Decoupling Integer Execution in Superscalar Processors

Subbarao Palacharla — Computer Sciences Department, University of Wisconsin-Madison, Madison, WI 53706, subbarao@cs.wisc.edu

J. E. Smith — Department of Electrical and Computer Engineering, University of Wisconsin-Madison, Madison, WI 53706, jes@ece.wisc.edu

Abstract

We propose that processor hardware can be used more effectively if floating-point units are augmented to perform simple integer operations. Existing floating-point registers and datapaths are used to support these integer operations. Some integer instructions, those not used for computing addresses and accessing memory, can then be *off-loaded* to the floating-point units. Consequently, these integer instructions are decoupled from memory accessing, and additional instruction bandwidth is available for integer programs.

This paper reports the results of a preliminary study of integer benchmark programs compiled for the SPARC architecture. The results indicate that between 10% and 39% of the instructions in the integer benchmarks can be executed in the augmented floating-point units. Furthermore, these instructions are all simple add, subtract and logical instructions.

1 Introduction

Over the past few years processor microarchitectures have converged to a decoupled implementation style. Figure 1 shows the microarchitecture of a typical superscalar processor [1, 5, 2] using this decoupled style. It comprises a fetch unit that feeds instructions to integer and floating-point subsystems. The integer subsystem is made up of a number of load/store, branch and ALU units operating out of the integer register file. The floating-point subsystem is similar to the integer subsystem except that it does not contain any load/store units, and it operates on floating-point operands in the floating-point registers. Buffers, in the form of reservation stations or queues, are used to decouple the instruction streams going to the functional units.

This implementation style was identified and discussed in the Decoupled Access/Execute work described in [7, 3, 6]. At least in part, this work had its roots in the early Control Data Corporation and Cray Research style of architectures where one set of functional units and registers is used for addressing and another set is used for scalar computation, *both integer and floating point*. For example, the CRAY-1 [4] uses A registers for address computation and S registers for scalar computation, both integer and floating point.

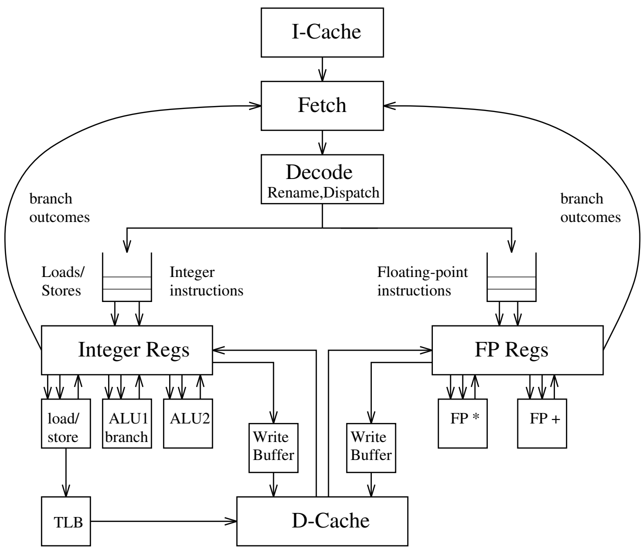

Figure 1: Conventional microarchitecture

This figure presents the typical microarchitecture employed by most current microprocessors. The floating-point (FP) units are idle while the processor is executing integer-intensive portions of a program.

Although many modern microprocessors have these general characteristics, their evolution has been slightly different. Early microprocessors were typically implemented as an integer processor with an attached floating-point subsystem in the form of a co-processor. The floating-point subsystem was later absorbed on-chip, resulting in a decoupled implementation. This style of decoupling uses one set of units and registers for addressing and integer computation and the other set for floating-point computation. The drawback of this style is that when an integer program or integer-intensive portion of a floating-point program is executing, the floating-point registers, units, datapaths, and instruction issue logic are idle.

However, if we employ the more general decoupled style, both floating point and some integer operations are executed in a computation unit. Then, some of the resources that would otherwise be dedicated to floating point can be utilized in non-floating point applications. This may lead to a duplication of some of the functional units, e.g. integer add/subtract and logical. However, these units are relatively inexpensive. Their only potential significant cost is in the datapaths, rather than in the gates themselves. The datapath cost can be eliminated by sharing paths with the floating-point units; i.e. by embedding the integer functions in the floating-point units.

Our primary motivation is to study the feasibility of the proposed microarchitecture consisting of a Load/Store (*LdSt*) subsystem and a Computation (*Comp*) subsystem. The LdSt subsystem executes instructions that are involved in effective address calculation and memory access. The Comp subsystem consists of floating-point functional units that are augmented to operate on integer operands as well. When running integer programs, the Comp unit supports some of the integer computation. To avoid adding datapaths, all data communication between the two subsystems takes place through loads and stores that already exist in the original integer program.

In this paper we analyze integer benchmark programs to determine the fraction of dynamic instructions that can be executed in integer-augmented floating-point units under the constraint that no additional datapaths are added. This is done by first studying dynamic instruction traces to find the instructions that are

- used to generate addresses and access memory
- used to evaluate branch conditions and perform branches
- used purely for computation, i.e. that fall into neither of the first two categories.

This information is then used to mark the static instructions as to whether they should be executed in the LdSt unit or the Comp unit. Finally, execution statistics from the static instructions are used to produce the fractions of dynamic instructions executed in the LdSt and Comp units.

The rest of the paper is divided into five sections. Section 2 defines some terminology and presents our methodology. Section 3 discusses the fundamental division of program execution into memory access, control and computation functions. Section 4 describes the way the proposed microarchitecture uses idle floating-point units while executing integer code and presents results. Section 5 discusses future work. Finally, we draw conclusions in Section 6.

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

     1  sethi %hi(a),%o3
     2  or %o3,%lo(a),%o3  ;load addr(a[0])
     3  sethi %hi(b),%o0
     4  or %o0,%lo(b),%o2  ;load addr(b[0])
     5  sethi %hi(c),%o0
     6  or %o0,%lo(c),%o1  ;load addr(c[0])
     7  sethi %hi(ck),%o0
     8  ldd [%o0+%lo(ck)],%f6  ;load ck
     9  mov 0,%o0          ;i=0
    10  ldd [%o0+%o1],%f2     ;load c[i]
L0: 11  fmuld %f6,%f2,%f2     ;ck*c[i]
    12  ldd [%o0+%o2],%f4     ;load b[i]
    13  faddd %f2,%f4,%f2     ;b[i]+ck*c[i]
    14  std %f2,[%o0+%o3]     ;assign to a[i]
    15  add %o0,8,%o0       ;i++
    16  subcc %o0,792,%g0     ;i < 100?
    17  ble,a L0
    18  ldd[%o0,%o1],%f2      ;load c[i]
```

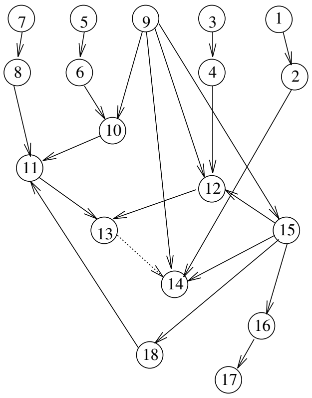

Figure 2: Static dependence graph

This figure shows the static dependence graph for the simple C loop shown at the top left. The corresponding assembly code is shown below the loop. There is an edge from instruction i to instruction j if instruction i produces a value that is used by instruction j. Memory and store value dependencies are depicted using special (dotted) edges.

2 Methodology

The analysis performed in this paper involves constructing various *slices* of a program. A slice [10] of a program with respect to a value *v*, denoted by Slice(P,*v*), is defined to be the subset of the program that computes the value *v*. This is illustrated by the simple example shown below.

```
a = b + c;           a = b + c;
e = b + g;
d = a * g;           d = a * g;
f = d + a;           f = d + a;

Program P            Slice(P,f)
```

To efficiently determine slices of a program we construct a data structure that we call the *static dependence graph* for the program. The static dependence graph for a program consists of N vertices, where N is the number of static instructions in the program binary. The graph has an edge from vertex v_i to vertex v_j if instruction i produces a value that is consumed by instruction j. In other words, there is an edge from vertex v_i to vertex v_j if there is a true dependence between instructions i and j. Register dependencies are shown using solid edges. Memory dependencies and store value [1] dependencies are shown using dotted edges. Figure 2 shows the static dependence graph for a simple loop.

We build the static dependence graph by scanning a dynamic instruction trace. If a data dependence is found in the dynamic trace, an edge connecting the corresponding instructions is added to the graph.

[1] value being stored

While this method is expedient, there are two caveats. First, the resulting program slices are only valid for the given input data. Different input data sets could result in different program slices. This could happen if the distinct data sets cause different dependencies to occur at run-time. However, we believe that in most cases dependences will not change appreciably if the data set is changed. Second, in a real implementation, the compiler would be responsible for forming the static dependence graph based on compile-time information. Hence, it would likely be more conservative than the method we use which takes advantage of run-time information. Here, we believe that most of the information the compiler needs will be available within individual basic blocks. However, only additional study of compiler methods (which we plan to carry out) will resolve this important issue.

Once we have constructed the static dependence graph, a slice is extracted by traversing the static dependence graph. To find the slice of the program with respect to a value v, we start at the node that created v and find the set of nodes from which v can be reached by traversing one or more edges. This set of predecessor nodes is the slice with respect to v. For example, in Figure 2, the slice of the program with respect to %f2 (floating-point register 2) produced by instruction 11 is $\{11,8,7,10,6,5,9\}$. The number of instructions executed by the slice is computed by adding the dynamic count of all the instructions in the slice.

We used *Shade* to generate dynamic instruction traces. Shade [9] is an evaluation tool developed at Sun Microsystems that aids instruction set simulation. We used five integer benchmarks from the SPEC suite [8]. All the benchmarks were compiled using `gcc` (version 2.5.8) with the `-O3` option. We analyzed complete program runs. Columns 2 through 4 of Table 1 give the total number of instructions, the number of load and store instructions, and the number of branch instructions executed by each of the benchmark programs.

3 Program slices

For a given program there are three functions that we would like to separate: memory access, control and execution. To achieve this separation, we decompose each benchmark program into three slices called the *load/store*, *branch* and *compute* slices. These slices are defined as follows:

- **Load/store slice**: The load/store slice consists of all the instructions of the program that are involved in the calculation of the base registers of all the load and store instructions in the program and the load and store instructions themselves. The load/store program does not include control flow (branch) instructions. Note that we are assuming a load/store architecture here.

- **Branch slice**: The branch slice contains all the instructions of the original program that are required to compute the predicates of conditional branches and all the control instructions themselves. In an architecture with condition codes, e.g. the SPARC, the branch slice includes all instructions that are involved in computing the condition codes and all the control instructions.

- **Compute slice**: The compute slice consists of instructions used purely for computing data values that will be stored to memory; i.e. they do not contribute to the determination of control flow or memory access. The compute slice can be most easily be defined to be whatever remains of the original program after the load/store and branch slices have been extracted. Strictly speaking, the compute slice does not adhere to our definition of slice presented earlier. However, we use the term for the sake of maintaining uniformity.

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

L B C
1 0 0        1  sethi %hi(a),%o3
1 0 0        2  or %o3,%lo(a),%o3  ;load addr(a[0])
1 0 0        3  sethi %hi(b),%o0
1 0 0        4  or %o0,%lo(b),%o2  ;load addr(b[0])
1 0 0        5  sethi %hi(c),%o0
1 0 0        6  or %o0,%lo(c),%o1  ;load addr(c[0])
1 0 0        7  sethi %hi(ck),%o0
1 0 0        8  ldd [%o0+%lo(ck)],%f6  ;load ck
1 1 0        9  mov 0,%o0          ;i=0
1 0 0       10  ldd [%o0+%o1],%f2    ;load c[i]
0 0 1   L1: 11  fmuld %f6,%f2,%f2    ;ck*c[i]
1 0 0       12  ldd [%o0+%o2],%f4    ;load b[i]
0 0 1       13  faddd %f2,%f4,%f2    ;b[i]+ck*c[i]
1 0 0       14  std %f2,[%o0+%o3]    ;assign to a[i]
1 1 0       15  add %o0,8,%o0        ;i++
0 1 0       16  subcc %o0,792,%g0    ;i < 100?
0 1 0       17  ble,a L1
1 0 0       18  ldd[%o0,%o1],%f2     ;load c[i]
```

Figure 3: Example slices

This figure presents the load/store, branch and compute slices for the loop shown at the top. The L, B and C columns stand for load/store, branch and compute slice respectively. An instruction belongs to a particular slice if the bit in the corresponding column is set.

Each of the load/store, branch and compute slices is, by definition, a subset of the original program. Figure 3 presents the SPARC assembly code for a small loop and the associated load/store, branch and compute slices. Note that there might be some instructions that are common to the load/store and the branch slice. For example, for the loop in Figure 3, the instructions manipulating the loop counter (maintained in register %o0) are present in both the load/store and branch slices.

Even though generating the load/store, branch and compute slices is an intermediate step for extracting computation that can be off-loaded to the floating-point subsystem, we believe that these slices are fundamental to a program and deserve further study. Hence, we quantified the relative sizes of these slices by measuring the number of dynamic instructions executed by each of them. The last three columns of Table 1 presents these statistics. For example, in the case of *compress*, the load/store slice executes 36.7M instructions or 51.1% of the total instructions executed by the benchmark. From the table we see that, for most benchmarks, the number of instructions executed by the load/store slice is less than or close to two times the number of load and store instructions executed. This implies that on an average each dynamic load/store instruction depends on at most one other instruction that aids in computing the effective address. Another observation that can be made from the table is the large size of the branch slice. For four of the benchmarks the branch slice size is well over 90% of the total instruction count. Here, we are not certain if this is a characteristic of our benchmarks (integer programs with frequent branches) or if this is an artifact of our definition of the branch slice. With our definition of the branch slice, a branch at the end of the program that tests the result of the entire computation, say in the `printf` routine that actually prints the result, can easily result in the whole program being included in the branch slice.

4 Harnessing idle floating-point units

In order to utilize the idle floating-point units while executing integer programs, we must assign some of the computation in the program to the Comp subsystem in the proposed microarchitecture. This requires partitioning the static program into two sets of instructions: the *LdSt program* that executes in the LdSt subsystem and the *Comp program* that executes in the Comp subsystem. Since the proposed microarchitecture supports load/store instructions only in the LdSt subsystem and there is no support for copying between the register files, the load/store instructions and the instructions involved in the effective address computation are assigned to the LdSt program. In other words, the LdSt program is initialized to be the load/store slice.

The rest of the instructions are assigned as follows. First, we consider that part of the branch slice that is not included in the load/store slice and identify computation in it that can be assigned to the Comp program. Specific eligibility conditions for assigning instructions to the Comp program are discussed below. Next, any remaining instructions in the branch slice are added to the LdSt program. Finally, we do a similar analysis on the compute slice; first finding any constituent computation that is eligible for assignment to the Comp program and adding the remaining compute slice instructions to the LdSt program.

Recall that the branch slice and compute slice produce branch outcomes and store values, respectively. The results of the branch computations are sent to the fetch unit where they are used to validate the predicted outcomes. The store values from the store value computations are deposited in the write buffer where they merge with the store addresses generated by the LdSt subsystem. Note that these two functions (sending branch outcomes to the fetch unit and depositing store values in the write buffer) are already implemented in the floating-point subsystems of existing microarchitectures.

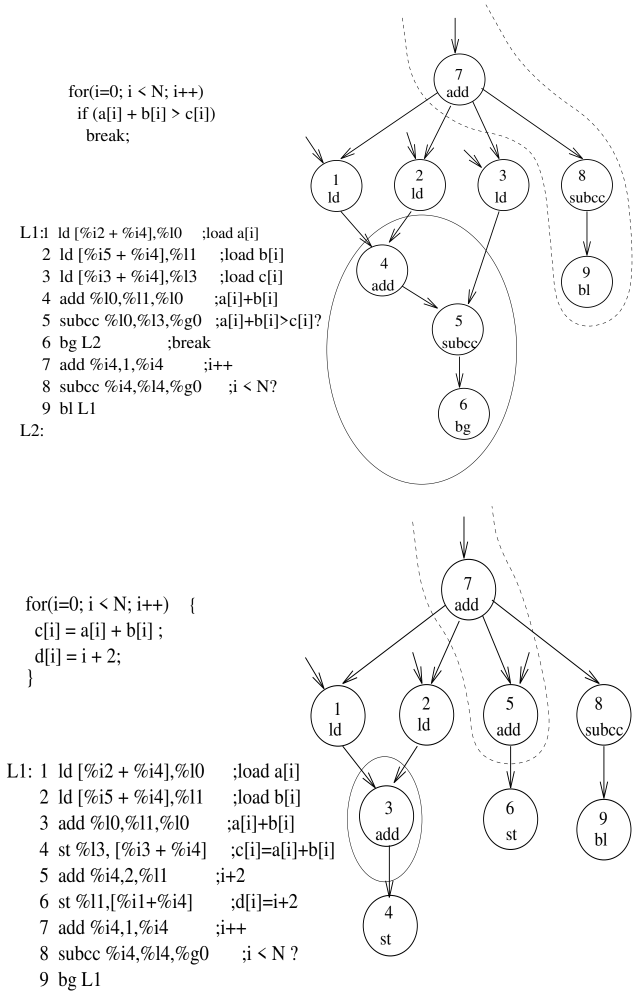

Figure 4: Branch and store value computations

The top graph shows two different kinds of branch computations. The one enclosed in the solid ellipse can be executed in the Comp subsystem. However, the slice of the branch represented by node 9 in the graph, demarcated by the dashed line, cannot be assigned to the Comp subsystem since it includes node 7 (part of the load/store slice) which is assigned to the LdSt subsystem. The bottom graph presents two different kinds of store value computations. The one enclosed in the solid ellipse can be assigned to the Comp subsystem. However, the one demarcated by the dashed curve cannot be assigned to the Comp subsystem as it includes node 7 (part of the load/store slice) which is assigned to the LdSt subsystem.

A particular instruction *I* can be assigned to the Comp program if it satisfies the following requirements:

- *I* should not read any register that is written by a non-

Benchmark	Total	Memory Ops	Control Ops	Load/Store slice	Branch slice	Compute slice
compress	71.8	19.8(27.6%)	14.4(20.0%)	36.7(51.1%)	62.2(86.6%)	2.4(3.4%)
espresso	683.3	188.8(27.6%)	142.9(20.9%)	323.8(47.4%)	656.8(96.1%)	15.9(2.3%)
eqntott	1705.6	590.5(34.6%)	342.3(20.1%)	793.5(46.5%)	1682.4(98.6%)	9.6(0.6%)
gcc	40.2	10.4(26.0%)	7.6(18.9%)	20.8(51.8%)	38.0(94.6%)	1.5(3.8%)
sc	89.2	23.2(26.0%)	18.1(20.3%)	40.9(45.8%)	83.8(93.9%)	3.7(4.2%)

Table 1: Benchmark slice statistics

This table presents the various slice statistics for the benchmarks used in this paper. All the benchmarks are from the SPEC integer suite. The Memory Ops and Control Ops columns show the number of load/store instructions and control instructions (branches, calls, etc.) executed. The number of instructions executed by the various slices are shown in the last three columns. All counts are in millions. The percentages are with respect to the total number of instructions executed by a benchmark.

load instruction already assigned to the LdSt program. This is enforced to avoid the need to communicate the register value from the LdSt register file to the Comp register file.

- Any load instructions that supply data to *I* must not supply data to an instruction assigned to the LdSt program. Once again, this restriction is placed to avoid the need to copy between the register files.
- Register values produced by *I* must not be used by an instruction assigned to the LdSt subsystem.

Figure 4 presents examples to clarify the assignment criteria. For example, in Figure 4, consider the computation enclosed in the solid ellipse (instructions 4, 5 and 6) in the top graph. This computation uses the results of load instructions 1, 2 and 3 as input and calculates a branch outcome (result of instruction 6). The values produced by instructions 4 and 5 are not used by any instruction external to the ellipse. Also, the load instructions do not feed any instruction that is not part of the enclosed computation. Hence, the computation enclosed in the solid ellipse can be executed in the Comp subsystem. Conversely, the computation demarcated by the dashed line (instructions 7, 8 and 9) cannot be assigned to the Comp subsystem since the result of the add instruction (instruction 7) is used by the load instructions 1, 2 and 3 which are part of the LdSt program. Similarly, the bottom graph of Figure 4 shows two types of store value computations - one that can be part of the Comp program and one that cannot be assigned to the Comp subsystem.

Using the above rules we identified the fraction of instructions that can potentially be assigned to the Comp subsystem. Table 2 presents this data. The second and third columns give the dynamic counts of Comp instructions extracted from branch and store value computation respectively. The last column shows the total fraction of dynamic instructions that can be supported in the Comp subsystem. From the table we can see that in the case of *eqntott*, *compress* and *sc* as much as 39%, 25% and 19% respectively of the total computation can be executed in the Comp subsystem. For the other two programs, *espresso* and

Name	branch computation	store value computation	Total Comp computation
compress	14.3(20.0%)	3.4(4.8%)	17.7(24.8%)
espresso	50.2(7.4%)	24.7(3.6%)	74.9(11.0%)
eqntott	655.9(38.5%)	6.4(0.4%)	662.3(38.9%)
gcc	3.7(9.2%)	0.4(1.0%)	4.1(10.2%)
sc	15.2(17.0%)	1.7(2.0%)	16.9(19.0%)

Table 2: Comp program statistics

This table presents statistics for the fraction of instructions that can be supported in the Comp subsystem for each of the integer benchmarks. This computation is divided into branch computation (shown in first column) and store value computation (shown in second column) that is eligible for execution in the Comp subsystem. All counts are in millions. The percentages are calculated with respect to the total number of instructions executed by the benchmark.

gcc, a small yet significant fraction of the total execution can be supported in the Comp units. Also, the data in the table shows that the majority of instructions assigned to the Comp subsystem come from branch computation. Overall, the data shows that even under a restricted model that does not require major modifications to existing datapaths, a modest to significant amount of computation can be executed in the augmented floating-point units resulting in higher utilization of hardware resources. Hence, we feel that the proposed microarchitecture has promise.

Mix of the Comp instructions

In order to study the extra functionality that has to be incorporated into the floating-point execution units for supporting integer computation, we analyzed the instruction mix of the computation that can be assigned to the Comp subsystem. Table 3 shows the breakup of the computation into arithmetic, logical and control operations. We found that the extra functionality required is the ability to perform integer add, subtract and logical operations. The integer multiply instruction was completely absent from the Comp subsystem computation for our benchmarks. This suggests that the integer multiplier, whose implementation is usually gate-intensive, need not be duplicated and can be located in its usual place in the integer subsystem.

Benchmark	Arithmetic	Logical	Control
compress	56%	1%	43%
espresso	28%	31%	41%
eqntott	39%	20%	41%
gcc	17%	29%	54%
sc	23%	23%	54%

Table 3: Instruction mix of Comp slice

This table presents the instruction mix of the integer instructions that can be supported in the Comp subsystem. The percentages are calculated with respect to the total number of dynamic instructions that can be assigned to the Comp subsystem.

5 Future work

We plan to extend this study in a number of ways. In this preliminary study we did not measure the performance gains that are possible (or the hardware savings) or the relationship between the performance and the hardware. There are a number of issues we plan to study.

First, can the traditional benefits of decoupling be tapped by the proposed microarchitecture? Can the load/store program in the LdSt subsystem naturally run out ahead or does the computation in the Comp subsystem hold it back? It is possible that with good branch prediction the load/store program can run ahead of the rest of the computation.

Second, is it possible to obtain a better balance between the LdSt and Comp subsystems by assigning instructions that are *almost* pure computation to the Comp subsystem, and provide additional instructions for copying between the register files and/or copying through memory?

Finally, we sidestepped the compiler issues by making use of run-time information. For the proposed microarchitecture to be useful, the compiler has to detect and mark computation that can be executed in the Comp subsystem. We plan to study this and other related compiler issues.

6 Summary and Conclusions

In this paper, we address the issue of how computation is partitioned in conventional microarchitectures. Existing microarchitectures suffer from idle floating-point units when executing integer codes. We use slicing analysis to decompose program execution into broad classes of memory access, control and pure computation. Using the resulting data, we propose an alternative microarchitecture that facilitates better utilization of hardware resources by using an augmented floating-point subsystem to perform some of the computation in integer programs. The set of instructions executing in the integer subsystem primarily consists of instructions involved in address computation and memory access. The set of instructions executing in the augmented floating-point subsystem mainly includes branch-related instructions that do not use register values computed by instructions executing in the integer subsystem. Our results show that between 10% and 39% of the total computation in our integer benchmarks can be supported in the augmented floating-point subsystem. Furthermore, the only extra functionality that has to be added to existing floating-point units is the ability to perform simple integer operations. Our conclusion is that the proposed microarchitecture has promise and deserves further study.

Acknowledgements

Thanks to the anonymous reviewers for informative comments. Doug Burger, Shamik Das Sharma and T.N. Vijaykumar read a draft of the paper and provided input that helped improve the presentation. This work was supported by grants from the National Science Foundation (grant MIP-9505853) and the University of Wisconsin Graduate School.

References

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

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

[3] A.R. Pleszkun and E.S. Davidson. Structured Memory Access Architecture. In *International Conference on Parallel Processing*, pages 461–471, 1983.

[4] Richard. M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, January 1978.

[5] Michael Slater. AMD's K5 Designed to Outrun Pentium. *Microprocessor Report*, 8(14), October 1994.

[6] J.E. Smith. Decoupled Access/Execute Computer Architecture. In *The 9th Annual International Symposium on Computer Architecture*, pages 112–119, April 1982.

[7] J.E. Smith and et. al. The ZS-1 Central Processor. In *The 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–204, October 1987.

[8] SPEC. (entire issue). *SPEC Newsletter*, 3(4), December 1991.

[9] Sun Microsystems Laboratories, Inc. *Introduction to Shade*, April 1993.

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