Exploiting Idle Floating-Point Resources For Integer Execution

S.Subramanya Sastry
Computer Sciences Dept.
University of Wisconsin-Madison
sastry@cs.wisc.edu

Subbarao Palacharla
Computer Sciences Dept.
University of Wisconsin-Madison
subbarao@cs.wisc.edu

James E. Smith
Dept. of ECE
University of Wisconsin-Madison
jes@ece.wisc.edu

Abstract

In conventional superscalar microarchitectures with partitioned integer and floating-point resources, all floating-point resources are idle during execution of integer programs. Palacharla and Smith [26] addressed this drawback and proposed that the floating-point subsystem be augmented to support integer operations. The hardware changes required are expected to be fairly minimal.

To exploit these idle floating resources, the compiler must identify integer code that can be profitably offloaded to the augmented floating-point subsystem. In this paper, we present two compiler algorithms to do this. The *basic* scheme offloads integer computation to the floating-point subsystem using existing program loads/stores for inter-partition communication. For the SPECINT95 benchmarks, we show that this scheme offloads from 5% to 29% of the total dynamic instructions to the floating-point subsystem. The *advanced* scheme inserts copy instructions and duplicates some instructions to further offload computation. We evaluate the effectiveness of the two schemes using timing simulation. We show that the advanced scheme can offload from 9% to 41% of the total dynamic instructions to the floating-point subsystem. In doing so, speedups from 3% to 23% are achieved over a conventional microarchitecture.

1 Introduction

Most current superscalar processors [17, 18, 16, 4] are based on the microarchitecture shown in Figure 1. The instruction fetch unit reads multiple instructions from the instruction cache, decodes them, and places them in instruction buffers for execution by the integer and floating-point subsystems. The integer subsystem contains the integer register file, integer instruction buffers, and a number of integer functional units that operate on integer operands. The floating-point subsystem is similar to the integer subsystem except its functional units perform floating-point arithmetic, and it does not control the execution of load and store instructions.

For the purpose of this work, the most important feature of this microarchitecture is the partitioning of processor resources into integer and floating-point subsystems. There are a number of reasons, both technical and historical, for this partitioning. The two most pertinent to our discussion follow.

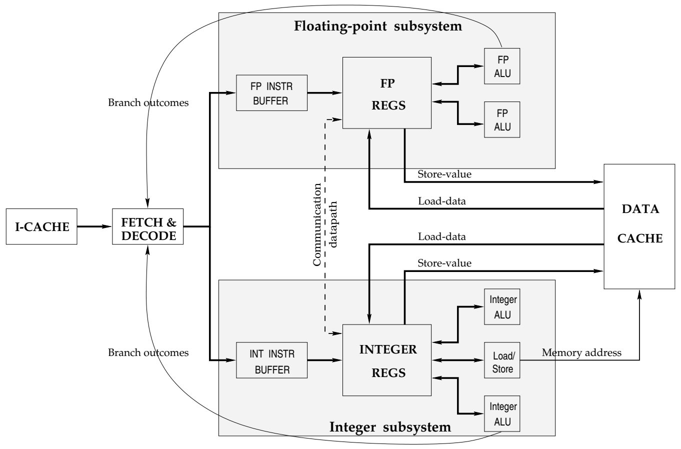

Figure 1: Datapath of a conventional microarchitecture.

- Dividing the hardware into two simpler subsystems tends to make both control and data paths faster. In each of the individual subsystems, there are fewer register file ports, fewer data bus sources and destinations, and fewer alternatives for many control decisions, as compared with a processor where all the units, registers and busses are shared. The result is a faster clock cycle.
- Using two subsystems allows specialization. In particular, the register files, data paths, and functional units can be of different widths. In older machines, especially when hardware costs were much higher, this was very important. Integer data could be half the width of floating-point data (or sometimes even less). This reason, however, is fast disappearing – the current trend is to make both integer and floating-point data 64 bits wide. This trend reinforces the optimization described in this paper.

The main disadvantage of the partitioned approach is that it can lead to an imbalance in resource usage. In particular, it leads to idle floating-point resources (instruction buffers, issue logic, and functional units) during execution of integer programs. A large number of common programs (compilers, editors, databases, operating systems) [28, 22] are integer programs that execute very few (if any)

To appear at the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementaiton, Montreal, Canada, June 17-19, 1998.

floating-point operations. Palacharla and Smith [26] addressed this drawback and proposed a more general microarchitecture in which the floating-point subsystem can also execute integer instructions. The hardware modifications required of existing architectures are minimal and are similar in spirit to the Intel MMX extensions to the IA-32 instruction set [20] and the Sun SPARC Visual Instruction Set (VIS) extensions [19].

- The floating-point functional units are augmented to support simple integer and logical operations. The more complex operations, integer multiplication and division, are fairly rare and do not have to be included. The simple integer units can be embedded in the existing floating-point units and do not require additional busses. With today's transistor budgets, this additional hardware cost is small.
- Additional opcodes have to be added to the instruction set to control the new integer instructions. This is probably a more significant consideration because it requires instruction set extensions and re-compilation. In our study, we used 22 extra opcodes.

In the rest of the paper, the integer subsystem is called the *INT* subsystem and the *augmented* floating-point subsystem is called the *FP*$_a$ subsystem. In order to use the proposed microarchitecture effectively, the compiler must identify integer operations that can be offloaded to the augmented floating-point subsystem, i.e., the compiler has to partition the program by assigning instructions to the INT and FP$_a$ subsystems. This paper presents and evaluates compiler algorithms for achieving such code partitioning. The most important goal of our algorithms is to maximize the utilization of the floating-point resources during execution of integer code. The main source of difficulty in doing so is handling inter-partition communication. Such communication can be achieved either through existing program loads/stores or through the use of *copy* instructions that move data between the INT and the FP$_a$ register files. Alternatively, communication can be entirely avoided by duplicating code. The problem is in striking the right balance between these communication mechanisms while maximizing the resultant benefits. In this paper, we present two code partitioning schemes. The first scheme, called the *basic* partitioning scheme only uses existing program loads and stores for inter-partition communication. The second scheme, called the *advanced* partitioning scheme adds copy instructions and code duplication for inter-partition communication. We use simulations to study the effectiveness of these compiler schemes. Results for the SPEC95 integer benchmarks show that the compiler can offload from 9% to 41% of the total dynamic instructions to the FP$_a$ subsystem. This results in performance improvements ranging from 2.5% to 23.1% on a 4-way (2 int + 2 fp) issue machine.

Before presenting further details of the proposed schemes, let us understand current microarchitectures in a little more detail. Consider the function shown in Figure 2 that computes the floating-point vector sum *c[] = a[] + b[]*. Instructions I3 and I4 are floating-point load instructions. Even though they are called "floating-point" instructions, they actually issue from the integer instruction buffers (Figure 1) and execute in the load/store unit of the integer subsystem. This unit computes the effective memory address and sends it to the data cache. The data cache retrieves the data (if there is a hit) or gets it from memory (if there is a cache miss) and sends the data over to the floating-point register file – via the appropriate "Load-data" path shown in Figure 1. Instruction I5, a floating-point add

```
void fp_vector_sum(int n,
                   float c[],
                   float a[],
                   float b[])
{
  int i;

  for (i = 0; i < n; i++)
      c[i] = a[i]+b[i];

}
```

C function to compute floating-point vector sum

```
I1:   move   $8, $0
I2:   blez   $4, $L21
   $L23:
I3:   l.s    $f0, 0($6)
I4:   l.s    $f2, 0($7)
I5:   add.s  $f0, $f0, $f2
I6:   s.s    $f0, 0($5)
I7:   addu   $5, $5, 4
I8:   addu   $6, $6, 4
I9:   addu   $7, $7, 4
I10:  addu   $8, $8, 1
I11:  slt    $2, $8, $4
I12:  bne    $2, $0, $L23
   $L21:
      j      $31
```

Assembly for the C code

Figure 2: Example floating-point code.

instruction, executes entirely in the floating-point subsystem. Instruction I6 is a floating-point store which computes the effective address in the load/store unit of the integer subsystem. The value to be stored is retrieved from the floating-point register file and gets written into the data cache. The only other instruction that will be of interest to us is the conditional branch instruction I12 which checks for loop termination and sends the outcome back to the instruction fetch unit.

If the above example were computing an *integer* vector sum instead of a *floating-point* vector sum, then the floating-point resources would be completely idle.[1] However, if the floating-point adder could also perform integer adds, then the integer add could be offloaded to the floating-point subsystem by using the same code as shown above, with the one exception that instruction I5 would have a new opcode specifying "integer add of floating-point registers". In the rest of the paper, we will strive for this type of function offloading, as well as offloading of some branch decisions.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents some terminology and describes the data structure used by the algorithms. Section 4 discusses the goals of the partitioning algorithms. Section 5 presents the basic partitioning scheme used by the compiler. In Section 6, the advanced partitioning scheme is presented. In Section 7, we present the evaluation methodology and then present performance improvements resulting from the partitioning schemes discussed in Sections 5 and 6. We present our conclusions in Section 8.

2 Related work

A number of clustered/partitioned architectures for exploiting fine grain instruction level parallelism have been proposed in the literature [24, 15, 13, 12, 27, 8, 9]. All of these architectures attempt to reduce hardware complexity to obtain a faster clock cycle. In all these architectures, the hardware resources are uniformly partitioned across multiple clusters resulting in homogeneous clusters. However, in the superscalar architecture considered in this paper, the clusters are heterogeneous and only one of the clusters (INT subsystem) can address memory. Since all loads and stores always execute on one cluster, it is much harder to achieve balanced partitioning for integer programs, because memory addressing and access forms a big portion of program execution in most integer programs.

[1] Although, in this example, the floating-point resources are nearly idle even when executing floating-point code! This is not uncommon, and the techniques proposed in this paper can probably be effective with floating-point code.

In the context of VLIW machines (ELI, Multiflow TRACE, LC-VLIW), there has been prior work to partition code across clusters [14, 23, 5, 3, 11]. Ellis' BUG (bottom-up greedy) assignment algorithm in the Bulldog compiler [14] assigns instructions to functional units in all clusters. The algorithm is based on the assumption that functional units are the only limiting resource in the machine. Inter-cluster communication bandwidth is not considered a limiting resource. In his thesis [5], Banerjia shows that when inter-cluster communication resources are scarce, the excessive copies introduced by BUG can hurt performance.

Capitanio, Dutt, and Nicolau [3] present compiler schemes to achieve balanced code partitions while minimizing inter-cluster communication. However, this study applied the code partitioning techniques *only* to straight-line loop bodies of floating-point codes.

Unlike the BUG algorithm, the partitioning algorithms presented by Banerjia [5], Capitanio et al. [3], and Desoli [11] attempt to maximize utilization of hardware resources in all clusters while minimizing inter-cluster communication. In this respect, these algorithms are similar in spirit to our partitioning algorithms. The main difference is that these algorithms are based on a statically-scheduled machine model with homogeneous clusters, whereas our algorithms are for a dynamically-scheduled superscalar machine with heterogeneous clusters. This simplifies the problem since it is not necessary to schedule for individual issue slots. A further difference between our approaches is that these earlier algorithms do not consider code duplication as a means of eliminating inter-cluster communication.

In the Multicluster architecture [15], the hardware takes care of inter-cluster communication automatically. The compiler does not insert explicit copy instructions. The hardware performs inter-cluster copying based on the architectural registers used by the operands of an instruction. The compiler performs code partitioning by assigning registers to instruction operands. The primary objective of the partitioning algorithms is to achieve load balance between clusters. Since all hardware resources including the register file are partitioned equally between the clusters, load balance is considered far more important than the the increased communication between clusters. Again, code duplication is not used to eliminate some of this communication, though it is recognized as a possible improvement to their algorithms.

The MMX extensions to the IA-32 instruction set [1] are aimed at speeding up multimedia programs by performing multiple bit, byte, or word operations in parallel like a SIMD processor. To maintain backward compatibility and avoid changes to the operating system, these extensions were implemented by augmenting the floating-point subsystem to perform these integer operations by using the floating-point registers to hold the integer data. To exploit these MMX extensions to their full potential, it is necessary to vectorize sub-word integer operations. However, for programs that do not have any SIMD parallelism (non-multimedia programs), the compiler can still profitably exploit these MMX extensions in the manner described in this paper.

3 Terminology and Data Structures

In this section, we first describe the data structure used by the partitioning algorithms. We then present some terminology to aid subsequent discussion.

The primary data structure used in code partitioning is the register dependence graph (RDG) which represents all the register dependences in a program. The RDG is a directed graph which has a node corresponding to each static instruction in the program. There is an edge from node v_i to node v_j if instruction i produces a value that *could* be consumed by instruction j. These edges are determined by solving the *reaching-definitions* dataflow problem [2]. Load and store instructions are special-cased in the RDG to simplify the partitioning algorithms. Each load instruction is split into two nodes - one representing the load address and the other representing the loaded value. Similarly, each store instruction is split into two nodes - one representing the store address and the other representing the store value. This is done because a load instruction computes the address in the INT subsystem, but the value can be loaded into either subsystem. Likewise, the value being stored can come from either the INT subsystem or the FP$_a$ subsystem.

Figure 3 shows C code from `invalidate_for_call`, a frequently executed function in the SPEC benchmark `gcc`. The for loop in the program runs through all pseudo registers and does some bookkeeping for those that are invalidated by function calls. The figure shows assembly code compiled for a conventional microarchitecture. The figure also shows the RDG[2] for the program fragment. Nodes $2, 8$, and 11 correspond to load instructions and have been split. Address nodes have the suffix "**a**" while value nodes have the suffix "**v**". To show that both nodes correspond to a single program instruction, the split nodes have been enclosed in a bigger oval node. Similarly, node 14 corresponds to a store instruction and has been split.

Given the RDG, the *backward slice* of G with respect to a node v, denoted by $Backward\text{-}Slice(G, v)$, is defined to be the set of all nodes from which v can be reached. Similarly, the *forward slice* of G with respect to v, denoted by $Forward\text{-}Slice(G, v)$, is defined to be the set of all nodes that can be reached from v. Note that backward slices, as defined, do not go past load-value nodes. Similarly, forward slices do not go past address nodes. This is the primary difference from the traditional definition of slices [21].

Given this definition of a forward slice, all forward slices in a RDG terminate at memory addresses, call arguments, return values, branch outcomes, or store values. Using these terminal nodes, we define various computational slices as follows.

The *LdSt slice* of a program is defined to be the set of all instructions that contribute to the computation of addresses for load/store instructions. Given the RDG G of a program, let

$LS(G)$ = Set of load/store *address* nodes in G. Then,
LdSt slice = $\cup_{v \in LS(G)} Backward\text{-}Slice(G, v)$.

In Figure 3, $LS(G) = \{2a, 8a, 11a, 14a\}$; the LdSt slice has been marked.

A branch slice is defined to be the set of instructions involved in computing a branch outcome. This can be computed by starting from a branch node and computing its backward slice. Store value, call argument, and return value slices are defined similarly.

4 Partitioning Goals

In an out-of-order superscalar processor, instruction window size and issue width primarily determine how much instruction level parallelism (ILP) can be exploited by the hardware. Bigger the instruction window and wider the issue width, more the ILP that can

[2] The RDG is based on the assembly code shown. However, in the compiler, it is based on the intermediate representation of the program.

```
extern unsigned long regs_invalidated_by_call;

for (regno = 0; regno < FIRST_PSEUDO_REGISTER; regno++)
     if (regs_invalidated_by_call & (1 << regno)) {
         delete_equiv_reg(regno);
          if (reg_tick[regno] >= 0)
              reg_tick[regno]++;
     }
```

Program fragment in C from *gcc*

```
I1:       move     $16, $0                       || regno = 0
    $L5:
I2:       lw       $2, regs_invalidated_by_call  ||
I3:       sra      $2, $2, $16                   || $2 = regs_invalidated_by_call & (1 << regno)
I4:       andi     $2, $2, 0x1                   ||
I5:       beq      $2, $0, $L4
I6:       move     $4, $16
I7:       jal      delete_equiv_reg
I8:       lw       $3, reg_tick                  ||
I9:       sll      $2, $16, 2                    || $4 = reg_tick[regno]
I10:      addu     $2, $2, $3                    ||
I11:      lw       $4, 0($2)                     ||
I12:      bltz     $4, $L4
I13:      addu     $4, $4, 1                     || reg_tick[regno]++
I14:      sw       $4, 0($2)                     ||
    $L4:
I15:      addu     $16, $16, 1                   || regno++
I16:      slt      $2, $16, 66                   || $2 = regno < FIRST_PSEUDO_REGISTER
I17:      bne      $2, $0, $L5
```

Assembly code for the program

Instruction format is : *<op>* *<dst>, <src1>, <src2>*

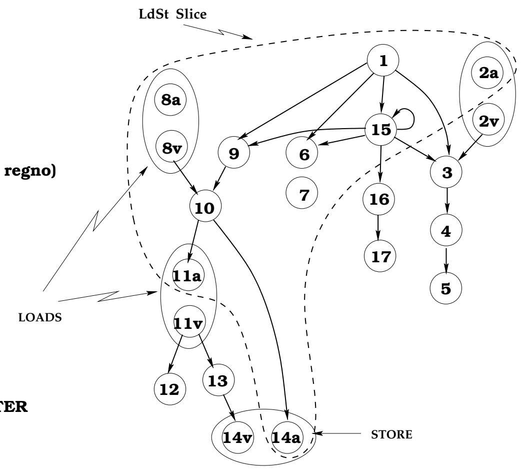

Figure 3: An example program fragment.

be exploited. In the augmented architecture, since the FP$_a$ subsystem is capable of executing integer instructions, if properly exploited, this effectively doubles the window size and issue width for integer programs. Thus, our partitioning schemes concentrate on maximizing the utilization of the floating-point instruction window and issue logic while keeping to a minimum any resulting communication and instruction overheads.

In our machine model, since only the INT subsystem can execute loads and stores, all loads and stores are assigned to the INT partition. As discussed previously, the LdSt slice of a program computes load and store addresses. Potentially, portions of this LdSt slice could be assigned to the FP$_a$ partition. However, since all memory addresses are ultimately needed in the INT subsystem, this necessitates the use of copy instructions on some address computation paths. This is undesirable since memory addressing and access tend to be on the critical path of most programs, integer or floating-point. Hence, in our partitioning algorithms, we assign the complete LdSt slice of a program to the INT partition.

Calling conventions impose further restrictions on our schemes. These conventions require integer-valued arguments and return values to be passed/returned in integer registers. So, call argument and return value nodes are always assigned to the INT partition. Further, if call argument and return value slices are assigned to the FP$_a$ partition, copies would be required to adhere to calling conventions. If it is unprofitable to introduce these copies, these slices would instead be assigned to the INT partition.

The remaining computational slices, branch slices and store-value slices, are potential candidates for assignment to the FP$_a$ partition. Some of these slices can be assigned to the FP$_a$ partition without requiring any inter-partition communication. It is shown by Palacharla and Smith [26] (and is borne out by our simulations) that the LdSt slices of integer programs account for close to 50% of all dynamic instructions executed. This puts an upper bound on the size of the FP$_a$ partition that our algorithms can identify. Calling conventions and communication overheads would reduce this size further.

Based on these observations, we decided to let the partitioning schemes greedily assign as much of the branch and store-value slice to the FP$_a$ partition as possible. The goal of the partitioning algorithms is to maximize the size of the FP$_a$ partition. The limitations of these greedy partitioning strategies are discussed in Section 6.6. We are now ready to present details of our code partitioning schemes. The next section discusses the basic partitioning scheme.

5 Basic Partitioning Scheme

In this section, we present the basic partitioning scheme that attempts to partition the program without introducing extra instructions for inter-partition communication. It achieves all inter-partition communication through existing program loads/stores. We first describe the conditions that need to be satisfied by the INT and FP$_a$ partitions to adhere to this restriction. We then describe a partitioning algorithm that identifies partitions satisfying these conditions.

5.1 Partitioning conditions

Given a program P and its RDG G, the goal is to partition G into a INT partition, $I(G)$, and a FP$_a$ partition, $F(G)$, that satisfy the following conditions:

1. $F(G)$ and $I(G)$ are disjoint.
2. If $v \in F(G)$, then $Backward\text{-}Slice(G, v) \cap I(G) = \emptyset$. For a node $v \in F(G)$, this condition specifies that v or any of its ancestors should not *receive* any value from $I(G)$ via a register.
3. If $v \in F(G)$, then $Forward\text{-}Slice(G, v) \cap I(G) = \emptyset$. For a node $v \in F(G)$, this condition specifies that v or any of its descendants should not *supply* any value to $I(G)$ via a register.

The last two conditions are a manifestation of the restriction not to introduce any extra communication instructions in the program. It can easily be shown that failure to satisfy either of these two conditions necessitates the use of extra copy instructions to maintain correctness of the generated assembly code.

5.2 Partitioning Algorithm

We now present a simple algorithm to find the largest set $F(G)$ that satisfies the partitioning conditions. We aim for the largest set because the goal of our algorithms is to maximize the size of the FP$_a$ partition. Let G_u be the undirected graph corresponding to G, i.e. G_u consists of the same vertices and edges as G, but the edges are undirected. Then, the partitioning conditions can be interpreted as: If $v \in F(G_u)$, then v is not reachable from any node in $I(G_u)$. So, every connected component in G_u either belongs to $I(G_u)$ or $F(G_u)$ but is not shared between the two partitions. Since load/store address nodes, argument and return-value nodes are assigned to the INT partition, connected components containing these nodes are assigned to the INT partition. All other components are assigned to the FP$_a$ partition. These components contain instructions that compute *only* branch outcomes and store values.

Consider the graph in Figure 4 which corresponds to the example presented in Figure 3. The partitions identified by the basic partitioning algorithm are marked. The graph has four connected components. One component consists of the nodes $\{11v, 12, 13, 14v\}$. This component computes the store value for the store instruction I14. Since this component does not contain any load/store address nodes, it is assigned to FP$_a$. In contrast, all the other components contain load/store address nodes and hence are assigned to INT. Figure 4 also shows the partitioned assembly code. Integer instructions that execute in FP$_a$ are shown in **bold** with a ”**,c**” suffix. The load and store instructions (I11 and I14) are italicized to point out that these instructions are converted to floating-point load and store instructions.

5.3 Limitations of the Basic Partitioning Scheme

The basic partitioning algorithm is simple and efficient[3]. However, it misses opportunities to assign more computation to the FP$_a$ partition because of the restriction not to introduce extra instructions. Consider the partitioning in Figure 4 again. The branch slices $\{1, 15, 16, 17\}$ and $\{1, 15, 2v, 3, 4, 5\}$ contain the addressing instructions $I1$ and $I15$ and hence, could not be assigned to FP$_a$. However, if we relax the restriction on introducing extra instructions in the program, we could assign these branch slices to the FP$_a$ subsystem as shown below. Suppose the architecture has instructions to copy values directly between the INT and FP$_a$ register files[4]. Then, copy instructions can be inserted in the INT subsystem to copy the results of $I1$ and $I15$ to the FP$_a$ subsystem. This allows the earlier branch slices to execute in FP$_a$. Figure 5 shows the resulting assembly code and the associated RDG. In this example, copy instructions have enabled the offloading of five more instructions to FP$_a$. Since I1$_c$ is outside the loop, copy overheads are incurred every loop iteration only for instruction I15$_c$.

For this example, code duplication can be used to achieve the same partitioning as realized by inserting copy instructions. In the C code fragment of our example shown in Figure 3, the loop induction variable `regno` is used both for address computation as well as for branch computation. By duplicating `regno` in FP$_a$, the two pieces of code can proceed independently without any communication[5]. Figure 6 shows the assembly code and the associated RDG when this is done. I1$_d$ and I15$_d$ are duplicated instructions and enable five more instructions (relative to the basic partitioning scheme) to be offloaded to the FP$_a$ subsystem. Since I1$_d$ is outside the loop, overheads are incurred each loop iteration only for instruction I15$_d$.

Calling conventions further limit the ability of the basic partitioning algorithm to offload computation to the FP$_a$ partition. All components containing argument and return value nodes are assigned to the INT partition by the basic partitioning scheme. Copy instructions can alleviate this problem too. One could let the partitioning algorithm ignore the restrictions imposed by the calling conventions and later, when necessary, introduce copy instructions to adhere to these conventions.

Thus, copy instructions and code duplication can achieve better code partitioning. However, copy and duplicate instructions not only increase the size of the FP$_a$ partition, but can also increase the total number of dynamic instructions executed, which can degrade performance. Care must be taken to minimize the drawbacks of copy and duplicate instructions. The advanced partitioning scheme to be presented in the next section uses a cost model to determine where copy instructions or duplicate code will help.

6 Advanced Partitioning Scheme

In this section, we study advanced partitioning techniques that give better partitions than the basic partitioning scheme. As discussed previously, this is achieved through the use of copy instructions to achieve inter-partition communication and through code duplication to eliminate inter-partition communication. However, it is important to minimize overheads introduced by these instructions. We now present a cost model that enables this.

6.1 Cost Model

The cost model to be presented computes profitability of offloading integer instructions to FP$_a$ while taking into account any overheads introduced due to copies and duplicates. Intuitively, the benefit from a copy or a duplicated instruction is the number of extra *dynamic* instructions that will execute in the FP$_a$ subsystem as a result of the copy/duplicate. Given a RDG G,

Let S_{copy} be the set of nodes in G for which copy instructions are inserted.
Let S_{dupl} be the set of nodes in G which are duplicated.
Let S_c be the set of nodes in G that can be moved from INT to FP$_a$ as a result of the copies and duplicates.

The nodes in S_c execute in FP$_a$ yielding a bigger FP$_a$ partition. However, execution of nodes in S_{copy} and S_{dupl} introduces overhead in the program. This is quantified by the following equations.

$$\begin{aligned} Benefit &= \sum_{v \in S_c} n_{B(v)} \\ Overhead &= o_{copy} * \sum_{v \in S_{copy}} n_{B(v)} \\ &\quad + o_{dupl} * \sum_{v \in S_{dupl}} n_{B(v)} \end{aligned}$$

where $B(I)$: Basic block containing instruction I
n_B : Runtime execution count of basic block B

[3] Linear in the number of nodes and edges of the RDG.

[4] Such instructions are present in a number of instruction sets (e.g. MIPS [10] and Alpha [6]).

[5] There is control flow communication between the two through the shared fetch unit as shown in Figure 1.

```
I1:        move      $16, $0
    $L5:
I2:        lw        $2, regs_invalidated_by_call
I3:        sra       $2, $2, $16
I4:        andi      $2, $2, 0x1
I5:        beq       $2, $0, $L4
I6:        move      $4, $16
I7:        jal       delete_equiv_reg
I8:        lw        $3, reg_tick
I9:        sll       $2, $16, 2
I10:       addu      $2, $2, $3
I11:       lw        $f0, 0($2)
I12:       bltz,c    $f0, $L4
I13:       addu,c    $f0, $f0, 1
I14:       sw        $f0, 0($2)
    $L4:
I15:       addu      $16, $16, 1
I16:       slt       $2, $16, 66
I17:       bne       $2, $0, $L5
```

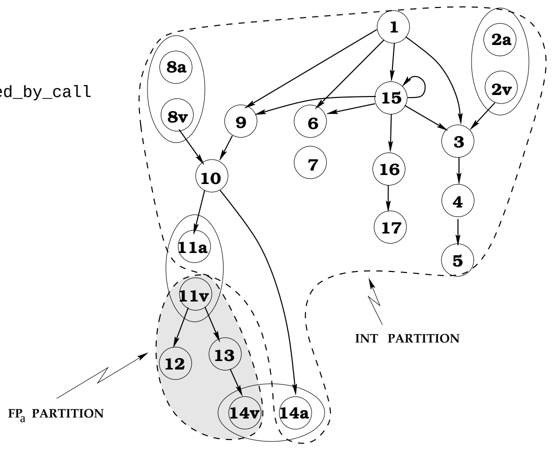

Figure 4: Basic code partitioning.

```
I1:        move      $16, $0
I1c:       cp_to_fp  $16, $f2
    $L5:
I2:        lw        $f4, regs_invalidated_by_call
I3:        sra,c     $f4, $f4, $f2
I4:        andi,c    $f4, $f4, 0x1
I5:        beq,c     $f4, $0, $L4
I6:        move      $4, $16
I7:        jal       delete_equiv_reg
I8:        lw        $3, reg_tick
I9:        sll       $2, $16, 2
I10:       addu      $2, $2, $3
I11:       lw        $f0, 0($2)
I12:       bltz,c    $f0, $L4
I13:       addu,c    $f0, $f0, 1
I14:       sw        $f0, 0($2)
    $L4:
I15:       addu      $16, $16, 1
I15c:      cp_to_fp  $16, $f2
I16:       slt,c     $f4, $f2, 66
I17:       bne,c     $f4, $0, $L5
```

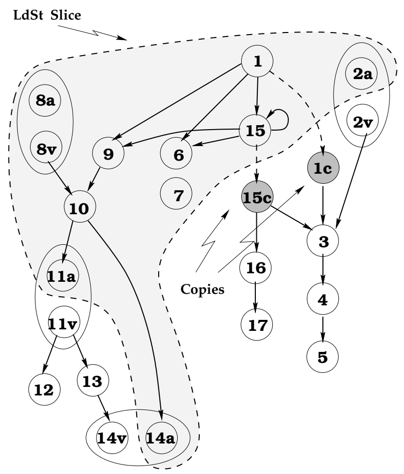

Figure 5: Partitioning with copy instructions.

o_{copy} : Overhead of a copy instruction
o_{dupl} : Overhead due to duplication
Then, $Profit = Benefit - Overhead$

It is beneficial to introduce copy and duplicate instructions only if $Profit \geq 0$. Thus, the problem reduces to determining S_{copy}, S_{dupl}, and S_c that maximize $Profit$. To compute $Profit$, n_B, the execution count of basic block B, needs to be estimated for all basic blocks. In this study, we obtained n_B using *basic-block execution profiles*. For functions that are not covered by the profile, we used *probabilistic* estimates for execution counts. If p_B is the probability that basic block B executes and d_B is the loop nesting depth of B, then n_B is set to $p_B * 5^{d_B}$. p_B's are computed on the assumption that both directions of a branch are equally likely to be taken.

The copy and duplication overheads, o_{copy} and o_{dupl}, were determined empirically. We experimented with different values for these parameters and picked the values that provided the best results. For our benchmarks, we empirically found that o_{copy} between 3 and 6 and o_{dupl} between 1.5 and 3 yield the best results. Next, we discuss the heuristics the compiler uses to determine profitable S_{copy}, S_{dupl}, and S_c.

6.2 Copying versus duplication

The advanced partitioning algorithm can either duplicate an instruction or insert a copy instruction. Code duplication avoids inter-partition communication whereas copy instructions cause inter-partition communication. However, when a node u is duplicated, all of its parents should either be duplicated or copy instructions should be inserted for them. If the parents are duplicated too, then the duplication effect fans out along the backward slice. This can be avoided if the result of u is copied instead. Thus, the decision of duplicating u depends on whether its parents communicate with the FP$_a$ partition, and, if so, whether they are copied or duplicated. This decision can change with any change in the status of these parents. A simple heuristic is presented below that makes these decisions during a prepass of the partitioning algorithm. Let $u_1, \ldots, u_k$ be the parents of v (excluding v). Initially, $dupl\_cost(v) = \infty$ and $copying\_cost(v) = o_{copy} * n_{B(v)}$ for all nodes v. The duplication cost is iteratively computed as:

$$dupl\_cost(v) = o_{dupl} * n_{B(v)} + \sum_{i=1}^{k} min(copying\_cost(u_i), dupl\_cost(u_i))$$

v is duplicated only if $dupl\_cost \leq copying\_cost$. This heuristic is based on the assumption that if v is to be duplicated, then each

```
I1:       move     $16, $0
I1d:      move,c   $f2, $0
    $L5:
I2:       lw       $f4, regs_invalidated_by_call
I3:       sra,c    $f4, $f4, $f2
I4:       andi,c   $f4, $f4, 0x1
I5:       beq,c    $f4, $0, $L4
I6:       move     $4, $16
I7:       jal      delete_equiv_reg
I8:       lw       $3, reg_tick
I9:       sll      $2, $16, 2
I10:      addu     $2, $2, $3
I11:      lw       $f0, 0($2)
I12:      bltz,c   $f0, $L4
I13:      addu,c   $f0, $f0, 1
I14:      sw       $f0, 0($2)
    $L4:
I15:      addu     $16, $16, 1
I15d:     addu,c   $f2, $f2, 1
I16:      slt,c    $f4, $f2, 66
I17:      bne,c    $f4, $0, $L5
```

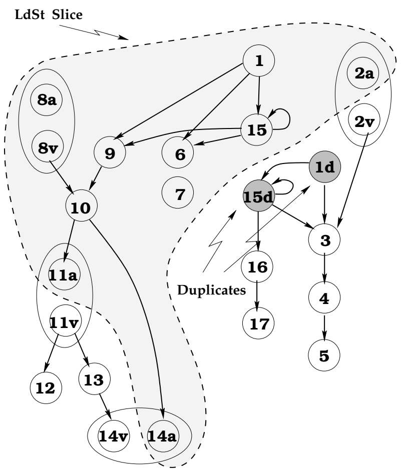

Figure 6: Partitioning with code duplication.

of the parents of v either needs to be copied or duplicated. This can yield non-optimal solutions because irrespective of whether v is duplicated or not, a parent might still be copied/duplicated and hence v need not be charged with the communication overhead for that parent.

If $o_{dupl} \geq o_{copy}$, no node will be duplicated. So, we require that $o_{dupl} < o_{copy}$. This is reasonable because code duplication can lead to inter-partition independence unlike copy instructions.

6.3 Algorithm for introducing copy instructions and duplication code

We are now ready to present an algorithm that identifies sites to introduce copy instructions and duplicate code to increase the size of the FP$_a$ partition. We restrict the algorithm to introduce copies only from INT to FP$_a$. This restriction keeps the decision procedure simple and also simplifies the actual insertion of copies and duplicates. As a result, if a node u is assigned to INT, $Backward\text{-}Slice(G, v)$ is also assigned to INT.

There are two distinct phases of the algorithm. Initially, the LdSt slice is assigned to the INT partition. In the first phase of the algorithm, the INT partition is expanded to include instructions that are not profitable for execution in the FP$_a$ subsystem. This expansion is done by analyzing the instructions on the *boundary* between the INT and FP$_a$ partitions. The boundary is made up of INT nodes some of whose children are not in INT. For each child of a boundary instruction, the algorithm checks if it is beneficial to retain the child instruction in the FP$_a$ subsystem. If not, the boundary is expanded to include the child in the INT partition.

During the second phase of the algorithm, copies and duplicates are tentatively introduced for instructions on the INT boundary. Then, for each connected component (in the undirected RDG) containing these copies and duplicates, $Profit$ is computed using the cost model presented earlier. Unprofitable ($Profit < 0$) components are assigned to INT during this phase. The algorithm is presented below.

```
0:  G = RDG for the function;
1:  Assign $LdSt\ slice$ to the INT partition;
2:  $Bdry$ = Boundary of the INT partition;
3:  $S$ = Non-INT children of nodes on $Bdry$;

       /******* Phase 1 ******/
4:  while ($S \neq \emptyset$) do {
5:    Pick a node $u$ from $S$;
6:    Let $P$ = FP$_a$ nodes in $Backward\text{-}Slice(G, u)$;
7:    Compute $loss$, the loss to FP$_a$ if $P$ is
        assigned to INT;
8:    if ($loss < 0$) then { /* Expand bdry */
9:      Move $P$ to INT and update $Bdry$;
10:     Add FP$_a$ children of nodes in $P$ to $S$;
11:   }
12:   elsif ($loss = 0$) then /*Defer decision*/
13:     Add FP$_a$ children of nodes in $P$ to $S$;
14:   Remove nodes in $P$ from $S$;
15: }

       /******* Phase 2 ******/
16: Use $Bdry$ to compute $S_{copy}$ and $S_{dupl}$;
17: Tentatively, introduce copies and duplicates
      in G for nodes in $S_{copy}$ and $S_{dupl}$;
18: Let $G_u$ = Undirected graph corresponding to G;
19: for each $u$ in $S_{copy}$ and $S_{dupl}$ do {
20:   Let $C$ = Connected component of $G_u$
        containing $u$;
21:   Compute $Profit$ for $C$;
22:   if ($Profit < 0$) {
23:     Assign $C$ to INT;
24:     Remove copies and duplicates from $C$;
25:   }
26: }
```

The primitive step in the first phase of the algorithm is the computation of loss if a FP$_a$ node u is moved to INT. As mentioned earlier, if u is moved to INT, all FP$_a$ nodes in $Backward\text{-}Slice(G, u)$ also get assigned to INT. $loss$ is this accumulated loss and is computed as follows.

$$\begin{aligned} P &= \text{Set of FP}_a \text{ nodes in } Backward\text{-}Slice(G,u) \\ &= \{v \mid v \in Backward\text{-}Slice(G,u);\ v \notin INT\} \\ Q &= \text{Boundary nodes that are parents of nodes in } P \\ &= \{v \mid v \in Bdry;\ \exists u \in P \text{ such that } v \text{ is } u\text{'s parent}\} \\ loss &= \textstyle\sum_{u \in P}[n_u + \alpha(u)] + \sum_{v \in Q} \delta(v) \end{aligned}$$

While computing $loss$, all nodes in P are considered to be in INT. Let us now examine the equation for $loss$ and account for each of the terms there. In the first summation, the first term(n_u) arises because u no longer executes in FP$_a$. If u has children in FP$_a$ and is assigned to INT, then u has to be copied if those children are to execute in FP$_a$. The second term, $\alpha(u)$, accounts for this overhead. The second summation accounts for changes in copying overheads of nodes in Q. For a node $v \in Q$, $\delta(v)$ is computed as follows. Since v is in INT[6], either:

- v does not have any child in FP$_a$. In this case, v no longer needs to be copied/duplicated when P gets moved to INT. So, $\delta(v) = -overhead$. Depending on whether v is copied or duplicated, $overhead$ is either $copying\_cost(v)$ or $duplication\_cost(v)$.
- v has at least one child in FP$_a$. In this case, irrespective of what happens to P, there is no change in the status of v. So, $\delta(v) = 0$.

In line 12, if $loss = 0$, this is considered to be insufficient information to assign P to INT. The decision is deferred until children of nodes in P are examined. It is possible to make a better decision when these children are examined because a bigger portion of the graph is analyzed.

At the end of Phase 1, the boundary of the INT partition will have stabilized. At this point, the sets S_{copy} and S_{dupl} can be computed by examining the nodes on the boundary. S_c is the set of nodes that are not in INT. For this partitioning, $Profit$ can be computed using the earlier specified cost model. However, this solution is not necessarily the best possible solution. It is possible to improve on this solution further. The first phase of the algorithm expands the INT boundary by making decisions about whether it is *unprofitable* to move a node u (and its backward slice) from FP$_a$ to INT. However, if u is retained in FP$_a$, it is not necessarily profitable. An example should make this clear.

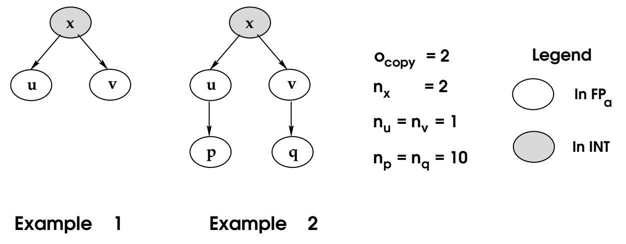

Figure 7: Examples for Phase 1 of the advanced scheme.

In the examples of Figure 7, suppose x has been assigned to INT. For both these examples, $Bdry = \{x\}, S = \{u, v\}$. In Phase 1 of the algorithm, nodes u and v are successively examined.

[6] all nodes in Q are in INT since they are in $Bdry$.

Consider node u. It can be verified that for both these examples, $loss > 0$ for both u and v. Hence, the boundary of the INT partition does not change. Let us now compute $Profit$ for both these examples.
For example 1, $S_c = \{u, v\}, S_{copy} = \{x\}, Profit = -2$.
For example 2, $S_c = \{u, v, p, q\}, S_{copy} = \{x\}, Profit = 18$.
The reason for this behavior is because Phase 1 uses only *local* information, i.e. while analyzing u, v is not examined. Further, only the immediate children of u are examined, not all its descendants.

Thus, it is necessary to refine the solution further. This is done during Phase 2. Initially, copies and duplicates are tentatively introduced for all nodes in S_{copy} and S_{dupl}. Note that introducing these copies and duplicates disconnects the graph at several places. For example, in Figure 5, the copies 1_c and 15_c introduce a new connected component $\{2_v, 3, 4, 5, 16, 17, 1_c, 15_c\}$ in the undirected RDG. Let G_u denote the undirected version of the RDG. After the copies and duplicates are introduced, the cost model is applied to each individual connected component of G_u containing a copy or a duplicate instruction. All unprofitable ($Profit < 0$) components are assigned to INT and the copies and duplicates in that component are eliminated.

6.4 Interaction with calling conventions

As discussed in Section 6.6, calling conventions require all integer-valued arguments and return values to be passed/returned in integer registers. This constrains the partitioning algorithms in how much code can be offloaded to FP$_a$. One solution to get around this restriction is to perform code partitioning by ignoring calling convention restrictions and introducing copies where necessary. However, it is important to evaluate the benefit in introducing such copies.

Let us consider how call arguments (both actual as well as formal) are handled. Return value nodes are handled just like call argument nodes. Within a function, if the call arguments that are passed in (formal parameters) are required in FP$_a$, these values have to be copied to FP$_a$. Similarly, if the computation of the call arguments (actual parameters) takes place in FP$_a$, then these values have to be copied to INT at call sites. This is the only case when copies are introduced from FP$_a$ to INT.

For each formal parameter, a dummy node is introduced representing the definition of the parameter. All these dummy nodes are pre-assigned to INT. Once this is done, the partitioning algorithm previously presented evaluates the benefit of introducing copies for these dummy nodes automatically.

Actual parameter nodes are handled differently. Initially, all computation of actual parameter values is assigned to FP$_a$. During Phase 1, the equation for $loss$ is extended to account for copies required for actual parameter nodes. If a node $u \in P$ is an actual parameter node, then the first term of the equation for this node changes from $[n_u + \alpha(u)]$ to $-copying\_cost(u)$ because it is beneficial to move an actual parameter node to INT since a copy is no longer needed for it. Further, during phase 2, when the cost model is applied to each connected component (line 21 of the algorithm), the cost of introducing a copy from FP$_a$ to INT is considered an extra overhead in computing $Profit$.

6.5 Optimality Issues

The advanced algorithm can yield non-optimal solutions for a number of reasons. First, as discussed earlier, the decisions for copying a node or duplicating it are made non-optimally. Second, during

Phase 1, better solutions might be obtained by using global information. Though the algorithm can yield non-optimal solutions, we found that these heuristics successfully yield bigger FP$_a$ partitions using very few copy and duplicate instructions. The number of extra dynamic instructions executed due to copies and duplicates is less than 1% for most benchmarks. Further, the increase in static code size is also negligible.

6.6 Limitations of the partitioning schemes

A major underlying assumption in our schemes is that the floating-point subsystem of superscalar architectures can be augmented to support integer operations *without* affecting the *latency* of these operations, i.e. a single cycle add in INT will still take a single cycle in FP$_a$. If the hardware does not support any single-cycle latency operations in the floating-point subsystem, then single-cycle result paths will have to be added when single-cycle integer operations are supported. This potentially introduces an additional hardware cost. An alternative would be to modify the algorithms to account for the increased latency of integer operations when they are moved to FP$_a$. However, in such a case, the resulting performance improvements would be smaller.

Both the partitioning algorithms presented earlier greedily assign as much computation as possible to FP$_a$ without considering whether this would underutilize the INT unit. The rationale behind this decision was discussed in Section 4. The primary assumption there was that most integer codes perform significant memory access. However, for functions that perform very little or no memory access, this strategy can backfire. For example, the *ran* function from the *compress* benchmark, that generates random numbers does not access memory at all. As a result, our partitioning schemes move the entire function from INT to FP$_a$! However, for integer programs, we expect this kind of behavior to be rare. Nevertheless, the algorithms could be improved to consider load balance while performing code partitioning.

The cost model presented does not take into account the availability of extra floating-point registers for register allocation. When some of the integer code is offloaded to FP$_a$, this can reduce the register pressure on the integer register file and can thus reduce register spills/refills. However, at the same time, there might be an increase in register saves/restores across calls. In our simulations, we found that there was a decrease in loads by 3.7% for *go*. At the other extreme, there was an increase in loads of about 2.6% for *gcc*. Considering that most spills/refills and saves/restores hit in the cache, these small changes in loads/stores might not be a significant performance factor. However, if these effects are accounted for in the cost model, better results might be obtained.

In the current schemes, all integer-valued arguments are passed in integer registers as required by calling conventions. If the partitioning algorithms deem it profitable, call arguments are copied to/from FP$_a$. By performing interprocedural analysis, it might be possible to reduce some of the copy overheads across calls by passing integer arguments in floating-point registers.

7 Performance Results

In this section, we first present our evaluation methodology. We then present results for the effectiveness of the two partitioning schemes and the net performance improvement over a conventional microarchitecture.

7.1 Evaluation Methodology

We used *gcc-2.7.1* as the base compiler for our work. The compiler was modified to generate code for the extended SimpleScalar [7] instruction set which is based on the MIPS instruction set. The SimpleScalar instruction set was extended by using new opcodes to encode integer instructions executing in the floating-point subsystem. We used 22 new opcodes for our study. All integer operations except integer multiply and divide are supported in the floating-point subsystem. Our simulation study shows that except for *ijpeg* which has about 3% of integer multiplications and divisions, all other benchmarks have negligible amount of these operations. Further, this keeps the hardware cost to a minimum since integer multiply and divide operations tend to be expensive (in terms of die area) to implement.

Code partitioning is performed on the intermediate representation of the program after all the initial machine-independent optimizations [2] are complete. Register allocation is performed after code partitioning. Operands of instructions assigned to the FP$_a$ partition are allocated floating-point registers.

We used a cycle-based timing simulator derived from the SimpleScalar tool set [7]. The timing simulator models both a conventional microarchitecture as well as a microarchitecture with the augmented floating-point subsystem. Both microarchitectures are identical in all other respects. The machine parameters we used in our simulations are presented in Table 1.

We used programs from the SPECint95 benchmark suite to conduct our evaluation. The benchmarks and the inputs used are given in Table 2. For the conventional microarchitecture, the benchmark programs are compiled by the base compiler (unmodified *gcc-2.7.1*). All the benchmarks are compiled at the *-O3* optimization level which enables common subexpression elimination, loop invariant removal, and jump optimizations among others. All the benchmarks were simulated to completion.

7.2 Percentage of computation offloaded to FP$_a$

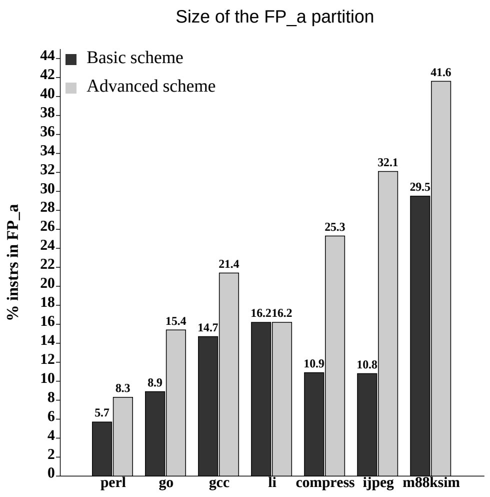

Figure 8: Size of the FP$_a$ partition.

Figure 8 shows the percentage of total dynamic instructions offloaded by the compiler for each of the benchmark programs. The graph shows the size of the FP$_a$ partition for both the basic and the

Parameter	4-way	8-way
Fetch width	any 4 instructions	any 8 instructions
I-cache	64KB, 2-way set-associative 128 byte lines,1 cycle hit time,6 cycle miss penalty	
Branch Predictor	McFarling's gshare [25] with 32K 2-bit counters, 15 bit global history Unconditional control flow instructions predicted perfectly	
Decode/Rename width	any 4 instructions	any 8 instructions
Issue window size	16 int/16 fp	32 int/32 fp
Max. in-flight instructions	32	64
Retire width	4	8
Functional Units	2 Int + 2 Fp units	4 Int + 4 Fp units
Functional Unit Latency	6 cycle mul, 12 cycle div, 1 cycle for rest	
Issue Mechanism	up to 4 ops/cycle	up to 8 ops/cycle
	out-of-order issue loads may execute when prior store addresses are known	
Physical Registers	48 int/48 fp	80 int/80 fp
D-cache	32KB, 2-way set-associative, write-back, write-allocate 32 byte lines,1 cycle hit time,6 cycle miss penalty	
	one load/store port	two load/store ports

Table 1: Machine parameters.

Benchmark	*compress*	*li*	*gcc*	*m88ksim*	*go*	*ijpeg*	*perl*
Input	test.in	browse.lsp	stmt.i	ctl.raw, dhry.big	2stone9.in	vigo.ppm	scrabbl.pl

Table 2: Benchmark programs.

advanced partitioning schemes. Because all the benchmark programs are integer programs that execute negligible floating-point operations, the bars in the graph correspond to the amount of integer computation that the compiler is able to identify and offload to the FP$_a$ subsystem. Overall, the compiler is successful in offloading a sizable fraction of the total computation to the FP$_a$ subsystem. In the case of *ijpeg*, *m88ksim*, *compress*, and *gcc*, more than 20% of the total computation is supported in the FP$_a$ subsystem. The graph also shows that the advanced partitioning scheme generates bigger partitions than the basic scheme for all benchmarks. For *go* and *compress*, the partitions generated by the advanced partitioning scheme are almost twice the size of those generated by the basic scheme. *Ijpeg* benefits the most from the advanced scheme: the FP$_a$ computation increases from 10.7% to 32.1%. However, for *li*, the advanced scheme does not perform better than the basic scheme because *li* is call intensive and has a number of small functions.

While the advanced partitioning scheme might be able to offload more computation, the percentages must be judged in conjunction with the change in the instruction cache performance and the total number of instructions executed due to the extra instructions introduced. Hence, we studied the overhead introduced by the advanced partitioning scheme. For all the benchmarks, we found the change in static code size to be *negligible*. As a result, there was very little change in instruction cache hit rates for all the benchmarks. The increase in the number of dynamic instructions executed is also small. The maximum increase is 4% for *compress*. Copies account for 3.4% and 0.6% is due to duplicates. Overall, these results show that the advanced partitioning scheme is successful in increasing the FP$_a$ partition sizes without introducing a lot of overhead.

7.3 Performance improvements on a 4-way issue machine

Figure 9 shows the performance improvements obtained by the proposed microarchitecture over a conventional microarchitecture for a 4-way issue (2 int + 2 fp) machine. Performance improvements

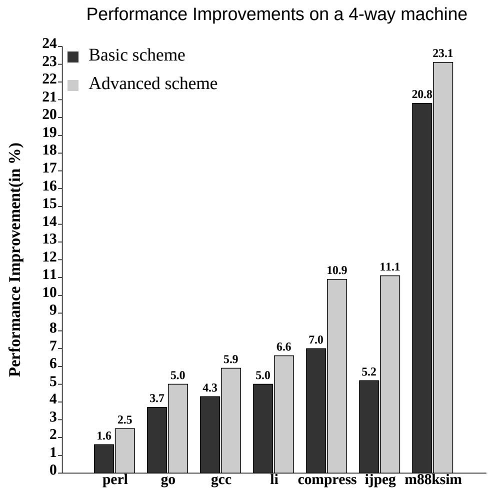

Figure 9: Speedups on a 4-way machine.

due to both the basic and advanced partitioning schemes are presented. For *m88ksim*, *ijpeg*, and *compress*, improvements over 10% are achieved with the advanced partitioning scheme. In the case of *m88ksim*, an impressive improvement of 23% is achieved with the advanced partitioning scheme. Overall, with the advanced partitioning scheme, the hardware is capable of providing modest to impressive performance improvements.

As expected, performance increases as more instructions are offloaded to the FP$_a$ subsystem. However, the performance does not directly reflect the size of the FP$_a$ partitions for two reasons. First, the critical path of execution may not be affected by partitioning. For example, in programs that are dominated by loads and stores, performance is largely determined by the available cache bandwidth. Second, as discussed in Section 6.6, INT resources could be underutilized due to load imbalance. This translates to lower performance than expected. For example, for *m88ksim*, with the advanced scheme, the INT subsystem is idle 12.4% of the cycles in which the FP$_a$ subsystem is executing one or more instructions. This partly explains why performance only improves by about 2.6% even though the size of the partition increases by 12%. This problem also occurs to a lesser degree in *ijpeg*.

The graph also shows that except for *li* and *m88ksim*, the advanced partitioning scheme yields much better performance improvements than the basic partitioning scheme. In the case of *li*, the increase in the size of the FP$_a$ partition is very small. For *m88ksim*, load imbalance seems to be the problem as mentioned earlier.

7.4 Performance improvements on a 8-way issue machine

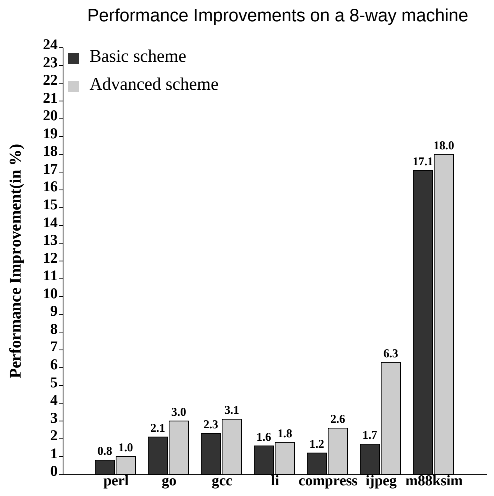

Figure 10: Speedups on a 8-way machine.

Figure 10 shows the performance improvements on a 8-way issue (4 int + 4 fp) machine. As expected, the improvements are much smaller because the issue width of INT gets within the range of the average parallelism in a program. So, the extra issue bandwidth of the FP$_a$ subsystem is not exploited as much. Programs like m88ksim, which have enough parallelism are able to exploit the presence of a bigger instruction window and wider issue and execution bandwidth.

7.5 Applicability to floating-point programs

We also experimented with the partitioning schemes on some floating-point programs from the SPEC92 and SPEC95 benchmark suites. For floating-point programs, there is not as much room for obtaining performance improvements since the floating-point subsystem is utilized for performing floating-point computation. Since our algorithms do not attempt to balance load across the partitions, we expected slowdowns when these algorithms were applied to floating-point programs because the offloaded integer instructions would compete with the floating-point instructions for issue and execution resources. However, for all but one of the benchmarks that we tried, there was negligible change (slowdown/speedup) in running time because the algorithms attempt to offload only branch and store-value slices to FP$_a$. For floating-point programs, most of the store-value slices and some of the branch slices are already in FP$_a$. In these programs, integer computation offloaded to FP$_a$ is very small. However, for one of the SPEC92 benchmarks, *ear*, as much as 18% of integer (branch and store-value) computation was offloaded to FP$_a$ that resulted in 18% speedup on a 4-way issue machine. Thus, it appears that the algorithms presented here can be applied to floating-point programs as well without hurting performance and even improving performance in certain cases. It might be possible to further improve performance on these programs if the algorithms are improved to account for load balance across the partitions.

8 Conclusions

In current superscalar processors, all floating-point resources are idle during the execution of integer programs. This problem can be alleviated if the floating-point subsystem is augmented to execute integer instructions and the compiler can identify integer instructions that can execute in such an augmented floating-point (FP$_a$) subsystem. The required modifications are minor and the resultant microarchitecture stays similar to a conventional microarchitecture. However, compiler support is required to identify integer computation that can execute in the FP$_a$ subsystem. We presented and evaluated two code partitioning schemes to do this and evaluated them. Our evaluation shows two things. First, for our benchmarks, the compiler is able to offload a significant fraction, from 9% to 41%, of the total computation in integer programs to the FP$_a$ subsystem. Second, the partitions identified by the compiler can speed up programs by 3% to 23% over a conventional 4-way issue superscalar processor. For three of the benchmarks, *compress*, *ijpeg*, and *m88ksim*, the performance improved from 11% to 23%. Hence, we believe that with minimal hardware support, idle floating-point resources on current superscalar processors can be profitably exploited by the compiler to speed up integer programs.

9 Acknowledgements

This work was supported in part by NSF Grants MIP-9505853 and MIP-9307830 and by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U. S. Army Intelligence Center and Fort Huachuca, or the U.S. Government. We would like to thank Prof. Charles Fischer, Prof. Susan Horwitz, Satish Chandra, Amir Roth, Glenn Ammons, Yiannakis Sazeides at UW-Madison and Anurag Acharya at UCSB for providing us with useful comments on earlier drafts of this paper. Thanks are due to Sanjiv Banerjia for providing us with chapters of his thesis and pointing us to other related work. We would also like to thank the anonymous referees for their comments and suggestions that helped improve the content and presentation of the paper.

References

[1] Alex Peleg, Sam Wilkie, and Uri Weiser. How Intel Built MMX Technology. *Communications of the ACM*, 40(1):25–38, January 1997.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques and Tools*. Addison Wesley, 1988.

[3] Andrea Capitanio, Nikil Dutt, and Alexandru Nicolau. Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 292–300, December 1992.

[4] Ashok Kumar. The HP-PA8000 RISC CPU: A High Performance Out-of-Order Processor. In *Proceedings of the Hot Chips VIII*, pages 9–20, August 1996.

[5] Sanjeev Banerjia. *Instruction Scheduling And Fetch Mechanisms For Clustered VLIW Processors*. PhD thesis, Dept. of Electrical and Computer Engineering, North Carolina State University, 1998.

[6] Digital Equipment Corporation. *Alpha Architecture Handbook, Version 3*, October 1996.

[7] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-96-1308 (Available from http://www.cs.wisc.edu/trs.html), University of Wisconsin-Madison, July 1996.

[8] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace Processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, December 1997.

[9] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[10] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[11] Giuseppe Desoli. Instruction Assignment For Clustered VLIW DSP Compilers: A New Approach. Technical Report HPL-98-13, HP Labs, January 1998.

[12] Gregory A. Kemp and Manoj Franklin. PEWS: A Decentralized Dynamic Scheduler for ILP Processing. In *Proceedings of the International Conference on Parallel Processing*, volume I, pages 239–246, 1996.

[13] J.A.Fisher. Very Long Instruction Word Architectures and ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, June 1983.

[14] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, Yale University, 1985.

[15] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. The Multicluster Architecture: Reducing Cycle Time Through Partitioning. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 149–159, December 1997.

[16] Jim Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution, October 1996. *9th Annual Microprocessor Forum*, San Jose, California.

[17] Linley Gwennap. MIPS R10000 Uses Decoupled Architecture. *Microprocessor Report*, 8(14), October 1994.

[18] Linley Gwennap. UltraSparc Unleashes SPARC Performance. *Microprocessor Report*, 8(13), October 1994.

[19] Linley Gwennap. UltraSparc Adds Multimedia Instructions. *Microprocessor Report*, 8(16), December 1995.

[20] Linley Gwennap. Intel's MMX Speeds Multimedia. *Microprocessor Report*, 10(3), March 1996.

[21] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[22] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, October 1994.

[23] P.Geoffrey Lowney, Stefan Freudenberger, Thomas Karzes, W.D.Lichtenstein, Robert P. Nix, John S. O'Donnel, and John C.Ruttenberg. The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing*, 7(1):51–142, May 1993.

[24] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. *IEEE Transactions on Computers*, 37(8), August 1988.

[25] Scott McFarling. Combining Branch Predictors. Technical Report DEC WRL Technical Note TN-36, DEC Western Research Laboratory, 1993.

[26] Subbarao Palacharla and J. E. Smith. Decoupling Integer Execution in Superscalar Processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 285–290, November 1995.

[27] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[28] Zarka Cvetanovic and Dileep Bhandarkar. Characterization of Alpha AXP Performance Using TP and SPEC Workloads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.