Partitioning Variables across Register Windows to Reduce Spill Code in a Low-Power Processor

Rajiv A. Ravindran, *Student Member*, *IEEE*, Robert M. Senger, Eric D. Marsman, Ganesh S. Dasika, *Student Member*, *IEEE*, Matthew R. Guthaus, Scott A. Mahlke, *Member*, *IEEE*, and Richard B. Brown, *Senior Member*, *IEEE*

Abstract—Low-power embedded processors utilize compact instruction encodings to achieve small code size. Such encodings place tight restrictions on the number of bits available to encode operand specifiers and, thus, on the number of architected registers. As a result, 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. This design gives the appearance of a large register file without compromising the instruction encoding. To support the windowed register file, we designed and implemented a graph partitioning-based compiler algorithm that partitions program variables and temporaries referenced within a procedure across multiple windows. On a 16-bit embedded processor, an average of 11 percent improvement in application performance and 25 percent reduction in system power was achieved as an 8-register design was scaled from one to two windows.

Index Terms—Code generation, embedded processor, graph partitioning, instruction encoding, low-power design, optimization, retargetable compilers, register window, spill code.

1 Introduction

In the embedded processing domain, power consumption is one of the dominant design concerns. 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 narrow bitwidth instruction and data designs (e.g., 8 or 16 bits), such as the Motorola-68HC12 [25]. Tight instruction encodings offer the advantage of compact code and, thus, smaller instruction memory requirements. Further, embedded applications such as sensor signal processing commonly operate on narrow precision data and, hence, are perfectly suited for such processors. Thus, these processors provide more efficient designs with datapaths optimized for narrow precision data.

While the efficiency of narrow bitwidth 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 narrow bitwidth processors, the architected registers. An instruction-set with limited encoding (8 or 16 bits) significantly reduces instruction fetch power by reducing the code footprint. But, reduced encoding limits the bits available to specify source and destination operand specifiers, thus restricting the number of architected registers to a small number (e.g., eight or less). For example, TMS320C54x [34] has eight address registers and ADSP-219x [2] has 16 data registers. Similarly, Thumb mode in ARM [28] uses a 16-bit instruction encoding with eight addressable registers. Restricting the number of addressable 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 [29]. 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 utilized to activate and move data between windows. The goal is to provide the appearance of a large monolithic register file by judiciously employing the register window.

Traditionally, register windows have been used to reduce the register save and restore overhead at procedure calls, such as in the SPARC architecture [31]. A similar but

- *R.A. Ravindran, R.M. Senger, E.D. Marsman, G.S. Dasika, M.R. Guthaus, and S.A. Mahlke are with the Department of Electrical Engineering and Computer Science, University of Michigan, 1301 Beal Ave., Ann Arbor, MI 48109-2122. E-mail: {rravindr, rsenger, emarsman, gdasika, mguthaus, mahlke}@umich.edu.*
- *R.B. Brown is with the College of Engineering, University of Utah, 1495 East 100 South, 214 KENNB, Salt Lake City, UT 84112. E-mail: brown@coe.utah.edu.*

Manuscript received 19 Apr. 2004; revised 21 Feb. 2005; accepted 6 Apr. 2005; published online 15 June 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-0137-0404.

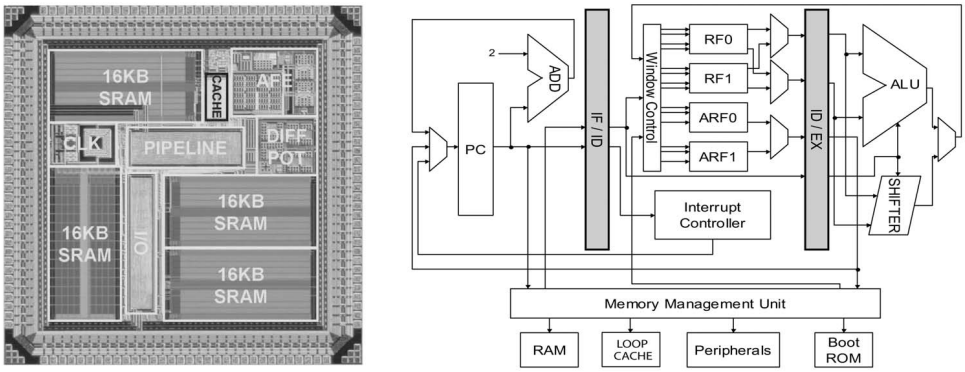


Fig. 1. The WIMS microcontroller in TSMC 0.18μm CMOS and the WIMS datapath.

more configurable scheme, called the Register Stack Engine (RSE), is implemented in the IA-64 architecture [15]. 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. Windowing techniques have also been employed in embedded microprocessors, including the ADSP-219x [2] and Tensilica's Xtensa [33]. These processors typically use register windows to reduce context switch overhead while handling real-time critical interrupts.

Our studies have shown that, for the more loop-dominated applications found in the embedded domain, the use of register windows to reduce procedure call overhead has limited impact on performance. We did a study where each procedure used a separate window with 8-registers per window. An infinite supply of windows was assumed, thus eliminating all caller/callee save/restore overhead. This resulted in less than 2 percent improvement in performance. The central problem is that a majority of embedded applications spend most of their time in loop nests contained within a single procedure [11]. 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 reduce spill code. Eliminating spill loads and stores reduces memory accesses and, thus, improves performance and power consumption.

To support intraprocedural window assignment, the compiler employs a 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 swaps) and moving data between windows (also referred to as interwindow 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. This paper is an extension of our earlier work [27].

2 Windowed Architecture

2.1 WIMS Microcontroller Overview

The WIMS Microcontroller was designed to control a variety of miniature, low-power embedded sensor systems [29]. The microcontroller, fabricated in TSMC 0.18μm CMOS, is shown in Fig. 1 and consists of three major subblocks: the digital core, the analog front-end (AFE), and the CMOS-MEMS clock reference. Power minimization was a key design constraint for each subblock.

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 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 has four 16KB banks of on-chip SRAM with a memory management unit that disables inactive banks of memory.

A 16-bit WIMS instruction set was custom designed and includes 77 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.

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

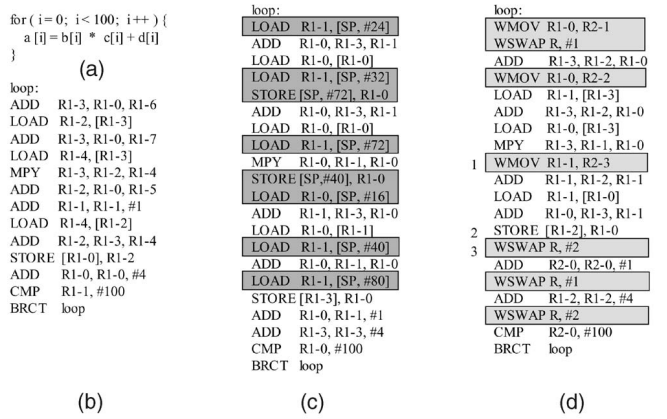


Fig. 2. Register window example. (a) C-source. Assembly code for (b) 1-window of 8-registers, (c) 1-window of 4-registers, and (d) 2-windows of 4-registers. Registers are denoted by window number "-" the allocated register number.

in 16 bits by reducing the number of bits required to encode the 16 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 nonwindowed 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 (WSWAP) switches register windows in a single cycle by changing the MSR window bit setting. Three additional nonwindowed address registers (a stack pointer, frame pointer, and link register) are provided for subroutine 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 Fig. 2. The original C-source is shown in Fig. 2a. The loop segment has been mapped to three different register window configurations: Fig. 2b shows 1-window of 8-registers, Fig. 2c shows 1-window of 4-registers, and Fig. 2d shows 2-windows of 4-registers. For clarity, we use a generic RISC-like instruction set instead of the WIMS instruction set and assume a unified register file instead of disjoint address and data files throughout all examples in this paper. In the assembly code in Fig. 2, the leftmost operand is the destination. We use the notation *Ri-j*, where *i* denotes the window number and *j* denotes the register number. For the window swap operation (*WSWAP*), the first operand specifies the register file, while the second argument specifies the new active window.

The windowed register file architecture restricts all operands within a single instruction to refer to the current active window. All operations following a *WSWAP* access their operands from the new active window. *WMOV* denotes the interwindow move instruction which can move values between any two register windows. If an operation refers to registers in different windows, one or more *WMOV* operations are required. Considering Fig. 2d, the *WMOV* instruction (instruction marked 1) transfers the value from register *R2-3* to the register *R1-1*, which is then used in the following *ADD* instruction. The *STORE* instruction (instruction marked 2) accesses all of its operands from window 1. The *WSWAP* (instruction marked 3) toggles the active window from 1 to 2 so that the following *ADD* instruction can source all of its operands from window 2.

In Fig. 2b, 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 (the load and store instructions shaded in dark gray) is generated as there are insufficient registers to hold the necessary values, as shown in Fig. 2c. In Fig. 2d, 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-interwindow moves (all shown in light gray). This configuration has the same number of total registers as that in Fig. 2b with the encoding benefits of Fig. 2c. On the WIMS processor, where every instruction executes in a single cycle, Fig. 2c has an 8-cycle overhead as compared to Fig. 2b, while Fig. 2d has only seven extra instructions. More importantly, Fig. 2d has fewer loads and stores (zero spill operations) to memory as compared to Fig. 2c (eight spill operations) and thus consumes significantly less memory power.

The remainder of the paper explains the compiler algorithm to automatically partition the variables and temporaries referenced in a procedure into the available register windows, as illustrated in Fig. 2d, such that the spill cost is reduced while minimizing the extra overhead due to the window swaps and interwindow moves.

3 Register Window Partitioning

3.1 Overview

The overall compilation system for register window partitioning is based on the Trimaran infrastructure [37] and is shown in Fig. 3. Ignoring the gray boxes, the base compiler system consists of the machine independent front end, which does profiling, classical code optimizations (such as common subexpression elimination, constant folding, induction variable elimination, etc.), loop unrolling, and procedure inlining to produce a generic assembly code for a load-store architecture. 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, connectivity of register files to function 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 of the target machine. After prepass scheduling, all VRs are partitioned into the available register windows. For each register file, the register allocator uses a graph coloring algorithm [19] to assign physical registers to the VRs, generating spills if required. Finally, the

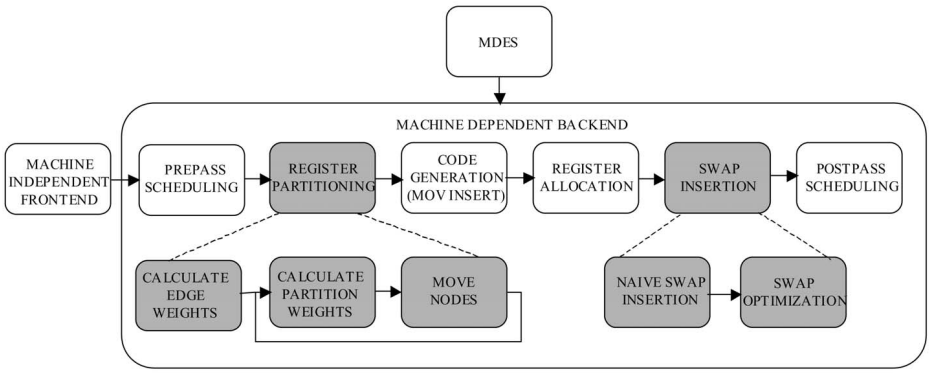


Fig. 3. Overview of the compiler system.

resultant code is postpass scheduled to produce the fully bound assembly code.

The new phases added to handle register window partitioning are shown in gray boxes in Fig. 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 an operation to different windows. The code generator inserts appropriate interwindow moves to honor the architectural constraints of all registers accessed by a single operation coming from the same window. 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. Prior to postpass scheduling, an edge drawing phase inserts additional dependence edges to ensure that operations do not move across window swaps.

The register window partitioning algorithm is modeled as a graph partitioning problem where the nodes in the graph correspond to VRs used in the assembly code and the edges represent the affinity between VRs. The goal is to partition the VRs into different register windows/partitions so as to minimize the overall spill, interwindow moves, and window swaps, which indirectly leads to our overall goal of performance/power improvement by reducing the number of memory accesses for data operands.

Partitioning consists of two distinct phases—weight calculation and node assignment. Each partition is assigned a weight that measures the cost of spilling the VRs assigned to that partition. The affinity between VRs is captured using edge weights, which represents the penalty incurred if two nodes connected by the edge are assigned to different partitions. The penalty can be an interwindow move, window swap, or both. If two VRs referenced within a single operation are assigned to different partitions, the code generator is forced to insert an interwindow move. Similarly, if two VRs in different operations are not assigned to the same window, 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.

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 interpartition edge weights. The register partitioning algorithm uses a modified version of the Fiduccia-Mattheyses graph partitioning algorithm [10], which is an extension of the Kernighan-Lin algorithm [18]. 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.

Fig. 4a is a code segment from the innermost loop of the finite impulse response (FIR) filter. The dynamic execution frequency, obtained from profiling the application on a sample input, is 3,104. This example will be used throughout this section to illustrate the weight calculation and node assignment process. The goal here is to partition this region into 2-windows of 4-registers each, although the WIMS processor has 2-windows of 8-registers per window. 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 Edge Weight Calculation

An edge is associated with every pair of VRs. The edge weight is used by the partitioning algorithm to represent the degree of affinity between two VRs. 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 when two VRs are placed in different windows. By placing two VRs with high affinity in a single partition, the algorithm reduces the number of swaps and moves. The edge weight between VRs is expressed as a matrix (see Fig. 4b) 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 interwindow move (*WMOV*) operation. For every pair of VRs, the number of operations weighted by frequency that reference both registers as operands is the estimated cost

1. A region is any block of code considered as a unit for scheduling like a basic block or superblock [14].

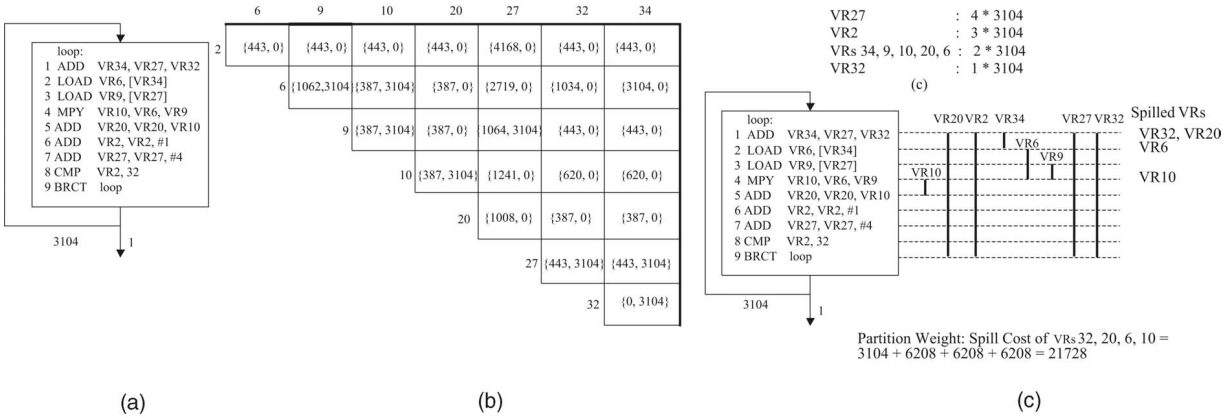


Fig. 4. Example of partition and edge weight calculations. (a) Example loop. (b) Edge weight matrix: Each cell contains {swap cost, move cost}. (c) Spill cost of VRs. (d) Partition weight computation assuming all VRs assigned to one partition.

move. In Fig. 4a, VRs 6 and 9 are referenced in operation 4. If these VRs are in different windows, a *WMOV* is required for this operation. Hence, the move cost for VRs 6 and 9 is the frequency of operation 4 or $3,104$. Conversely, VRs 27 and 6 are not referenced together in any operation and, hence, do not require a move. This process is carried out for all pairs of VRs, producing the matrix of values in Fig. 4b (right entry in each cell).

Swap Cost. If two VRs are assigned to different windows/partitions, a window swap (*WSWAP*) 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 for consecutive references to the same VR. At the end of the region, the total number of swaps gives an estimate of the number of swaps required between this pair of VRs. The swap cost is therefore the number of swaps times the profile weight of the region under consideration.

In Fig. 4a, between VRs 27 and 34, the swaps are computed as follows: We assume that VRs 27 and 34 are assigned to windows 1 and 2, respectively. VR27 is referenced in operations 1, 3, and 7 and, so, these operations are assigned to window 1. VR 34 is referenced in operation 2 and, so, the operation is assigned to window 2. In the sequential execution, swaps need to be inserted after operations 1 (window 2 activated) and 2 (window 1 activated). Hence, the total swap cost is $2 * 3,104 = 6,208$.

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 Fig. 4a. 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 9-10, 9-6, 27-10, and 27-6, although only a single swap is necessary.

To deal with overcounting, swap counts are used 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, five swaps are required. These swaps are due to VR pairs 10-27, 20-27, 2-27, 27-9, and 27-6. Generalizing this, let $c_1, c_2 \ldots c_k$ be the swap count due to swaps required by all pairs of VRs after operations $op_1, op_2 \ldots op_k$. 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

$$(1/c_1 + 1/c_2 + \ldots + 1/c_k) * cost\_of\_swap,$$

where $cost\_of\_swap$ is the cost of a single swap operation. Intuitively, a swap after an operation could be shared by multiple VR pairs. Further, regardless of the number of windows, at most one swap is required between every pair of operations. Thus, 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 Fig. 4a, VRs 10 and 27 require a swap after operations 3 and 6. Since operation 3 has a swap count of 5 (due to the five 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 $(1/5 + 1/5) * 3,104 = 1,241$ (Fig. 4b, left entry in each cell).

3.3 Partition Weight Calculation

The partition weight estimates the spill cost for the VRs assigned to each partition. The node assignment phase tries to minimize the sum of the weights of all the partitions. The partition weights are computed using a crude linear scan register allocation algorithm [26] to compute the estimated dynamic spill cost.

Given a set of VRs assigned to a partition, the live-ranges (the range of operations from all defines to all uses of the value) of the VRs are computed. For each VR, its dynamic reference count is calculated using the profile information. If the VR is spilled, then the dynamic reference count gives an estimate of the load/store overhead for spilling that VR. 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 estimating spills. The actual spill code insertion is done during register allocation within each window after the window assignment process.

The VRs are chosen for spilling in increasing order of dynamic reference count. If two VRs have the same dynamic reference count value, the one with the 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.

In Fig. 4c, the spill cost/dynamic reference count for each VR is shown, while, in Fig. 4d, the live-ranges of the VRs are shown on the right. Assume that all VRs are assigned to a single partition and three physical registers are available per partition. At operation 1, five VRs (20, 2, 34, 27, and 32) are live simultaneously. Since there are only three registers available in the partition, VRs 32 and 20 are spilled. VR 32 has a spill cost of 3,104 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 3,104. Hence, VR 32 is picked first. VRs 20 and 34 both have a dynamic reference count of 6,208, but VR 20 has a larger live-range and is chosen next for spilling. At operation 2, VRs 20, 2, 34, 6, 27, and 32 are live. Since 32 and 20 are already spilled, only VR 6 gets spilled as it has a smaller dynamic reference count than VRs 2 and 27 and larger live-range than VR 34. At operation 3, VRs 20, 2, 6, 9, 27, and 32 are live. Since 32, 20, and 6 are already spilled, no more VRs are spilled as the number of remaining live VRs is equal to three. VR 10 is spilled at operation 4. For the rest of the operations, no additional VRs are spilled. So, for this partition, the partition weight is the spill cost of the spilled VRs 32, 20, 6, 10, which is $3,104 + 6,208 + 6,208 + 6,208 = 21,728$. In actual implementation, instead of considering every operation, only operations which are at the start/end points of any live-range are considered. So, in Fig. 4d, only operations 1, 2, 3, 4, 5, and 9 are considered. For the other operations, the live-range information does not change and, hence, is ignored.

Since region-based partitioning is performed, window assignments of a higher priority region can affect the decisions in a lower priority region. While computing the partition weights, it is possible that there are live VRs that are already assigned to partitions from processing higher priority regions. If these VRs were not spilled, then they are assumed to be prebound to a register. Thus, the window has one fewer register available per such prebound register.

3.4 Node Partitioning

The goal of the node partitioning phase is to reduce the overall spill cost while minimizing the impact due to interwindow moves and swaps. Starting from an initial partition, the node partitioning algorithm tries to iteratively distribute the 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. The node partitioning technique that we used is a modified version of Fiduccia-Mattheyses's [10] graph partitioning algorithm (FM). It consists of two phases —*initial partitioning* and *iterative refinement*.

```
1) Sort all VRs in decreasing order of
   dynamic reference count (sorted_list)
2) For every partition p {
3)  For every vr in the sorted_list {
4)   LR = list of ops in which vr is live
5)   spill = false;
6)   For every op in LR {
7)    alloc_vrs =
        list of vrs at op that are
        live and allocated to partition p
8)    num_avail_regs =
        num_registers(p) - sizeof(alloc_vrs)
9)    if (num_avail_regs <= 0) {
10)    spill = true;
11)    break;
12)   }
13)  }
14)  if (!spill) {
15)   add vr to partition p
16)   mark vr as allocated to partition p
17)   remove vr from sorted_list
18)  }
19) }
20) if (!sorted_list.is_empty())
21)  add all VRs in sorted_list
     to first partition p
```

Fig. 5. Algorithm for initial node partitioning.

Initial Partitioning. Placing all VRs in the first partition can create an unbalanced initial configuration, which can affect the quality of the partitioning algorithm. The initial partitioning algorithm tries to distribute the VRs into partitions so as to start with an initial configuration of relatively less register pressure while being incognizant of the swap and move overhead. If a given window/partition has sufficient registers to accommodate the VRs, then all of the VRs are allocated to that partition. If not, the VRs are assigned based on a priority order to a particular window/partition until no more VRs can be assigned to it without the need for spilling. The remaining VRs are then placed in the next partition until it gets saturated and so on.

The algorithm for the initial distribution of VRs to windows/partitions is given in Fig. 5. Initially, all VRs are sorted based on the dynamic reference count (Step 1) so that the most important VRs are assigned first. For a given partition, allocation is attempted for every VR in the sorted list. The allocation is done using a simple linear scan register allocation algorithm similar to the technique described in Section 3.3. It should again be noted that this heuristic generates an initial partition of VRs to register windows without actually register allocating them. For every VR, its live-range (*LR*) is computed (Step 4) as a list of operations. For every operation *op* in *LR* (Step 6), *num_avail_regs*, which is the difference between the total number of registers in the current partition and number of allocated registers that are live at *op*, is computed (Steps 7 and 8). If there are free registers (Step 9), then this VR is assigned to the current partition (Steps 14, 15, and 16), else it

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 interwindow moves that are inserted later.

```
1)  pset = create_n_parts();
2)  initialize_parts(pset);
3)  min_overall_wt = comp_overall_wt(pset);
4)  save_partition_info();
5)  saved_info = false;
6)  while (min_overall_wt > 0) {
7)   src_part = find_unbalanced_partition(pset)
8)   if (src_part == NULL) {
9)    if (saved_info) {
10)    restore_part_info();
11)    saved_info = false;
12)   } else
13)    break; // terminate partitioning
14)   unlock all locked vrs in pset
15)  }
16)  <vr,dest_part> = find_best_vr(src_part)
17)  if (dest_part == 0) {
18)   remove src_part from pset
19)   continue;
20)  }
21)  do_move_vr(vr, dest_part)
22)  lock_vr(vr, dest_part)
23)  overall_wt = comp_overall_part_wt(pset);
24)  if (overall_wt < min_overall_wt) {
25)   min_overall_wt = overall_wt;
26)   save_partition_info();
27)   saved_info = true;
28)  }
29) }
30) restore_part_info();
```

Fig. 6. Algorithm for node partitioning.

is assumed to be spilled. This process continues for all VRs in the sorted list such that they are either assigned to the current partition or spilled. The spilled VRs are then attempted for assignment to the next partition (Step 2) using the same algorithm. Once all of the partitions are processed, any remaining spilled VRs that could not be assigned to any partition are simply assigned to the first partition (Step 21).

Iterative Refinement: After the initial assignment, the iterative refinement phase tries to move VRs across partitions to reduce the overall spill cost (partition weights) while minimizing the overhead due to swaps and moves (edge weights between partitions). The algorithm for the node partitioning is given in Fig. 6. Initially, a set of n-partitions (where n is the number of register windows) is created (*pset*, Step 1) and initialized using the initial partitioning algorithm described in Fig. 5 (Step 2). This partition configuration is used as the seed configuration for the first pass. In Step 3, the overall weight of the set of partitions is computed. The overall weight is the sum of the partition weights and the weights of the cut edges across all partitions. The initial partition configuration is then saved in *save_partition_info* in Step 4, assuming that it is the best seen yet. During a single iteration of the loop in Step 6, the partition (*src_part*) with the maximum weight is selected (*find_unbalanced_partition*, Step 7). If such a partition exists (Step 16), the node with the largest gain (*find_best_vr*) is selected.

Fig. 7 gives the algorithm 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. 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

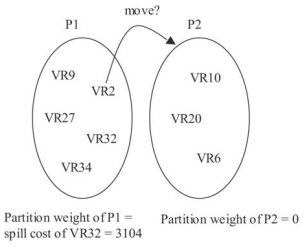
```
1) find_best_vr(src_part) {
3)  for every node in src partition {
4)   for every dest_part in pset {
5)    move_node(node, dest_part)
6)    old_total_wt =
       srcp_wt_old+destp_wt_old
7)    new_total_wt =
       srcp_wt_new+destp_wt_new
8)    partition_wt_gain =
       old_total_wt-new_total_wt
9)    edge_wt_gain =
       old_edge_wt-new_edge_wt
10)   gain =
       partition_wt_gain+edge_wt_gain
11)   if (gain > maxgain) {
12)    bestnode = node
13)    maxgain = gain
14)   }
15)  }
16) }
17) return (bestnode,dest_part)
18)}
```

Fig. 7. Algorithm to find best VR.

to the destination partition. Similarly, *edge_wt_gain* is the reduction in edge weights between nodes in the source and destination partitions. In Fig. 7, *srcp_wt_old/destp_wt_old* is the weight of the source/destination partition before the node is moved, while *srcp_wt_new/destp_wt_new* is the weight of the source/destination partition after the node is moved. The node (*bestnode*) with the highest gain and the destination partition (*dest_part*) to which it is to be moved are returned.

The partitioning algorithm then picks the node with the highest gain (*vr*) and performs the move to the destination partition (*dest_part*, Step 21). It should be noted that the highest gain could be a negative value. Allowing negative gains helps avoid local minima. Once a node is moved over to the new partition, it is locked in the new partition and not considered in the current pass (Step 22). The overall partition weight is then recomputed in Step 23. If the overall weight is less than the minimum overall weight, it implies that the resultant partition configuration is the best configuration seen so far and, hence, the configuration is saved. If *find_best_vr* has no more VRs to move (either because all VRs have been locked or there are no destination partitions to move to), the *src_part* is removed from the set of partitions (Steps 17 and 18). In Step 8, if there are no more partitions left, the current pass is ended. If, during the previous pass, a better configuration was seen and saved (*saved_info* flag is set to true), then the best configuration seen so far is restored and used as the seed for the next pass (Step 10). Before commencing the next pass, all VRs are unlocked (Step 14). If *saved_info* is set to false, it implies that, during the previous pass, a better configuration was not seen and the partitioning is terminated. The partitioning is also terminated if the *min_overall_wt* reaches 0 (Step 6). On exit, the best partition seen over all passes is restored as the final partition (Step 30).

Example. The initial partition configuration is shown in Fig. 8a. To compute the initial partitions, the live-ranges and the dynamic reference counts of the VRs are shown in Fig. 4c and Fig. 4d, respectively. The VRs are considered in the decreasing order of dynamic reference

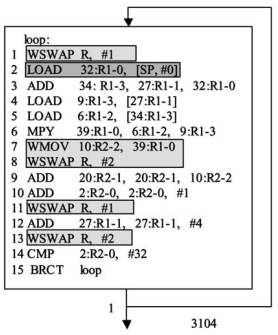


(a)

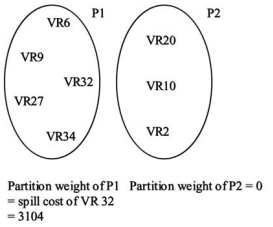
VR	Partition Gain	Edge Gain	Total Gain
2	-3104	-4168	-7272
9	-6208	2547	-3661
27	-3104	-10462	-13566
32	-3104	-5496	-8600
34	0	-3426	-3426

(b)

Fig. 8. Partitioning applied to example. (a) Initial partition. (b) Gains for each VR moving from P1 to P2 after the initial partition.



(a)



(b)

Fig. 9. Example after window assignment. The notation $VR : R_i - j$ is used, where VR is the original virtual register number from Fig. 4a, i is the window number, and j is the allocated register number.

count. We assume 3-registers per window. Initially, the top two VRs, 27 and 2, are allocated to partition 1. Next, VRs 34 and 9 are allocated as they do not interfere with each other and the number of maximum interfering live-ranges is three. Since partition 1 is now saturated, the rest of the VRs (except VR 32) are assigned to partition 2. VR 32, which has the lowest reference count, cannot be assigned to either partition withour spilling, therefore, by convention, it is assigned to partition 1. The initial partitioning algorithm thus distributes the VRs such that the total partition weight is minimized $(3, 104)$.

The iterative refinement phase then tries to move each VR from P1, which is the highest weight partition, to P2 and computes the resultant partition and edge gains. Fig. 8b shows the partition, edge, and total gain for moving each VR. The VR with the maximum total gain is chosen. Here, all gains are negative, so VR 34 with the smallest negative gain is moved to partition 2. This VR is then locked in partition 2. Subsequently, VR 6 is moved to partition 1 and VR 2 is moved to partition 2 to obtain the final partition in Fig. 9b.

Edge weights are computed statically before partitioning. So, *find_best_vr* need only do a lookup of the edge weight matrix (Fig. 4b) 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 3.3) on the fly. Each move of a node would thus require an $O(n)$ scan of the operations in the region and, hence, the complexity of the partitioning algorithm is $O(n^2)$ per round of refinement. Fig. 9 shows the final partition configuration. The total partition weight is $3, 104$, which is the same as the initial partition weight. The initial partitioning algorithm minimized the number of spills, but did not consider the impact of swaps and moves. The iterative refinement converged at a configuration such that both spill cost (partition weights) and swap and move cost (edge weights) are minimized. The final code after register allocation and swap insertion is also shown in Fig. 9. The code has one spill (operation 2), four swaps (operations 1, 8, 11, and 13), and one interwindow move (operation 7).

3.5 Partitioning Algorithm Optimizations

In order to speed up the partitioning process, two optimizations were implemented over the core algorithm described above.

Fast Spill Pressure Estimation. A cycle-by-cycle linear scan of the operations in a region to determine what VRs are spilled is inherently slow as this has to be done every time a VR is moved from a source to a destination partition. This dynamic computation of partition weight was required because the set of VRs that are spilled is a function of the interfering live-ranges of the current assignment of VRs to that partition. Although accurate, since this is done within the core of the FM partitioning algorithm, it slowed down the partitioning process. To optimize this process, instead of scanning every operation, one could only scan the operation with the maximum number of interfering live-ranges to approximate what VRs are spilled. The process is still dynamic, but less accurate than the linear scan approach.

Restricting the Number of VRs. Although the partitioning was performed a region at a time, some of the larger benchmarks had regions with a large number of VRs that slowed down the partitioning algorithm. This large number of VRs was generated mainly because the original application source had loop kernels that were written in an unrolled manner. In order to restrict the number of VRs, a compile-time fixed subset of VRs is partitioned at a time. The VRs in a region are sorted in decreasing order of dynamic reference count. By sorting the VRs, the algorithm can consider the most important subset of VRs for partitioning, followed by the next most important subset, and so on. This is done until all VRs in that region are exhausted. Partitioning decisions for a given subset are honored while partitioning the next subset (similar to the pre-bounds described at the end of Section 3.3).

3.6 Window Swap Insertion and Optimization

Window swap operations are inserted after window assignment and register allocation (see Fig. 3). Initially, a naive window assignment is performed by walking the region in sequential program order. A window swap operation is inserted at the beginning of the region to set the active window appropriately for the first operation in the block. Scanning each operation, if the assigned window is different from the current register window, a swap to the new window is inserted. Following every procedure call, a window swap operation is used to set the current active window to the window of the operation following the

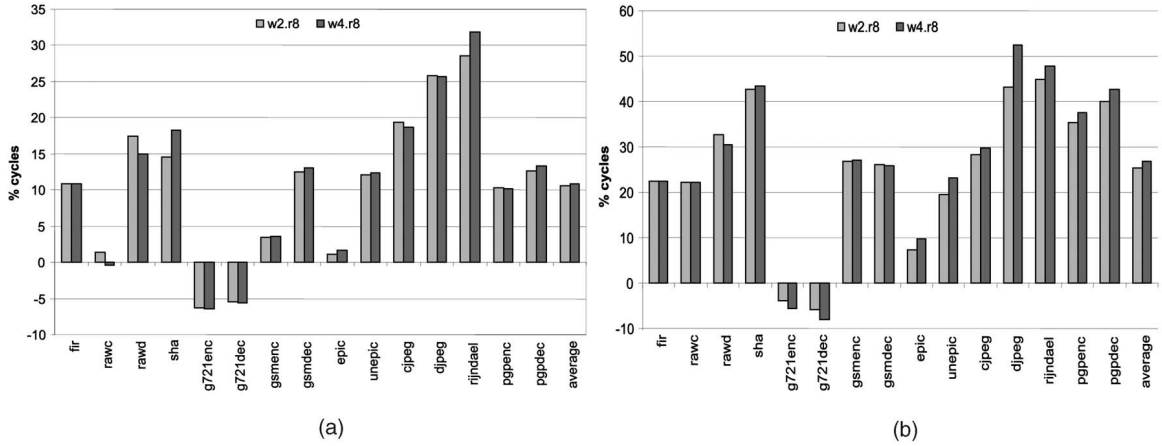


Fig. 10. Performance improvement shown as percent reduction in cycles for the 8-register (a) WIMS processor and (b) VLIW processor for two and four window designs. For each benchmark, the results for w2.r8 and w4.r8 are plotted relative to w1.r8.

procedure call. This is necessary as we assume separate compilation and the state of the active window is unknown after a procedure return.

Swap optimizations. This naive 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 upward 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 that the new window swap instruction inserted at the end of the predecessor is the last instruction 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.

4 Experimental Results

4.1 Methodology

We implemented the register partitioning algorithm using the Trimaran infrastructure, a retargetable compiler for VLIW processors [37]. 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 the Mediabench [22] and MiBench [11] suites was evaluated. All of the benchmarks were compiled with control-flow profiling, superblock formation, function inlining, and loop unrolling. 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: two integer, one floating-point, one memory, and one branch. The VLIW-machine uses the HPL-PD [17] ISA with latencies similar to an Itanium machine with perfect caches and support for compile-time speculation and predication. The VLIW-style architecture was chosen due to its increasing popularity in the embedded domain [35]. 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 instruction slot. Interwindow moves execute on the integer unit.

We considered the power/performance improvement of a range of register file configurations consisting of one, two, four, and eight identical windows containing four and eight registers per window. The following specific 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); and 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). This helped illuminate the power/performance benefits of increasing the effective number of registers without changing the instruction set architecture. We also fixed the total number of registers while partitioning them into two and four equally sized register windows. In particular, the performance of window configurations w2.r8 and w4.r4 was compared against the base w1.r16. This helped clarify the performance degradation suffered by a windowed architecture compared to a nonwindowed architecture having the same number of physical registers. 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.

4.2 Results

Increasing the number of available registers with a fixed window size. The graph in Fig. 10a compares the percent performance improvement in total execution cycles of the w2.r8 and w4.r8 configuration against the base w1.r8

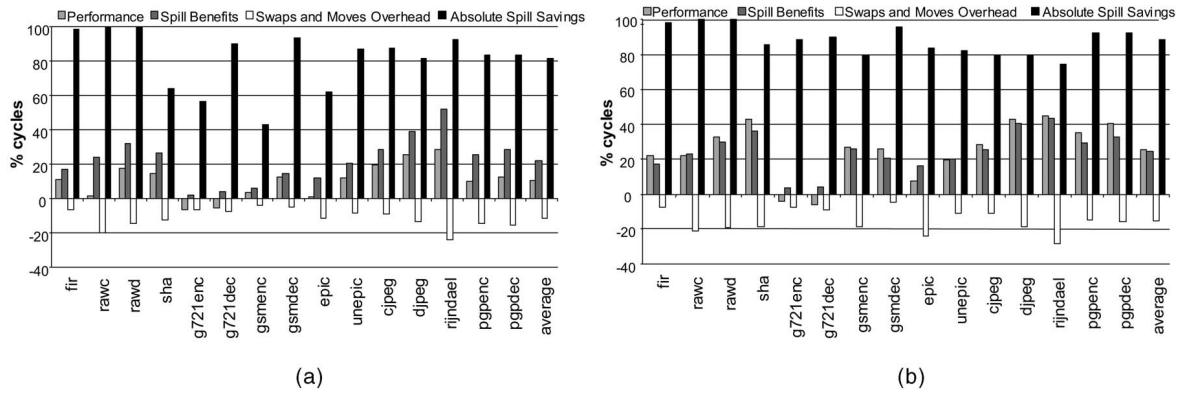


Fig. 11. Performance analysis for the 8-register (a) WIMS processor and (b) VLIW processor. For each benchmark, w2.r8 is plotted relative to w1.r8.

configuration for the WIMS processor. Averages of 11 percent and 12 percent improvement in performance are observed for the 2-window and 4-window designs, respectively. It should be noted that the performance improvement is the result of increasing the effective number of registers while retaining a 3-bit operand encoding. The performance ranges from a maximum of 28 percent for djpeg to a loss of 6 percent for g721enc. We observe only a marginal improvement in performance for a 4-window design. The additional two windows enable significant reduction in spill code, but the resulting advantage is offset by an almost equal increase in interwindow swaps and moves. The partitioning algorithm is able to identify this overhead using the edge weights and, hence, prevents excessive partitioning. But, in benchmarks such as rawd, the heuristic breaks and a net loss in performance is suffered in scaling from two to four windows.

Fig. 10b compares the performance improvement for the w2.r8 and w4.r8 configurations over the base w1.r8 configuration for the VLIW-machine. We observe an average of 25 percent and 28 percent improvement in performance for 2-window and 4-window designs, respectively. This gain 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. 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 more compact schedules and larger performance gains than for a 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, the affects of eliminating spill code 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.

An analysis of the performance for the w2.r8 configuration is presented in Fig. 11a for the WIMS processor. The percent performance improvement in total execution cycles, which is identical to the plot shown in Fig. 10, is shown in the leftmost bar of each set. The rightmost bar shows the percent savings in dynamic spill code. The middle pair of bars show two components—1) spill benefit, which is the percent savings in total execution cycle count due to savings in spill (second bar) and 2) percent swap and move overhead, which is the percent of overall execution cycles due to the extra interwindow moves and window swaps, thereby reducing performance (third bar).

In 11 of the 15 benchmarks, we observe more than 80 percent reduction in spill code. Performance improvement is obtained when the spill benefit exceeds the swap and move overhead. This occurs in 13 of the 15 benchmarks (except g721enc and g721dec). The graph illustrates the competing effects of spill code reduction and swap/move overhead. For example, in gsmdec, a 92 percent reduction in spill code is seen, which accounts for 12 percent savings in total cycles, while, for g721dec, there is a 90 percent reduction in spill code, but this contributes to only 4 percent savings in total cycles. This implies that the impact of spill is small for g721enc in the w1.r8 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 greedy nature of the partitioning algorithm causes VRs to be aggressively separated into different partitions, thus increasing the swap/move cost.

The graph in Fig. 11b illustrates the same performance analysis for the w2.r8 VLIW machine. The spills, swaps, and moves shown in this figure are measured in percent dynamic operations and not dynamic cycles. Observe that, for reasons described earlier, the impact due to savings in spill is greater than in the WIMS processor. Also, the overhead of swaps and moves is lower as they can execute in parallel with other instructions. Hence, performance improvement is often larger than the difference between spill benefit and swap/move overhead.

Fig. 12 compares the performance improvement of 4-register per window configurations on the WIMS (Fig. 12a) and VLIW (Fig. 12b) 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 four registers are insufficient for both the WIMS and the VLIW machines. Djpeg, due to loop unrolling, had a high register pressure and, hence, benefited significantly when the number of windows was increased

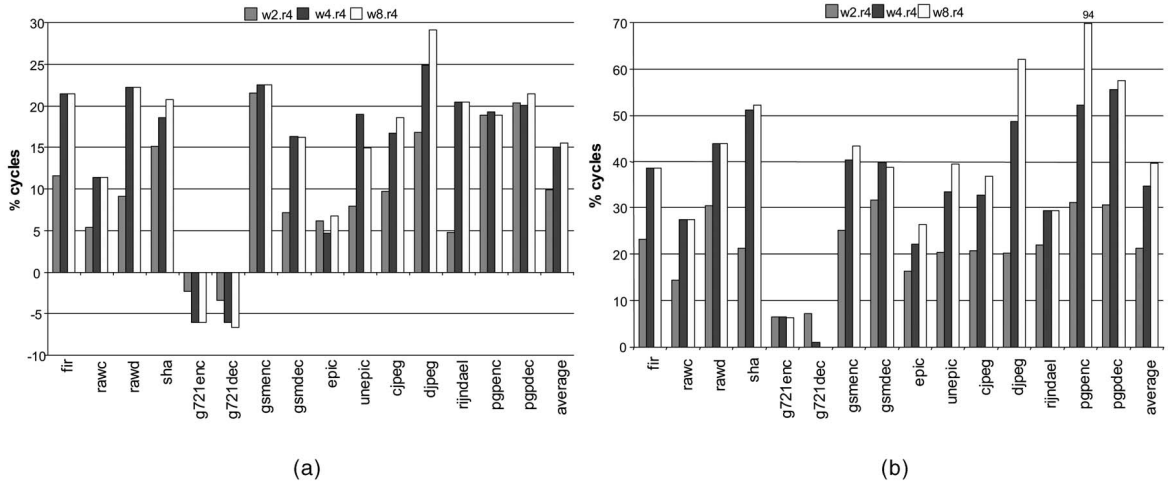


Fig. 12. Performance improvement shown as percent reduction in cycles in increasing the number of register windows in a 4-register (a) WIMS processor and (b) VLIW processor. For each benchmark, three sets of data are shown: w2.r4 (left), w4.r4 (middle), and w8.r4 (right), plotted relative to w1.r4.

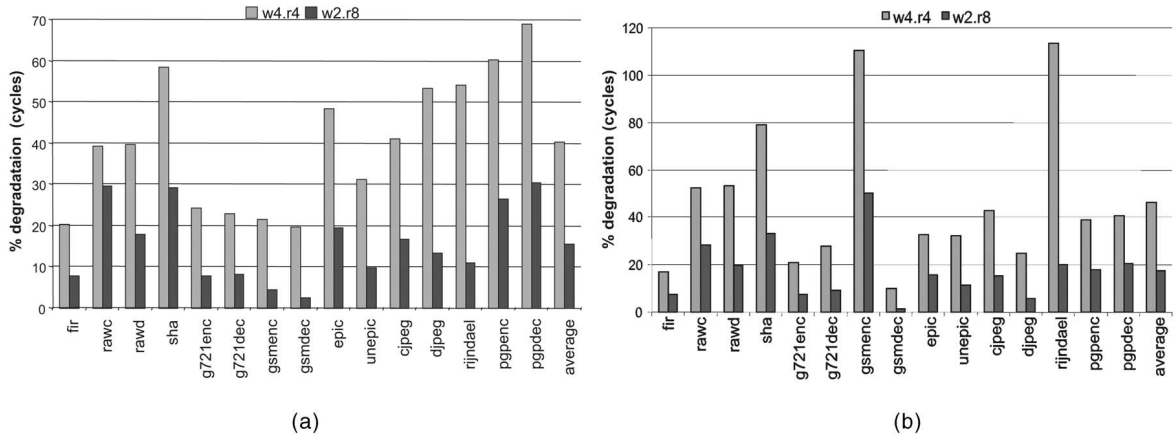


Fig. 13. Performance degradation while partitioning a 16-entry register file into 2 and 4 windows for (a) WIMS and (b) VLIW. For each benchmark, w2.r8 and w4.r4 are plotted relative to w1.r16.

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 for the WIMS processor.

Increasing the number of windows with a fixed number of total registers. Fig. 13a shows the percent slowdown in dynamic execution cycles due to partitioning a 16-entry register file into two and four windows on the WIMS processor. The base w1.r16 has no swap and move overhead. This provides an unachievable upper bound of the partitioning heuristic and helps to gauge how well our heuristics perform against an idealistic case. It should be noted that, as we partition the register file, the instruction encoding size decreases as fewer bits are required in the register field specifier within the instruction format, but we have not accounted for this in the data. For the 16-register case, w2.r8 achieves an average of only 16 percent degradation in performance, while the w4.r4 suffers an average of 40 percent degradation in performance. As we partition the register file, swaps and moves are required to distribute the VRs into all of the windows to reduce the spill pressure. Also, VRs referenced within the same instruction have to be assigned to the same partition as a single instruction must source all of its operands from the current active window. This prevents a perfect partitioning and increases the spill pressure on a given register window. Although having a large number of registers is preferred, encoding restrictions limit the actual size. A windowed design can give the appearance of a larger register file with moderate overhead.

Fig. 13b illustrates a similar experiment on the VLIW processor. We observe slightly worse results compared to the WIMS processor as partitioning can increase the register pressure on a single window, thus increasing spill code, which in turn has a larger impact on the VLIW machine.

Power Benefits. The impact of register windows on the total system power is now examined. By reducing spill code, the burden on the memory system to provide the operands is reduced, thereby increasing energy efficiency. Fig. 14 illustrates 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 (Automatic Place-and-Route) back-annotated parasitics. Input vectors were created at 1.8V and 100MHz operation by running assembled test cases through the pipeline and capturing

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

Fig. 14. Per instruction class energy and execution time for the WIMS processor at 100MHz.

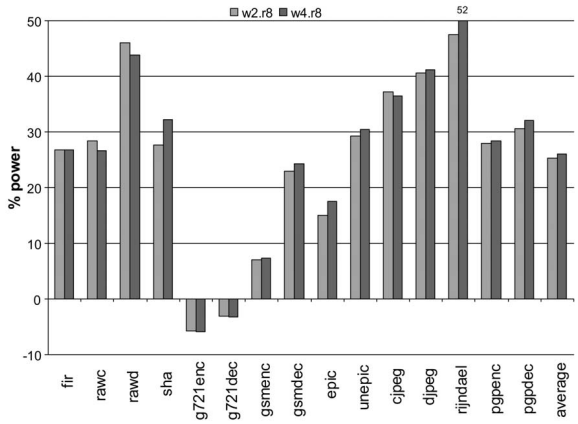


Fig. 15. Percent dynamic power improvement of w2.r8 and w4.r8 over the base case of w1.r8.

the switching activity [36]. The power due to different register file sizes was negligible (< 5 percent) when compared to the pipeline and memory power as the number of registers considered was no more than 32.

The graph in Fig. 15 shows the improvement in total dynamic power as the number of 8-register windows is increased on the WIMS processor. The total power includes the pipeline and memory power (instruction fetch, loads, and stores). Unlike performance, since spills dissipate more power to access memory in comparison to swaps/moves, spill reduction can result in a significant improvement in power consumed. For example, rijndael achieves a 52 percent reduction in power in the w4.r8 configuration. 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 as the number of memory accesses is reduced. It should be noted that, like performance, the power reduction is observed as we increase the effective number of available registers while restricting the ability to address only eight registers within an instruction. Although our heuristics were geared toward improving performance as opposed to power, it can be easily retargeted to optimize for power by weighting the savings in spill code more than the savings in swaps and moves.

4.3 Comparison among Different Partitioning Heuristics of Varying Estimation Accuracy

Optimal partitioning of VRs into partitions to minimize spills, swaps, and moves is an NP-hard problem. If there are n VRs and m partitions, an exponential number ($O(m^n)$) of possible assignments needs to be evaluated, which is clearly impractical. Compiler heuristics for register partitioning provides a trade-off between quality of solution and runtime. In this section, we explore this trade-off by comparing against two other heuristics—*global* and *fast*.

A region-based heuristic (called *region* and described in Section 3), where edge weights were computed statically prior to the partitioning process and partition weights were computed on the fly, was used in all previous experiments. This process, although fast, is inaccurate as the actual swap and move cost is a function of the current assignment of VRs to partitions. Accurate estimates of the swap and move penalties are possible if the number of swaps and moves are recomputed dynamically during partitioning by scanning the operations in the procedure. This method, though accurate, is inherently slow. In addition, region-based partitioning can result in suboptimal decisions if there are multiple regions with comparable profile weights.

To evaluate a more accurate heuristic, a semibrute force method was implemented wherein the basic FM-based partitioning methodology was retained, but two changes were made: The edge weights were computed during partitioning and the scope was extended to the whole procedure rather than a region. In order to reduce the computational complexity of considering too many VRs, a fixed number (set at compile time) of the most frequently occurring VRs is considered at a time. We implemented this slower semibrute force-like method (called *global*) to compare against our preferred *region* method to quantify the loss in partition quality. The partition weights for *global* were computed just as in *region*.

To evaluate another faster (compared to the *region* method), but less accurate heuristic, an FM-based *fast* heuristic was implemented with static estimation of edge weights while operating within a region scope. The spill estimation was performed by considering only the operation with the maximum intersecting live-ranges (see Section 3.5). Both the *fast* and *global* methods considered only 64 VRs at a time during partitioning. The *fast*, *region*, and *global* methods represent heuristics that are progressively more complex, while attempting to more accurately estimate the swap, move, and spill costs.

The graph in Fig. 16a compares the percent performance improvement in total execution cycles of the three heuristics for the w2.r8 configuration against the base w1.r8 for the WIMS processor. Overall, an average performance improvement of 8 percent, 11 percent, and 13 percent was obtained for the *fast*, *region*, and *global* methods, respectively. The *region* heuristic did considerably better than the *fast* method because it considered a larger set of VRs (whole region) and estimated spills more accurately. The *global* method had a procedure scope that used a more accurate dynamic estimation of swaps and moves, thereby performing better than the *region* method.

However, some of the results are not monotonically increasing as one would expect for more accurate methods. In some cases (gsmdec, rijndael), the *region* method performed better than the *global* method. These benchmarks

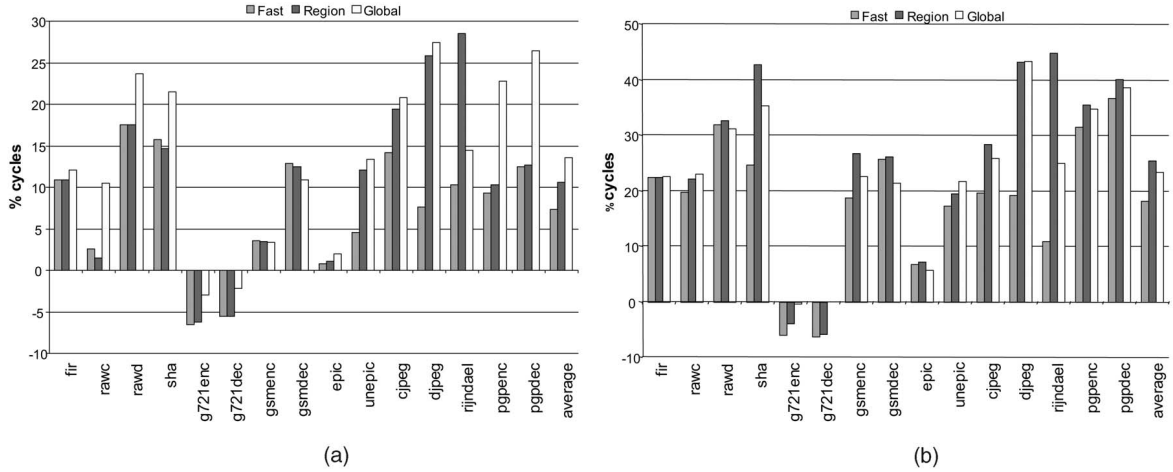


Fig. 16. Comparing performance between *fast*, *region*, and *global* heuristics for the 8-register (a) WIMS processor and the (b) VLIW processor. For each heuristic, w2.r8 is plotted relative to w1.r8.

had a single region that dominated execution time. The *region* method was more effective because it considered all VRs in the dominant region, whereas the global method could only examine 64 at a time. Also, since the underlying partitioning algorithm was greedy, an inaccurate estimation of swap, move, or spill cost sometimes results in the *fast* or the *region* heuristics doing better than the *global* method or the *fast* method doing better than the *region* method. While running our experiments, we observed that the compile time for the *global* method was 75 percent more than the *fast* method, while the *region* method was 40 percent less then the *global* method.

The graph in Fig. 16b repeats the previous experiment for the VLIW machine. On average, a performance improvement of 19 percent, 25 percent, and 23 percent is observed for the *fast*, *region*, and *global* methods, respectively. Again, as described previously, the heuristic uncertainties combined with the larger set of analyzed VRs caused the comparatively faster *region* method to perform better than the slower *global* method.

In summary, we prefer the *region* method as it performed close to the *global* method by employing a more intelligent heuristic with substantially faster compilation times.

5 Related Work

As an alternative to register windows, hardware and software schemes have been proposed in prior work to increase the effective number of registers. On the hardware side, register connection [20] and register queues [9], [30] 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 [9], [30]. 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 is 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.

A combination of 16-bit and 32-bit instructions have been used in mixed-mode architectures like the Thumb instruction set extensions in ARM [28] and MIPS-16 [32] to provide a balance between reducing code size and retaining performance [21]. The register windows have the advantage over this approach of allowing scalability. The number of effective registers can be increased to any large number using a fixed encoding.

On the software side, code generation for DSP processors has proven to be a challenge for compilers [24]. The irregularities of such architectures have motivated the use of new compiler techniques which were initially considered to be complex and time consuming. Graph partitioning has been used in compilers for multiclustered VLIW processors [6], [1], [4]. Several graph partitioning-based tools like Chaco [12] and Metis [16] have been widely used to implement mutlilevel Fiduccia-Mattheyses and other more sophisticated algorithms. A global register partitioning and interference graph-based approach has been used in the context of multicluster and multiregister file processors [13], [3]. Graph partitioning has also been explored in the context of partitioning program variables into multiple memory

banks [23]. 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, interwindow moves, and window swaps.

6 Conclusion

In this paper, we developed and implemented a 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 while maintaining a fixed instruction encoding. The compiler partitions the virtual registers in a procedure into multiple register windows, thus reducing the overall spill code while minimizing the overhead due to interwindow moves and window swaps. We evaluated our design over a wide range of processor and window configurations. Increasing the number of windows from one to two yielded an average performance improvement of 10 percent for the 4-register case and 11 percent for the 8-register case on the WIMS processor. The corresponding experiment on a 5-wide VLIW machine achieved an average performance improvement of 21 percent and 25 percent for the four and eight register configurations, respectively. An average power reduction of 25 percent for the 2-window 8-register over the 1-window case was observed on the WIMS processor. In the future, we plan to explore the use of register windows in software pipelined loops.

Acknowledgments

The authors 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 by the Engineering Research Centers Program of the US National Science Foundation (NSF) under award number EEC-9986866, NSF grant CCF-0347411, and equipment donated by Hewlett-Packard and Intel Corp.

References

[1] A. Aletà, J. Codina, J. Sánchez, A. González, and D. Kaeli, "Exploiting Pseudo-Schedules to Guide Data Dependence Graph Partitioning," *Proc. 11th Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 281-290, Sept. 2002.

[2] Analog Devices, *ADSP-219x/2191 DSP Hardware Reference Manual,* July 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," *Proc. ACM SIGPLAN Conf. Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems,* pp. 130-138, June 2002.

[4] M. Chu, K. Fan, and S. Mahlke, "Region-Based Hierarchical Operation Partitioning for Multicluster Processors," *Proc. SIGPLAN '03 Conf. Programming Language Design and Implementation,* pp. 300-311, June 2003.

[5] J.-L. Cruz, A. Gonzalez, M. Valero, and N. Topham, "Multiple-Banked Register File Architecture," *Proc. 27th Ann. Int'l Symp. Computer Architecture,* pp. 316-325, June 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," *Proc. 30th Ann. Intl Symp. Microarchitecture,* pp. 149-159, Dec. 1997.

[9] M.M. Fernandes, J. Llosa, and N. Topham, "Allocating Lifetimes to Queues in Software Pipelined Architectures," *Proc. Third Int'l Euro-Par Conf.,* pp. 1066-1073, Aug. 1997.

[10] C. Fiduccia and R. Mattheyses, "A Linear Time Heuristic for Improving Network Partitions," *Proc. 19th Design Automation Conf.,* pp. 175-181, 1982.

[11] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *Proc. Fourth IEEE Workshop Workload Characterization,* pp. 10-22, Dec. 2001.

[12] B. Hendrickson and R. Leland, *The Chaco User's Guide.* Sandia Nat'l Laboratories, July 1995.

[13] J. Hiser, S. Carr, and P. Sweany, "Global Register Partitioning," *Proc. Ninth Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 13-23, Oct. 2000.

[14] W.M. Hwu et al., "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *J. Supercomputing,* vol. 7, no. 1, pp. 229-248, May 1993.

[15] Intel Corp., *Intel IA-64 Software Developer's Manual,* 2002.

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

[17] V. Kathail, M. Schlansker, and B. Rau, "HPL PlayDoh Architecture Specification: Version 1.0," Technical Report HPL-93-80, Hewlett-Packard Laboratories, Feb. 1993.

[18] B. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical J.,* vol. 49, no. 2, pp. 291-207, Feb. 1970.

[19] H. Kim, "Region-Based Register Allocation for EPIC Architectures," PhD thesis, Dept. of Computer Science, New York Univ., 2001, www.crest.gatech.edu/publications/hansooth.pdf.

[20] K. Kiyohara, S.A. Mahlke, W.Y. Chen, R.A. Bringmann, R.E. Hank, S. Anik, and W.W. Hwu, "Register Connection: A New Approach to Adding Registers into Instruction Set Architectures," *Proc. 20th Ann. Int'l Symp. Computer Architecture,* pp. 247-256, May 1993.

[21] A. Krishnaswamy and R. Gupta, "Profile Guided Selection of ARM and Thumb Instructions," *Proc. ACM SIGPLAN Conf. Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems,* pp. 55-63, June 2002.

[22] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *Proc. 30th Ann. Int'l Symp. Microarchitecture,* pp. 330-335, 1997.

[23] R. Leupers and D. Kotte, "Variable Partitioning for Dual Memory Bank dsps," *Proc. IEEE Int'l Conf. Acoustics Speech and Signal Processing,* pp. 1121-1124, May 2001.

[24] P. Marwedel and G. Goossens, *Code Generation for Embedded Processors.* Boston: Kluwer Academic, 1995.

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

[26] M. Poletto and V. Sarkar, "Linear Scan Register Allocation," *ACM Trans. Programming Languages and Systems,* vol. 21, no. 5, pp. 895-913, Sept. 1999.

[27] R. Ravindran, R. Senger, E. Marsman, G. Dasika, M. Guthaus, S. Mahlke, and R. Brown, "Increasing the Number of Effective Registers in a Low-Power Processor Using a Windowed Register File," *Proc. 2003 Int'l Conf. Compilers, Architecture, and Synthesis for Embedded Systems,* pp. 125-136, 2003.

[28] D. Seal, *ARM Architecture Reference Manual.* London: Addison-Wesley, 2000.

[29] R. Senger, E. Marsman, M. McCorquodale, F. Gebara, K. Kraver, M. Guthaus, and R. Brown, "A 16-Bit Mixed-Signal Microsystem with Integrated CMOS-MEMS Clock Reference," *Proc. 40th Design Automation Conf.,* pp. 520-525, 2003.

[30] M. Smelyanskiy, G. Tyson, and E. Davidson, "Register Queues: A New Hardware/Software Approach to Efficient Software Pipelining," *Proc. Ninth Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 3-12, Oct. 2000.

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

[32] *MIPS32 Architecture for Programmers Volume IV-a: The MIPS16 Application Specific Extension to the MIPS32 Architecture,* MIPS Technologies, Mar. 2001.

[33] Tensilica Inc., *Xtensa Architecture and Performance,* Sept. 2002, http://www.tensilica.com/xtensa_arch_white_paper.pdf.
[34] Texas Instruments, *TMS320C54X DSP Reference Set,* Mar. 2001, http://www-s.ti.com/sc/psheets/spru131g/spru131g.pdf.
[35] Texas Instruments, *TMS320C6000 CPU and Instruction Set Reference Guide,* June 2004, http://focus.ti.com/lit/ug/spru189f/spru189f.pdf.
[36] V. Tiwari, S. Malik, and A. Wolfe, "Power Analysis of Embedded Software: A First Step towards Software Power Minimization," *IEEE Trans. Very Large Scale Integration (VLSI) Systems,* vol. 2, no. 4, pp. 437-445, 1994.
[37] Trimaran, "An Infrastructure for Research in ILP," 2000, http://www.trimaran.org.

Rajiv A. Ravindran received the MTech degree in computer science from the Indian Institute of Technology, Kanpur. He is a PhD student in the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. His research interests include compilation for low-power embedded DSPs, automatic compiler and architecture synthesis, and compilation for high-performance processors. He is a student member of the IEEE and the ACM.

Robert M. Senger received the ScB degree from Brown University in May 2000 in computer engineering and subsequently began his graduate studies in VLSI design at the University of Michigan, Ann Arbor. His research interests include low-power, mixed-signal chip design which is being explored within the context of low power microcontrollers for remote sensor systems. Currently, he is interning at the IBM Austin Research Lab, where he is investigating low-power design in advanced processing technologies. He hopes to receive the PhD degree by the end of 2005.

Eric D. Marsman received the BSE degree in electrical engineering magna cum laude in April 2000 from the University of Michigan. He received the MSE degree in VLSI in January 2002 from the University of Michigan. He is currently pursuing the PhD degree in VLSI, also at the University of Michigan. During his undergraduate career, he had internships with Motorola working in the Test Engineering Department. He also worked on the CAD team for the Hammer family of products with Advanced Micro Devices. Agilent Technologies also gave him experience working in an ASIC design lab on high-speed communication chips. During his graduate experience, he was a graduate student instructor for the Introduction to VLSI class. For his outstanding work over the course of three semesters, he received the EECS Outstanding GSI for 2001, the ASEE Outstanding GSI for 2001, and the Rackham Outstanding GSI for 2002.

Ganesh S. Dasika received the BSE degree in computer engineering from the University of Michigan. He is a PhD student in the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. His research interests mainly include designing and compiling for embedded architectures. He is a student member of the IEEE.

Matthew R. Guthaus received the BSE degree in computer engineering and the MSE degree in electrical engineering from the University of Michigan in 1998 and 2000, respectively. He is currently a PhD candidate in electrical engineering at the University of Michigan. While working on his master's degree, he designed and tested the digital parts of a mixed-signal, 8-bit microcontroller for sensor and actuator applications. His dissertation research is on design automation for the optimization of parametric yield. Other research interests include physical design automation, low-power architecture for embedded systems, and algorithm specific microprocessors.

Scott A. Mahlke received the PhD degree in electrical and computer engineering from the University of Illinois at Urbana-Champaign. He is the Morris Wellman Assistant Professor of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. He directs the Compilers Creating Custom Processors research group, which focuses on the design of application specific processors and hardware accelerators. His research interests include compilers, computer architecture, and high-level synthesis. He is a member of the IEEE and the ACM.

Richard B. Brown received the BS and MS degrees in electrical engineering from Brigham Young University, Provo, Utah, in 1976 and the PhD degree in electrical engineering (solid-state sensors) from the University of Utah, Salt Lake City, in 1985. From 1976 to 1981, he was vice-president of engineering at Holman Industries, Oakdale, California, and then manager of computer development at Cardinal Industries, Webb City, Missouri. In 1985, he joined the faculty of the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor. He became dean of engineering at the University of Utah, Salt Lake City, in July 2004. He has conducted major research projects in the areas of solid-state sensors, mixed-signal circuits, GaAs and silicon-on-insulator circuits, and high-performance and low-power microprocessors. He is a member of the ACM and a senior member of the IEEE. He is chairman of the MOSIS Advisory Council for Education.