	{443, 0}	{4168, 0}	{443, 0}	{443, 0}
6		{1062,3104}	{387, 3104}	{387, 0}	{2719, 0}	{1034, 0}	{3104, 0}
9			{387, 3104}	{387, 0}	{1064, 3104}	{443, 0}	{443, 0}
10				{387, 3104}	{1241, 0}	{620, 0}	{620, 0}
20					{1008, 0}	{387, 0}	{387, 0}
27						{443, 3104}	{443, 3104}
32							{0, 3104}

VR	Partition Gain	Edge Gain	Total Gain
2	6208	-2276	3932
6	6208	-11669	-5461
9	6208	-10723	-4515
10	6208	-10675	-4467
20	6208	-4234	1974
27	6208	-13008	-6800
32	3104	-7332	-4228
34	0	-10436	-10436

Figure 5: Partitioning applied to FIR. (a) Initial partition (b) Edge weight matrix with swap cost on the left and move cost on the right (c) Gains for each VR for the initial partition.

ber of swaps and moves. The edge weight between VRs is expressed as a matrix (see Figure 5(b)) computed prior to the node assignment process. The edge weight is the sum of two components - the estimated move cost and swap cost.

Move cost: An operation may only reference registers from a single window. Thus, VRs referenced by a single operation that are assigned to different partitions require an additional inter-window move (*IW_MOV*) operation. For every pair of VRs, the total number of dynamic instances of operations in which both the VRs occur is computed. Each operation would need a move if the VRs referenced by it were assigned to different partitions. Thus, the total dynamic instances of each operation is the estimated move cost. In Figure 4, VRs 6 and 9 occur in operation 4 and so the move cost is 3104. This process is carried out for all pairs of VRs producing the matrix of values in Figure 5(b) (right entry in each cell).

Swap cost: If two VRs are assigned to two different windows/partitions, a window swap (*WIN_SWAP*) is required before the operation that refers to the second VR. Swap cost estimates the number of swaps required between every pair of VRs assuming that they are assigned to different partitions. For every pair of VRs, the region is scanned in linear order. On reaching the first VR, the current active window is assumed to be 1. On encountering the second VR, the current active window becomes 2 and hence a swap is required right before the operation which references the second VR. Continuing further, on seeing an instance of the first VR again, the active window changes and another swap is required. No swap is required on reaching another instance of the recent most seen VR. At the end of the region, the total number of swaps seen so far gives an estimate of the number of swaps required between this pair of VRs.

Adding swap cost between every pair of VRs can overestimate the importance of swaps as the number of swaps is a function of the partition assignment of all the VRs and not just between two VRs. For example, consider operations 3 and 4 in Figure 4. If VRs 9 and 27 are assumed to be in window 1 and VRs 10 and 6 in window 2, the above method would count the swap four times, between 9 and 10, 9 and 6, 27 and 10, and 27 and 6, although in reality only a single swap is necessary.

To deal with the swap over-counting, we use swap counts to normalize the swap cost between every pair of VRs. The swap count is the number of swaps between every pair of operations due to every pair of VRs. For example, between operations 6 and 7, 5 swaps are required. These swaps are due to VR pairs 10 and 27, 20 and 27, 2 and 27, 27 and 9, and 27 and 6. Similarly, between operations 3 and 4, 5 swaps are required - these are due to VR pairs 10 and 27, 10 and 32, 10 and 34, 27 and 6, and, 9 and 6. Generalizing this for 2-windows, let $c_1, c_2 ... c_k$ be the swap count due to swaps required by all pair of VRs after operations $op_1, op_2 ... op_k$. If two VRs vr_i and vr_j, require a swap after these k operations, then the normalized *swap cost estimate* between vr_i and vr_j is $(1/c_1 + 1/c_2 + ... + 1/c_k) * cost\_of\_swap$, where $cost\_of\_swap$ is the cost of a single swap operation which is just the dynamic frequency count of the region where the swap would be inserted. Intuitively, a swap after an operation could be shared by multiple VR pairs. So, if n VR pairs introduce a swap after an operation, then the contribution to the swap cost by any one of those VR pairs is $1/n$. In Figure 4(a), VRs 10 and 27 require a swap after operations 3 and 6. Since operation 3 has a swap count of 5 (due to the 5 pair of VRs including 10 and 27 listed above) and operation 6 also has a swap count of 5 (including 10 and 27), the swap cost estimate between VRs 10 and 27 is $(1/5 + 1/5) * 3104 = 1241$ (Figure 5(b), left entry in each cell). Beyond 2-windows, multiple swaps may be required after an operation for swapping between different windows. For n-windows, we pessimistically assume a swap for every pair of windows and so multiply the *swap cost estimate* by $n * (n - 1)/2$.

Partition Weight: The partition weight estimates the spill cost when VRs assigned to the partition are allocated to the register window corresponding to that partition. The node assignment phase tries to minimize the sum of the weights of all the partitions. Only the VRs in the current region are considered. The partition weights are computed using a crude priority function based on register allocation using graph coloring.

Given a set of VRs assigned to a partition, the live-ranges of the VRs are computed. For each VR, its dynamic reference count is calculated. The dynamic reference count is the dynamic execution frequency of the region times the number of occurrences of the VR as either a source or a destination operand within an operation in that region. If the VR is spilled, then the dynamic reference count gives an estimate of the load/store overhead for every instance of that VR. For every VR, the estimated dynamic cycles associated with spilling the VR is computed. For every operation in the region spanned by the live-range of the VRs under consideration, the interfering VRs are considered as candidates for spill. If the number of overlapping live-ranges for that operation is more than the number of registers in that partition (size of the register window)[2], the interfering VRs are spilled until the overlapping live-range is less than the register window size. Note, we are only estimating the weight of the partition by mimicking the process of spilling without actually inserting the spill code. The actual spill code insertion is done during register allocation within each window after the window assignment process. The VRs are chosen in increasing order of dynamic reference count. If two VRs have the same dynamic reference count value, the one with larger live-range is spilled. Once a VR is spilled, it no longer interferes with the rest of the operations and hence is not considered for subsequent operations if there is an overlap. The cost of the partition is the sum of the dynamic reference counts of the spilled VRs. If a VR is already assigned to the register window (from an earlier region) then, for the rest of the VRs interfering with this VR, the number of available registers is one less than the partition size.

In Figure 4, the live-ranges of the VRs are shown on the right. Assume that all VRs are assigned to a single partition. Operation 1 has 5 VRs (20,2,34,27,32) live simultaneously. Since there are only 3 registers available in the partition, VRs 32 and 20 are spilled. VR 32 has a spill cost of 3104 as there is only a single reference of that VR in operation 1, while other VRs are referenced more than once and have spill cost greater than 3104. Hence, VR 32 is picked first. VRs 20 and 34 both have a dynamic reference count of 6208, but VR 20 has a larger live-range and so is chosen for spilling. At operation 2, VRs 20, 2, 34, 6, 27 and 32 are live. Since 32 and 20 are already spilled, therefore only 6 gets spilled as 6 has a smaller dynamic reference count than VRs 2 and 27, and larger live-range than 34. At operation 3, VRs 20, 2, 6, 9, 27 and 32 are live. Since 32, 20, 6 are already spilled, no more VRs are spilled here as the number of remaining live VRs is equal to 3. VR 10 is spilled at operation 4. So, for this partition, the partition weight is the spill cost of the spilled VRs 32, 20, 6, 10, which is 3104+6208+6208+6208 = 21728.

3.3 Node Partitioning

The goal of the node partitioning phase is to reduce the overall spill cost while minimizing the impact due to inter-window moves and swaps. Starting from an initial partition, the graph partitioning algorithm tries to iteratively distribute VRs into different partitions so as to reduce the sum of the weights of all partitions, while trying to minimize the edge weights between nodes assigned to different partitions.

```
1)  set_of_parts = create_parts(n)
2)  num_tries = 0; total_gain = 0; did_move = true;
3)  add all unbound VRs to the first partition
4)  while (did_move == true && num_tries < MAX_TRIES) {
5)   did_move = false;
6)   src_part = find_unbalanced_partition(set_of_parts)
7)   if (src_part == NULL) {
8)    unlock all locked vrs in current set of partitions
9)    add current set of partitions to set_of_parts
10)   num_tries++
11)   continue
12)  } else {
13)   if (src_part == last_part) {
14)    remove src_part from set_of_parts
15)    continue
16)   }
17)   while (total_gain >= 0) {
18)    {gain,vr,dest_part} = find_best_vr(src_part)
19)    total_gain += gain
20)    do_move_vr(vr, dest_part)
21)    lock_vr(vr, dest_part)
22)    did_move = true
23)   }
24)   last_part = current_part
25)  }
26) }
```

Figure 6: Pseudo code for node partitioning.

```
find_best_vr(src_part) {
 for every node in src partition {
  for every dest_part in destination partition {
   move_node(node, dest_part)
   old_total_wt = src_part_wt_old + dest_part_wt_old
   new_total_wt = src_part_wt_new + dest_part_wt_new
   partition_wt_gain = old_total_wt - new_total_wt
   edge_wt_gain = old_edge_wt - new_edge_wt
   gain = partition_wt_gain + edge_wt_gain
   if (gain > maxgain) {
    bestnode = node
    maxgain = gain
   }
  }
 }
 return (maxgain,bestnode,dest_part)
}
```

Figure 7: Pseudo code to find best VR.

The node partitioning technique is a modified version of Kernighan-Lin's graph partitioning algorithm [15]. The pseudo code for the algorithm is given in Figure 6. Initially, all nodes are placed in a single partition. The motivation to start with this initial configuration was to let the partitioner use one register window as much as possible and move VRs to the other windows only when the inter-window move and window-swap cost (edge weight) is dominated by the spill cost (partition weight). During a single iteration of the outer loop of the algorithm (step 4), the partition (*src_part*) with the maximum weight is selected (*find_unbalanced_partition*). If such a partition exists (step 7) and is not the same as the previously selected partition (step 13), proceed to step 18. In step 18, from the current partition (*src_part*), the best node that can be moved to any other partition (*find_best_vr*) is found.

Figure 7 gives the pseudo code for *find_best_vr*. For every node in the source partition, *find_best_vr* computes the gain in moving the node to all other destination partitions.

[2] In our implementation, we assume the number of available register is one less than the window size. This is done to factor in the interferences due to inter-window moves that would be inserted later.

Gain is defined as the sum of the *partition_wt_gain* and *edge_wt_gain*, where *partition_wt_gain* is the reduction in total partition weights when the node is moved from the source to the destination partition. Similarly, *edge_wt_gain* is the reduction in edge weights between nodes in the source and destination partitions. In Figure 7, *src_part_wt_old/dest_part_wt_old* is the weight of the source/destination partition before the node is moved, while *src_part_wt_new/dest_part_wt_new* is the weight of the source/destination partition after the node is moved.

The partitioning algorithm then picks the node with the highest gain and performs the move (step 20). It also keeps track of the cumulative gain, which is the sum of the gains obtained during each move (step 19). Note that, it is possible for the gain of a move to be negative. But as long as the total gain is positive, the algorithm tries moving nodes from the current partition. Allowing negative gains helps avoid local minima. When the total gain is negative (i.e no more moves are possible from the current partition, step 17), it picks the next most unbalanced partition. If the new partition is same as the old partition (step 13), it disregards the current partition (step 14) and tries a new one. Also, once a node is moved over to the new partition, it is locked in the new partition and not considered in the current iteration (step 21). The inner loop (steps 17-23) changes the current partition configuration to generate a new partition configuration.

Figure 5(a) gives the initial configuration. The partition weight of P1 is 21728 while that of P2 is 0 (as there are no VRs assigned to P2). The algorithm tries to move each VR from P1 to P2 and computes the resultant partition and edge gains. Figure 5(c) shows the gain for each VR. The partition gain (shown in column 2 of Figure 5(c)), is the reduction in total spill when the given VR is moved from P1 to P2. The edge gain is shown in column 3 of Figure 5(c). Since all VRs are in P1, any move results in a positive edge weight between P1 and P2 and hence a negative edge gain. The VR with the maximum total gain (partition gain + edge gain, as shown in column 4 of Figure 5(c)) is chosen. Here, VR 2, with a total gain of 3932 is moved to partition 2.

Once all the partitions are done (step 7) in a single iteration of the outer loop, all nodes are unlocked (step 8). This allows nodes that were moved in the current iteration to move back to their original partition in the next iteration. If any move was performed in this iteration, the algorithm loops back to step 4 with the new partition configuration to start the next iteration. From the new configuration, the partition with the greatest weight is selected and the process continues. A node could possibly go back-and-forth multiple times between two partitions before eventually settling in one. There is an upper threshold (MAX_TRIES) on the number of runs so that the algorithm does not fall into an infinite loop. Experimental results have shown that this helps to overcome some of the greedy decisions made during a single run. Also, in our experiments the total iterations of the outer loop never exceeded more than 4.

Edge weights are computed statically before partitioning. So, *find_best_vr* need only do a lookup of the edge weight matrix (Figure 5(b)) to get the edge weights between a pair of VRs. But, this is not the case with the partition weights. As nodes migrate from partition to partition, the interferences among VRs can change and so the partition weight (spill cost) has to be recomputed (section 3.2) on the fly. Figure 8 shows the final partition configuration. The total partition weight is 3104 as compared to the initial partition weight of 21728 (Figure 5(a)). The final code after register allocation and swap insertion is also shown in Figure 8. The code has 1 spill (operation 2) 4 swaps (operations 1, 8, 11, 13) and 1 inter-window move (operation 7).

3.4 Window Swap Insertion and Optimization

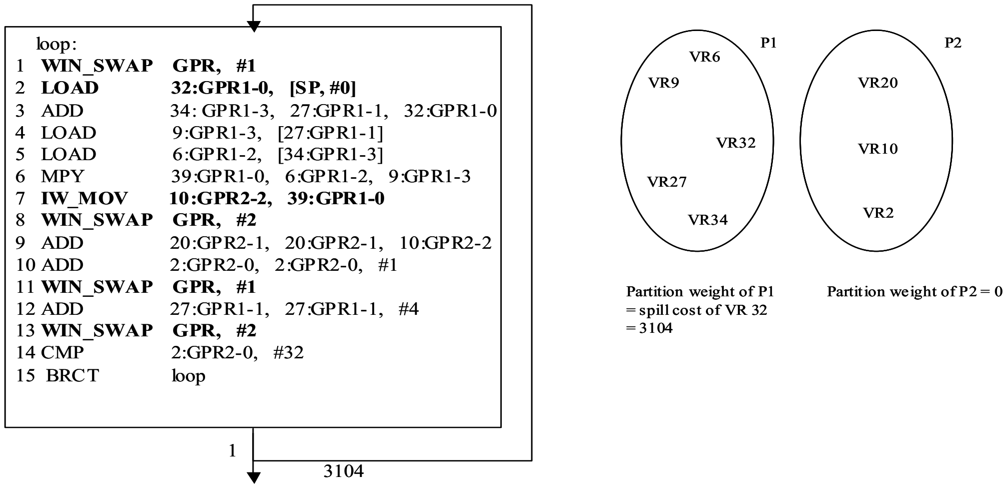
Window swap operations are inserted after window assignment and register allocation. Initially, a naïve window assignment is performed by walking the region in sequential program order. A window swap operation is inserted at the beginning of the region depending on the register window of the first operation in the block. The current register window is set to this window. Scanning each operation, if the assigned window is different from the current register window, a swap to the new window is inserted. This becomes the new current register window. Following every procedure call, a window swap operation is forced to set the current active window to the window of the operation following the procedure call. This is because we assume separate compilation and the state of the active window is unknown after a procedure return.

Swap optimizations: This naïve method inserts many unnecessary swap operations. Three swap optimizations were implemented to reduce the swap overhead.

- A swap at the beginning of a region is unnecessary if all control paths leading to that block have trailing operations which are in the same window as the first operation in the region.
- It is also possible to hoist a window swap upwards from the beginning of a more frequently occurring region to the end of less frequently occurring predecessors and thus reduce the total number of dynamic swaps. This is legal provided the new window swap operation inserted at the end of the predecessor is the last operation of that predecessor (this might not be the case for superblocks which have multiple exits)
- To prevent redundant swaps after procedure calls, the return from subroutine operation forces the window to be set to 1. So, if an operation following the procedure call is assigned to window 1, a swap is not needed.

Instruction combining: To further reduce the swap overhead, experimental studies were conducted on which operations frequently preceded a swap to identify opportunities for merging swap with regular operations. Combining must be constrained by the availability of free opcode encoding bits within the instruction encoding. Inter-window register move and window-swap was found to be a likely candidate as the WIMS move operations have free encoding bits. In Figure 8, swap operation 8 can be combined with the inter-window move operation 7. Another interesting complex operation can be formed by combining a conditional branch with a swap such that the swap would be executed only if the branch is either taken or fallthrough. The effects of operation combining are evaluated in the next section.

Edge drawing: After the swaps are inserted, control dependence edges are inserted between the swap and all operations preceding and following it. This is done so that the postpass scheduler does not intermix operations from different windows. All operations preceding the swap except



```
loop:
WIN_SWAP   GPR,  #1
LOAD       32:GPR1-0,  [SP, #0]
ADD        34: GPR1-3,  27:GPR1-1,  32:GPR1-0
LOAD       9:GPR1-3,  [27:GPR1-1]
LOAD       6:GPR1-2,  [34:GPR1-3]
MPY        39:GPR1-0,  6:GPR1-2,  9:GPR1-3
IW_MOV     10:GPR2-2,  39:GPR1-0
WIN_SWAP   GPR,  #2
ADD        20:GPR2-1,  20:GPR2-1,  10:GPR2-2
ADD        2:GPR2-0,  2:GPR2-0,  #1
WIN_SWAP   GPR,  #1
ADD        27:GPR1-1,  27:GPR1-1,  #4
WIN_SWAP   GPR,  #2
CMP        2:GPR2-0,  #32
BRCT       loop
```

Figure 8: FIR after window assignment. GPR1 denotes window 1 of the integer register file (GPR) and GPR2 denotes window 2. The number after the hyphen '-' is the allocated register number. The number before ':' is the original VR from Figure 4.

procedure calls and branches have a 0-cycle control dependency with the swap. All operations following the swap have a 1-cycle dependency with the swap and so would be executed strictly after the swap. The postpass scheduler is then invoked to schedule the swaps, inter-window moves and the spill code while honoring the new control dependences.

4. EXPERIMENTAL RESULTS

We implemented the register partitioning algorithm using the Trimaran compiler toolset, a retargetable compiler for VLIW processors [29]. For our study, only the integer register file was assumed to be windowed and so a set of integer-dominated benchmarks from a mix of Mediabench [17], MiBench [9] and Unix utilities were evaluated. All the benchmarks were compiled with control-flow profiling, superblock formation, function inlining, and loop unrolling turned on. For the experiments, the number of windows and the number of registers per window were varied to evaluate the power and performance impact. Two machine configurations were used - the WIMS processor and a 5-wide VLIW machine with the following function units: 2 integer, 1 floating-point, 1 memory, and 1 branch. For the VLIW machine, the swap instruction is assumed to be compatible with any slot in the VLIW word and thus can be assigned to any free slot. In our experiments, the floating-point unit is often free, thus the swap occupies that slot. The swap completes in the decode stage and does not occupy a function unit. The inter-window move executes on the integer unit.

We considered the power/performance improvement of a range of register file configurations consisting of 1, 2, 4, and 8 indentical windows containing 4, 8, and 16 registers per window. More specifically, the following configurations were evaluated: 2-window, 4-window, and 8-window with 4-registers per window (w2.r4, w4.r4, w8.r4) were compared against a base 1-window of 4-registers (w1.r4); 2-window and 4-window with 8-registers per window (w2.r8, w4.r8) were compared against a base 1-window of 8-registers (w1.r8); and finally, 2-window with 16-registers per window (w2.r16) was compared against a base 1-window of 16-registers (w1.r16). This helped in understanding the power/performance benefits of increasing the effective number of registers without changing the instruction set architecture. For all the experiments, window swaps were combined with an immediately preceding inter-window move whenever possible.

The graph in Figure 9 compares the 2-window with 8-registers per window case (left set of bars) and the 4-windows with 8-registers per window case (right set of bars) with the base 1-window of 8-registers for the WIMS processor. For each set of bars, the percent performance improvement in total execution cycles (leftmost bar of each set) is shown. The performance numbers were obtained by multiplying the schedule length of each region by its execution frequency to get the total dynamic execution cycles for the whole program. Since we use a single cycle memory system for the WIMS and the VLIW processors, this approach is quite accurate. The rightmost bar of each set shows two components - (i) spill benefit, which is the percent savings in total execution cycle count due to savings in spill, (ii) percent swap and move overhead, which is the percent of overall execution cycles due to the extra inter-window moves and window swaps. The numbers over each set of bars shows the percent savings in dynamic spill code as we scale the number of windows. Performance improvement is obtained when the spill benefit exceeds the swap and move overhead.

Overall, an average performance improvement of 10% is observed with 4-windows of 8 registers. Performance ranges from a maximum of a 30% gain for djpeg to a loss of 0.8% for g721dec. This range of results is due to the competing affects of spill code reduction and swap/move overhead. In djpeg, an 86% reduction in spill code (for the 4-window 8-register case) is seen, which accounts for 42% savings in total cycles. While for g721dec, there is a 77% reduction in spill code, but this contributes to only 10% savings in total cycles. This implies that the impact of spill is significantly less for g721dec in the base 1-window of 8-registers case. Since all instructions take a single cycle, any gain in performance due to savings in spill is offset by a corresponding reduction in performance due to swaps and moves. The negative performance in a few of the multi-window cases are

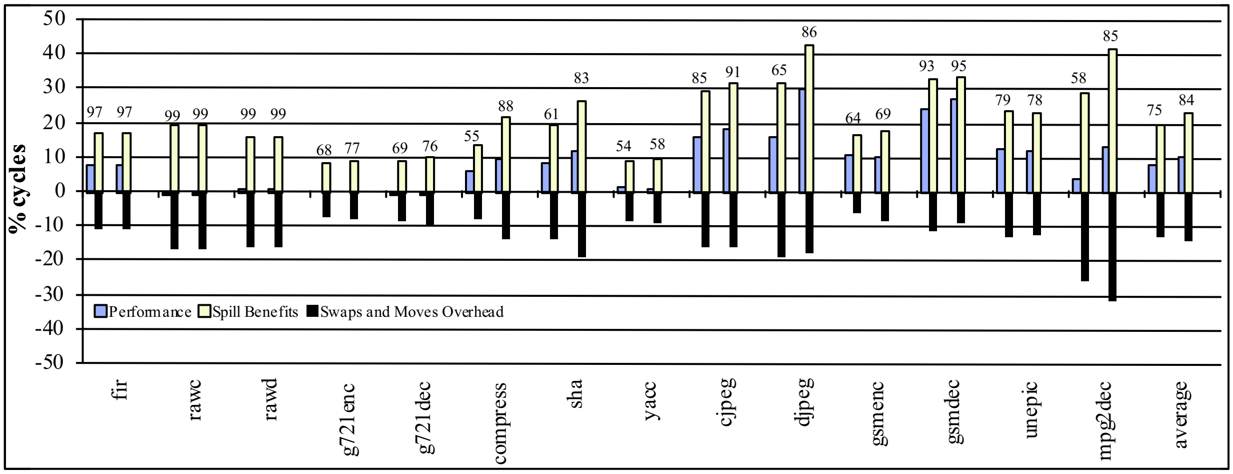


Figure 9: Effect of increasing the number of register windows for the 8-register WIMS processor. For each benchmark, two sets of data are shown: w2.r8 (left) and w4.r8 (right), plotted relative to w1.r8.

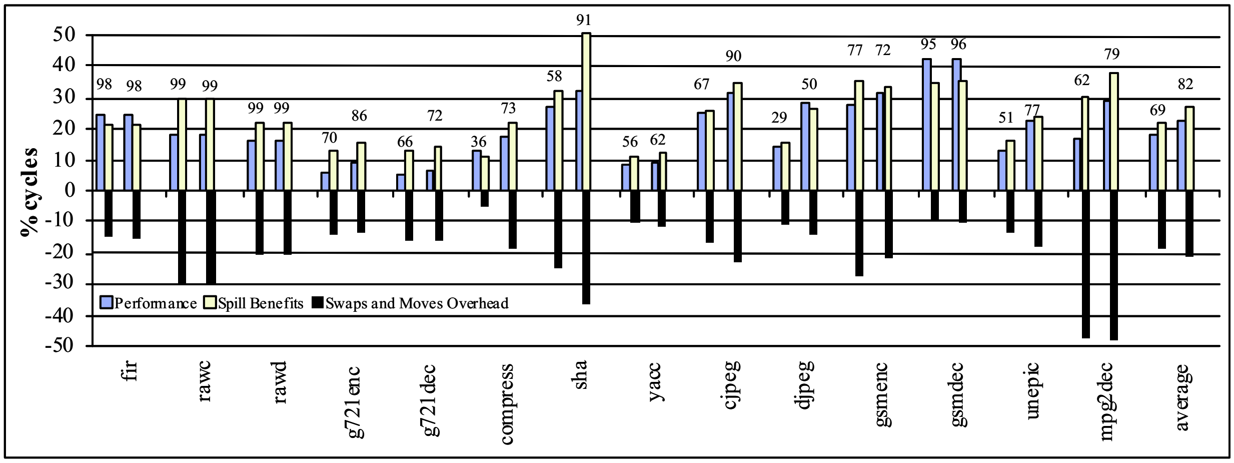


Figure 10: Effect of increasing the number of register windows for the 8-register VLIW machine. For each benchmark, two sets of data are shown: w2.r8 (left) and w4.r8 (right), plotted relative to w1.r8.

seen because we insert redundant swaps, for example, after a procedure call or at the beginning of a function, caused by separate compilation. Also, the greedy nature of the partitioning algorithm causes VRs to be moved to partitions and thus increase the swap/move cost.

The graph in Figure 10 repeats the previous experiment for the VLIW machine. We achieve an average of 22% performance improvement for the 4-window with 8-registers per window case, which is more than double the gain observed for the WIMS processor. The larger gains are due to several reasons related to the multi-issue capabilities of the VLIW machine. First, the spill code often sequentializes program execution by increasing the lengths of critical dependence chains through the code. For the VLIW machine, these critical dependence chains often determine the program execution time. Thus, the elimination of spills translates into compacter schedules and larger performance gains than for single-issue WIMS processor. Second, there is a larger demand for registers to maintain the necessary intermediate values to support the inherent instruction level parallelism. Thus, more spill code is present and its affects are more pronounced. Third, the overhead of swaps and moves is lower as they can execute in parallel with other instructions. In particular, the swap often executes in the floating-point slot making it almost ”free” for the integer dominated applications that are evaluated.

Table 1 compares the performance improvement of different window configurations with 4 and 16 registers per window for the WIMS and VLIW machines. As compared to the 8-register configurations, the w4.r4 per window case shows much improved performance as compared to the w1.r4 case as 4-registers are insufficient for both the WIMS and the VLIW machines. But the performance of w2.r16 over w1.r16 is not significant as the compiler is able to do a good job in eliminating most of the spills using 1-window of 16-registers. Djpeg, due to loop unrolling, had a high register pressure and hence benefited significantly when the number of windows were increased for all window file sizes. As with the 8-register case, the swap and move overhead outweighs the spill savings and hence a decrease in performance is observed in g721enc and g721dec.

Table 2 shows the impact that the swap-move combin-

Benchmark	**w2.r4**	**w4.r4**	**w8.r4**	**w2.r16**
fir	1.24	25.81	28.26	0.00
rawc	14.47	21.19	20.91	0.00
rawd	16.81	24.49	24.52	0.00
g721enc	-8.47	0.11	-1.11	-0.64
g721dec	-9.02	-0.98	-2.35	-0.51
compress	6.08	11.75	10.78	0.21
sha	6.54	8.07	25.38	2.86
yacc	-0.37	6.15	6.78	-0.50
cjpeg	-2.33	18.26	22.96	-0.69
djpeg	2.47	11.04	16.51	17.92
gsmenc	13.63	19.94	23.17	2.30
gsmdec	-1.36	14.34	23.42	-0.04
unepic	4.74	13.14	17.13	6.77
mpeg2dec	-2.91	5.45	10.21	8.02
average	2.96	12.7	16.18	2.55

Benchmark	**w2.r4**	**w4.r4**	**w8.r4**	**w2.r16**
fir	2.70	35.35	42.10	0.00
rawc	26.71	48.29	47.79	0.00
rawd	27.04	48.57	47.77	0.00
g721enc	-0.31	7.83	12.30	-0.58
g721dec	-0.96	13.88	16.47	-0.13
compress	9.22	28.37	35.30	1.90
sha	7.48	20.83	41.38	-1.56
yacc	5.36	21.32	22.53	-0.27
cjpeg	4.42	31.84	39.24	13.36
djpeg	6.51	16.06	28.82	42.53
gsmenc	9.01	12.48	39.10	5.32
gsmdec	28.94	40.96	43.24	0.09
unepic	10.22	19.81	28.41	9.52
mpeg2dec	11.90	23.78	29.23	0.35
average	10.58	26.38	33.83	5.03

Table 1: Effect of increasing the number of register windows for the 4 and 16-register WIMS (left) and VLIW (right) machines. For each table, columns 2, 3, 4 show the performance improvement of w2.r4, w4.r4 and w8.r4 over w1.r4 and column 5 shows the performance improvement of w2.r16 over w1.r16.

Benchmark	**fir**	**rawc**	**rawd**	**g721enc**	**g721dec**	**compress**	**sha**
w2.r8	2.70	2.73	3.17	2.14	2.73	2.54	1.93
w4.r8	2.70	2.73	3.17	2.04	2.89	3.26	4.01
w2.r16	0.00	0.00	0.00	0.11	0.06	0.06	0.00
Benchmark	**yacc**	**cjpeg**	**djpeg**	**gsmenc**	**gsmdec**	**unepic**	**mpeg2dec**
w2.r8	0.00	2.73	6.63	2.33	5.20	2.75	1.47
w4.r8	3.40	3.02	6.03	3.11	3.40	2.86	1.49
w2.r16	0.26	0.23	0.96	0.29	0.00	0.62	0.05

Table 2: Percent savings in execution cycles when inter-window move and window swap operations are combined on the WIMS processor.

ing optimization has on the overall execution cycles for the WIMS processor. Although the benefit is generally small, it does matter more for configurations with a small number of registers and large number of windows. Also, for the WIMS processor, where the swap consumes a single cycle, this helps in reducing the overhead due to the swap instruction. We also studied the effect of combining a conditional branch with a swap, but found the frequency of their occurrence to be less than 1%.

Table 3 shows the energy breakdown and execution times for different instruction types for the WIMS processor. The energy measurements were obtained from Synopsys' Nanosim using post-APR back-annotated parasitics. Input vectors were created at 1.8V and 100MHz operation by running assembled test cases through the pipeline and capturing the switching activity [27]. The power due to different register file sizes was negligible ($< 5\%$) when compared to the pipeline and memory power as the number of registers considered was no more than 32.

The graph in Figure 11 shows the improvement in total dynamic power on the WIMS processor computed using the above table. The total power includes the pipeline and memory power (instruction fetch, loads and stores). Unlike performance, since spills dissipate power to access memory, spill reduction can result in significant improvement in power consumed. For example, gsmdec achieves a 19% reduction in power in the 4-windows of 8-registers per window case. Here, power reduction is obtained by exchanging spill for a swap/move. Unlike performance, where a significant reduction in spill is required to offset the overhead due to moves and swaps, equal exchange is good for power.

To compare the traditional uses of a register window to reduce register save/restore overhead at procedure calls, we did a study where each procedure used a separate window. An infinite supply of windows was assumed, thus eliminating all caller/callee save/restore overhead. But this resulted in less than 2% improvement in performance. As described earlier, the procedure calling overhead in most embedded applications is not high. Thus, there is little opportunity for performance improvement using a register window in a traditional manner.

5. RELATED WORK

Hardware and software schemes have been proposed in prior work to increase the effective number of registers. On the hardware side, windowed architectures similar to ours have been used in a number of processors, including the ADSP-219x [2] and Tensilica's Xtensa [25]. These processors use register windows to reduce procedure call and context switch overhead while handling real-time critical interrupts. The SPARC architecture [24] uses a novel register window scheme to avoid procedure call overheads. A window consists of three groups of 8 registers, the *out*, *local*, and *in* registers. At any given time only one window is visible, as determined by the current window pointer. The *in* registers contain incoming parameters, the *local* registers constitute scratch registers, and the *out* registers contain outgoing parameters. The register windows overlap partially. On a procedure call, the *out* registers of the caller are renamed as the *in* registers of the callee. By using extra register windows,

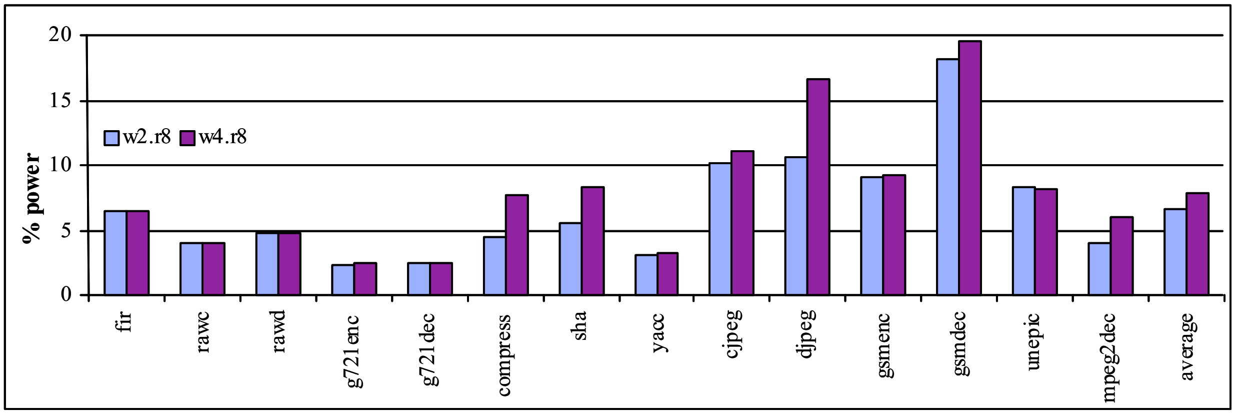


Figure 11: Percent dynamic power improvement of w2.r8 and w4.r8 over the base case of w1.r8.

Instr. class	add-sub	bool	cmp	div	mul	shift	jmp-abs	jmp-rel
Energy (nJ)	0.55	0.38	0.52	2.27	2.22	0.35	0.90	0.64
Time (ns)	10	10	10	180	180	10	30	20
Instr. class	**ld-abs**	**ld-rel**	**st-abs**	**st-rel**	**br-taken**	**br-nottaken**	**win-swap**	**win-move**
Energy (nJ)	0.98	0.74	0.93	0.74	1.00	0.39	0.37	0.47
Time (ns)	20	10	20	10	30	10	10	10

Table 3: Per instruction class energy and execution time for the WIMS processor at 100MHz.

these schemes avoid costly saving and restoring overhead of registers at procedure boundaries.

A similar but more configurable scheme called the Register Stack Engine (RSE) is implemented in the IA-64 architecture [13]. The register stack supports a variable sized window for each procedure, wherein the size is determined by the compiler and communicated to the hardware through special instructions. When the number of physical registers is exceeded, a hardware engine is invoked to save and restore the registers to memory. Again, the RSE is primarily targeted at reducing the save/restore overhead incurred by procedure calls. This work is different in that the focus is on the use of multiple register windows within a single procedure to reduce spill code. In embedded applications, procedure call overhead is generally small as most of the computation is performed within loop nests contained in a single procedure. However, spill code within a single procedure can be high due to small numbers of architected registers.

Register connection [28] and register queues [19, 23] have been proposed to increase the effective number of physical registers without changing the number of architectural registers using hardware/compiler support. Register connection uses special instructions to dynamically connect the core architectural registers to a larger set of physical registers. With register queues, each register is connected to a queue of registers that are effective at maintaining values across multiple loop iterations in software pipelined loops [19, 23]. Both techniques introduce a layer of indirection to access every register operand. Further, additional hardware structures are used in their implementation to maintain the mapping between architected registers and physical registers. These techniques are generally targeted at high-performance platforms as their cost/power overhead are too large for embedded processors.

The register file can also be reorganized to deal with the problems of large register file sizes. Register caches [5] allow low latency register access while supporting a large architectural register file by caching a subset of the values of the register file in a smaller but faster register cache. The function units source their operands from the register cache. Clustering breaks up a centralized register file into several smaller register files, thereby creating a decentralized architecture [7, 8]. Each of the smaller register files supplies operands to a subset of the function units, and can be designed more efficiently. However, these techniques are used to reduce register file access time, porting, and interconnect complexity. They do not deal with the problem of limited encoding space and thus focus on orthogonal problems.

On the software side, code generation for DSP processors has proved to be a challenge for compilers [20]. Irregularities of such architectures has motivated the use of new compiler techniques which were initially considered to be complex and time consuming. Graph partitioning is one such approach. Operation partitioning has been used in compilers for multi-clustered VLIW processors [6, 1, 4]. Several graph partitioning based tools like Chaco [10] and Metis [14] have been widely used to implement mutli-level Kernighan-Lin and other more sophisticated algorithms. These tools assign static weights to nodes and edges while our problem requires dynamic assignment of partition weights. A global register partitioning and interference graph-based approach has been used in the context of multi-cluster and multi-register file processors [11, 3]. Graph partitioning-based approach has been explored in the context of partitioning program variables into multiple memory banks [18]. Our approach, on the other hand, tries to partition virtual registers into multiple register windows within a given procedure scope while trying to minimize spill code, inter-window moves and window swaps.

6. CONCLUSION

In this work, we developed and implemented a novel graph partitioning based compiler algorithm to evaluate the benefits of a windowed register file design. Such a design increases the effective number of available registers without increasing code size. The graph partitioning algorithm partitions the virtual registers in a procedure into multiple register windows, thus reducing the overall spill code while minimizing the overhead due to inter-window moves and window swaps. We evaluated our design over a wide range of machine and window configurations and achieved an average performance improvement of 15% for the 4-register, 10% for the 8-register and 2% for the 16-register case on the WIMS processor, On the VLIW machine, we got an average performance improvement of 26%, 20%, and 5% for the 4, 8, and 16 register cases respectively. The average power reduction was 7% for the 8-register case on the WIMS microcontroller.

7. ACKNOWLEDGEMENTS

We thank Michael Chu, Nathan Clark, and K.V. Manjunath for their comments and suggestions. Fabrication of this work at TSMC was supported by the MOSIS Educational Program. Digital cell libraries and SRAMs were supplied by Artisan Components, Inc. This work was supported primarily by the Engineering Research Centers Program of the National Science Foundation under award number EEC-9986866. Last, we thank Intel Corporation for their generous equipment donation.

8. REFERENCES

[1] A. Aletà et al. Exploiting pseudo-schedules to guide data dependence graph partitioning. In *PACT 2002*, Sept. 2002.

[2] Analog Devices. *ADSP-219x/2191 DSP Hardware Reference Manual*, Jul. 2001. http://www.analog.com/Analog_Root/static/library/dspManuals/ADSP-2191_hardware_reference.html.

[3] J. Cho, Y. Paek, and D. Whalley. Register and Memory Assignment for Non-orthogonal Architectures via Graph Coloring and MST Algorithms. In *LCTES/SCOPES 2002*, Jun. 2002.

[4] M. Chu, K. Fan, and S. Mahlke. Region-based Hierarchical Operation Partitioning for Multicluster Processors. In *PLDI '03*, Jun. 2003.

[5] J.-L. Cruz et al. Multiple-banked register file architecture. In *ISCA-27*, Jun. 2000.

[6] G. Desoli. Instruction assignment for clustered VLIW DSP compilers: A new approach. Technical Report HPL-98-13, Hewlett-Packard Laboratories, Feb. 1998.

[7] P. Faraboschi, G. Desoli, and J. Fisher. Clustered instruction-level parallel processors. Technical Report HPL-98-204, Hewlett-Packard Laboratories, Dec. 1998.

[8] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Micro-30*, Dec. 1997.

[9] M. R. Guthaus et al. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Dec. 2001.

[10] B. Hendrickson and R. Leland. *The Chaco User's Guide*. Sandia National Laboratories, Jul. 1995.

[11] J. Hiser, S. Carr, and P. Sweany. Global register partitioning. In *PACT 2000*, Oct. 2000.

[12] W. M. Hwu et al. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *Journal of Supercomputing*, May 1993.

[13] Intel Corporation, Santa Clara, CA. *Intel IA-64 Software Developer's Manual*, 2002.

[14] G. Karypis and V. Kumar. *Metis: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparse Matrices*. University of Minnesota, Sept. 1998.

[15] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, Feb. 1970.

[16] H. Kim. *Region-Based register allocation for EPIC Architectures*. PhD thesis, Department of Computer Science, New York University, 2001. www.crest.gatech.edu/publications/hansooth.pdf.

[17] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Micro-30*, Dec. 1997.

[18] R. Leupers and D. Kotte. Variable Partitioning for Dual Memory Bank DSPs. In *ICASSP 2001*, May 2001.

[19] J. L. Marcio M. Fernandes and N. Topham. Allocating Lifetimes to Queues in Software Pipelined Architectures. In *Euro-Par '97*, Aug. 1997.

[20] P. Marwedel and G. Goossens. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, Boston, MA, 1995.

[21] Motorola. *CPU12 Reference Manual*, Jun. 2003. http://e-www.motorola.com/brdata/PDFDB/docs/CPU12RM.pdf.

[22] R. M. Senger et al. A 16-Bit Mixed-Signal Microsystem with Integrated CMOS-MEMS Clock Reference. In *DAC '03*, Jun. 2003.

[23] M. Smelyanskiy, G. Tyson, and E. Davidson. Register Queues: A New Hardware/Software approach to Efficient Software Pipelining. In *PACT 2001*, Oct. 2001.

[24] SPARC International Inc. *The SPARC Architecture Manual, Version 8*, 1992. www.sparc.com/standards/V8.pdf.

[25] Tensilica Inc. *Xtensa Architecture and Performance*, Sep. 2002. http://www.tensilica.com/xtensa_arch_white_paper.pdf.

[26] Texas Instruments. *TMS320C54X DSP Reference Set*, Mar. 2001. http://www-s.ti.com/sc/psheets/spru131g/spru131g.pdf.

[27] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, 1994.

[28] T.Kiyohara et al. Register Connection: A New Approach to adding Registers into Instruction Set Architectures. In *ISCA-20*, May 1993.

[29] Trimaran. An infrastructure for research in ILP. http://www.trimaran.org.