Proc. 25th Annual International Symposium on Microarchitecture (MICRO-25), 1992

Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors

Manoj Franklin and Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53706, USA

Abstract

Traditionally, register files have been the primary agent for inter-operation communication in load/store architectures. As processors start issuing multiple instructions per cycle, a centralized register file can easily become a bottleneck. This paper analyzes the register file traffic in a load/store architecture with a view to motivate the development of alternate inter-operation communication mechanisms that reduce the bandwidth demanded of a centralized register file.

We first provide metrics to characterize the register traffic. These metrics deal with the degree and locality of use of the register instances created. We then present the results of a simulation study that uses the MIPS R2000 architecture and the SPEC benchmark programs. We have two major results. First, a large number of the register instances are used only once, and the average degree of use of register instances is about 2. Second, most of the register instances are used up soon after they are created (within about 30-40 instructions). This suggests that alternate inter-operation communication mechanisms that exploit the temporal locality of use of register instances are likely to be effective in reducing the traffic burden on the centralized register file. The second result was pivotal in the design of the distributed register file for the multiscalar processing paradigm.

1. Introduction

The execution of a program, in an abstract form, is a dynamic dataflow graph whose nodes represent *computation* operations and arcs represent the *communication* of values between the computation nodes. The abstract dataflow graph encapsulates the data dependencies in the program, but contains no information about the distances involved in the communication of values, or for that matter, even the type of mechanism used for the communication. When the graph is mapped onto a processing structure, irrespective of the processing model used, computation operations are always carried out by means of functional units; communication of values, on the other hand, depends on the processing model used.

Most modern CPUs use a load/store instruction set architecture (*ISA*), in which a set of *ISA-visible registers* are the primary means of inter-operation communication. Specifically, all computation instructions fetch their operands from, and place their results back in the ISA-visible registers, and explicit load and store instructions move data between the memory and the registers.

An adequate inter-operation communication mechanism is central to the design of any processor. In a processor that issues a peak of one instruction per cycle, a register file with 2 read ports and 1 write port is sufficient to provide the required bandwidth. Modern fine-grain parallel processors, on the other hand, attempt to issue multiple instructions per cycle, posing higher bandwidth demands on the inter-operation communication mechanism. If a centralized register file is the primary means of this communication, it will inevitably be burdened by the high demands of multiple instruction issue. Decentralizing the inter-operation communication mechanism is essential.

1.1. Register Instances and Register File Traffic

Each time a datum is written into a register, a new *register instance* (or register name) is *created*. Succeeding reads to the register *use* the latest register instance. Normally, the creation of each register instance requires a write access to the register file, and each use of a register instance requires a read access to the register file. The writing and reading of the centralized register file is the fundamental inter-operation communication mechanism in a load/store architecture.

To sustain an *average* issue rate of S instructions per cycle (in a load/store architecture with dyadic ALU operations), we need a register bandwidth of *at least* $2S$ reads and S writes per cycle. Supporting $2S$ reads and S writes with a single register file, for the values of S (5-10) that we expect to see in the next few years, requires a complex register file as well as a complex interconnect between the register file and the functional units. In CMOS, the area required to build read ports is proportional to the square of the number of ports [12]. Whereas a few multi-ported register files have been built, for example the register files for the Cydra 5 [15], the SIMP processor [14], Intel's iWarp [12], and the XIMD processor [19], we feel that centralized, multi-ported register files are not a good long-term solution, and that alternate means need to be explored. Ideally, these alternate means should retain the elegance of the ISA-visible register file (for example the easy and familiar

0-8186-3175-9/92 $3.00 © 1992 IEEE

compilation model), but at the same time provide the needed bandwidth with a decentralized realization.

The first step that was proposed in this direction was to split the register file into an integer file and a floating-point (FP) file, each file having fewer ports. It was even proposed to provide independent control to the integer and FP units, resulting in a decoupled architecture [2, 17]. While in theory this partitioning could reduce the bandwidth requirements of each register file by a factor of 2, in practice the bandwidth reduction achieved and the issue rate achieved depends upon the mix of FP and integer operations in the program. Moreover, because of data movement between the two register files, the total traffic handled by the two files might actually be larger than the traffic handled by a single, consolidated register file. In any case, this separation of register files is only a temporary solution to the bandwidth problem.

VLIW processors, such as the TRACE /300 [5], take the above approach one step further by having multiple, ISA-visible integer and FP register files. Whereas this approach may be suitable for a VLIW processor having a sophisticated compiler that can move data between the various register files of each type, it is not suitable for superscalar or other fine-grain parallel processors that have the notion of a single set of ISA-visible registers. Interestingly, Multiflow retracted to the single register file model in their later model TRACE /500 [6].

1.2. Reducing the Register File Traffic

How can we reduce the register file traffic? The obvious answer is to provide some form of auxiliary storage elements that can carry out most of the inter-operation communication in a decentralized manner. Previously, such mechanisms have been used explicitly, *i.e.*, in a manner visible to the ISA. For example, decoupled architectures use explicit memory-to-processor queues for loads to communicate with computation instructions, and processor-to-memory queues for computation instructions to communicate with stores [2, 17]. Likewise, systolic arrays use systolic queues that allow some computation instructions to communicate with others directly [3, 7]. If one adheres to a load/store architecture in which all explicit communication is through the ISA-visible registers (as we expect most superscalar implementations to be), then the auxiliary storage must take on the role of implicitly forwarding a register instance directly from the producer to the consumer(s), without going through the register file.

Whereas register read traffic can be reduced by simple data forwarding, register write traffic reduction is harder. Discarding the write of a register instance to the register file requires a guarantee that the instance will *never* be used again in a valid execution of the program. If such a guarantee is available, the instance can simply be discarded from the auxiliary storage after any potential cusumers have consumed it. There are 2 ways to enforce this guarantee. First, if adequate compile-time support is available to recognize the *last use* of a register instance, the compiler could mark the instruction corresponding to the last use. If, at run-time, the marked instruction is issued before the register instance is written to the register file, then the write can be suppressed.

If the use of compile-time support is not an option (for example if object code compatibility is required), or adequate compile-time support is not available to detect the last use of register instances, then at run-time the only way we can guarantee that a register instance will never be used in the future is when a new instance has been created for the same register.

1.3. Paper Objective and Outline

The purpose of this paper is two-fold. The first is to study (both qualitatively and quantitatively) the nature of inter-operation communication in real programs. For a load/store architecture, this reduces to the study of the register traffic, since all inter-operation communication is through registers. Such a study is important to develop alternate mechanisms that can satisfy the inter-operation communication bandwidth demands of fine-grain parallel implementations of load/store architectures.

The second is to show how the register file usage characteristics were exploited in the design of a distributed register file for the *multiscalar*, erstwhile ESW (Expandable Split Window), processing paradigm [9]. The central idea of the distributed register file design was to have multiple versions of the register file, thereby keeping the number of read and write ports of each register file much less than that in the register file of conventional VLIW and superscalar processors.

The outline of this paper is as follows. Section 2 introduces metrics to characterize the register file traffic. Section 3 presents the results of an empirical study of the register file traffic. Section 4 describes how the register traffic characteristics were exploited in the design of a distributed register file. Section 5 summarizes and presents our conclusions.

2. Register Traffic Metrics

The first step in the design of a streamlined inter-operation communication mechanism is an understanding of the nature of communication that takes place in a typical program. In this context, two characteristics of the register traffic are important: (i) the *degree of use of register instances*, and (ii) the *temporal locality of creation and use of register instances.*

2.1. Degree of Use of Register Instances

This metric indicates the number of times register instances are used by other instructions. The motivation for this metric is that, if most register instances are used only once or a few times, and the uses are within a small window of dynamic instructions, then techniques such as data forwarding within the small window are likely to be effective. If register instances are used many times, and the uses are spread out across the program, auxiliary storage in the execution unit is not likely to suffice as a viable inter-operation communication mechanism, and the register file is perhaps the best mechanism.

Before conducting empirical studies with a specific architecture and specific compiler(s), we shall first do a qualitative analysis. Such an analysis transcends architectures and compilers, and adds confidence to the results obtained in a specific study. Indeed, a simple analysis of program execution can provide us with the *average* degree of use of register instances, $\bar{D}_u$. This analysis is based on the fact that the average degree of use is equal to the total number of register instance uses (*i.e.,* total number of read accesses R) divided by the total number of register instances created (W). That is,

$$\bar{D}_u = \frac{R}{W}$$

What would be the value of $\bar{D}_u$ in an arbitrary program? A naive guess, prompted by the myth that registers hold only frequently-accessed variables, is that the average degree of use would be very high. This guess is incorrect for two reasons. First, even if a particular register holding a frequently-accessed variable may be accessed repeatedly with reads and writes, each *instance* of the register is not. Second, many register instances are created to hold intermediate computation values that do not appear in the source program, and are used only once or a few times. The average degree of use of register instances, therefore, can be expected to be low.

In load/store architectures, most of the dynamic instructions are dyadic computation instructions, which read 2 registers and write 1 register. (Among the rest, some read 2 registers and write none (stores and some branches), and some read 1 register and write 1 register (loads and moves)). This means that, over the execution of the program, the total number of register reads R can be expected to be about twice the total number of register writes W. That is, *the average degree of use of register instances in a load/store architecture is only about 2*. Such a low value of $\bar{D}_u$ implies the existence of a large number of register instances that are used either 0, 1, or 2 times, *regardless of the program being executed*. In section 3, we will indeed see that our experimental results tally with this qualitative assessment.

2.2. Temporal Locality of Register References

If many of the register instances are indeed used only a few times, then the best way of exploiting this phenomenon to reduce the register file traffic is to use some form of data forwarding. However, data forwarding schemes are effective only if instances are used up soon after creation. Since the ‘‘locality’’ of use of register instances is important in determining the efficacy of such techniques, our second set of metrics deals with the temporal locality of creation and use of register instances. For studying temporal locality, we measure temporal distance in terms of the number of dynamic instructions executed. We consider three measures of locality: (i) the age of register operands (*i.e.,* the temporal distance between a register instance use and its creation), (ii) the useful lifetime of register instances (*i.e.,* the temporal distance between the creation and last use of register instances), and (iii) the lifetime of register instances.

If our auxiliary storage is of the form that it buffers the results of the last N instructions before writing them into the register file, if need be, the first measure gives us an indication of the register file read traffic that could be eliminated with such buffering, the second measure gives an upper bound on the register file write traffic that could be eliminated with perfect knowledge about the last use of register instances, and the third measure gives an indication of the amount of register file write traffic that could be eliminated in practice (with no knowledge of the last use of register instances).

3. Empirical Analysis of Register Traffic

In this section, we present the results of an empirical study of the register traffic in a MIPS R2000 processor with an R2010 FP coprocessor [13]. The MIPS R2000 architecture is representative of the class of load/store architectures that have emerged recently; other architectures in this class have very similar traits [10]. Briefly, the MIPS R2000, with an R2010 FP coprocessor, has separate integer and FP register files. The integer register file has 32 registers, each of which is 32 bits wide. R0 is used as a special register to hold the constant value 0. Since R0 is not a general-purpose register for which new register instances can be created, we exclude it from all of our statistics. The R2010 has 16 FP registers that can hold single-precision (32 bits) or double-precision (64 bits) values.

3.1. Data Gathering Tools and Benchmarks

All data reported in this paper are gathered with a simulator that accepts programs written for the MIPS R2000-based DECstation 3100, and simulates their execution, keeping track of relevant information on a cycle-by-cycle basis. System calls made by the simulated program are handled with the help of traps to the operating system. The collected statistics therefore exclude the code executed during system calls, but includes all other code portions, including the library routines.

For benchmarks, we use the SPEC ’89 suite. Data is presented for the following 10 benchmark programs: **`eqntott`** with input file **`int_pri_3.eqn`**, **`espresso`** with input file **`bca`**, **`gcc`** with input file **`stmt.i`**, **`xlisp`** with input file **`li-input.lsp`**, which are integer-intensive programs written in C, and **`dnasa7`**, **`doduc`**, **`fpppp`**, **`matrix300`**,

spice2g6, and **tomcatv**, which are floating-point-intensive programs (FP programs) written in Fortran. The programs were compiled using the MIPS C and FORTRAN compilers (version 1.31). Compiler effects are discussed in Section 3.4. For the C benchmarks, statistics are collected for the entire run, and for the FORTRAN benchmarks, statistics are collected for the first *1 billion* instructions.

Table 1 presents the total number of instructions executed (in million), and the total number of reads and writes (in million) to each register file. Blank entries indicate zero or negligible number of accesses.

Table 1: Number of Instructions Executed and Register Read/Write Traffic (in million)

Benchmarks	Instr.	Register Traffic			
		Floating Point		Integer	
		Writes	Reads	Writes	Reads
eqntott	1485			855	1587
espresso	522			352	540
gcc	147			81	146
xlisp	1247			599	1101
dnasa7	1000	953	1248	228	706
doduc	1000	698	949	216	633
fpppp	1000	867	1001	192	593
matrix300	1000	361	451	592	1136
spice2g6	1000	40	49	625	1051
tomcatv	1000	823	1034	248	800

3.2. Degree of Use of Register Instances

Table 2 presents the degree of use distributions of the FP and integer register instances. The entries in each column are the percentage of register instances of a particular type (FP or integer) with the listed degree of use. For example, in Table 2 we see that for **xlisp**, 6.97% of the integer instances are used 4 times or more. Surprisingly, several instances are not used at all, *i.e.*, there are cases where a value is written into a register and later overwritten before being read even once. This happens because the use of register instances during program execution depend upon the dynamic execution path taken through the code. Since a compiler cannot accurately predict the paths that would be dynamically taken through the code, it is forced to place values (e.g. parameters of procedure calls) in registers that might not be read at run-time.

From Table 2 we see that, just as we argued qualitatively, *a large number of register instances in all the benchmarks are used only once*. For example, 99.83% of the FP instances in **dnasa7** and 99.92% of the FP instances in **matrix300** are used only once. This single-use property is not surprising; it has been routinely used in vector machines in the form of chaining to reduce the traffic to the vector register file (and of course to decrease the overall latency and increase the throughput of a sequence of vector operations) [16]. More recently, superscalar architectures such as the IBM RS/6000 [1] attempt to exploit this phenomenon, in a limited manner, by replacing a sequence of operations that produce single-use results by higher strength operations (e.g., IBM RS/6000's multiply-add fused unit [11]). Recall that in load/store architectures, single-use instances are typically created because of the register file being used as an intermediate storage for computations.

In the integer programs also, most register instances are used only once; however, the percentage is lower than that of the FP register instances of the FP programs. The reason for this is that, apart from being used as an intermediate storage for communicating results from one instruction to another in the computation stream, the integer registers are also used for address manipulation and loop control. Whereas the former class of register instances can be expected to be mostly single-use instances, the latter class can be expected to have a higher degree of use. This is borne out by the integer registers of the FP benchmarks, which are used mainly for address manipulation and loop control.

Finally, we note that although the register instances that are used ≥ 4 times are a small percentage of all register instances, they can contribute heavily to the register read traffic. For example, the 0.08% of FP register instances that are used ≥ 4 times in **matrix300** (c.f. Table 2) account for 20% of the read traffic to the FP register file (as we shall see in Figure 1(i)).

3.3. Temporal Locality of Register References

We now consider the issue of temporal locality of register references. In keeping with our overall goal of reducing the traffic to the centralized register file, we also consider the potential reduction in read/write traffic to the register file with auxiliary storage in the execution unit.

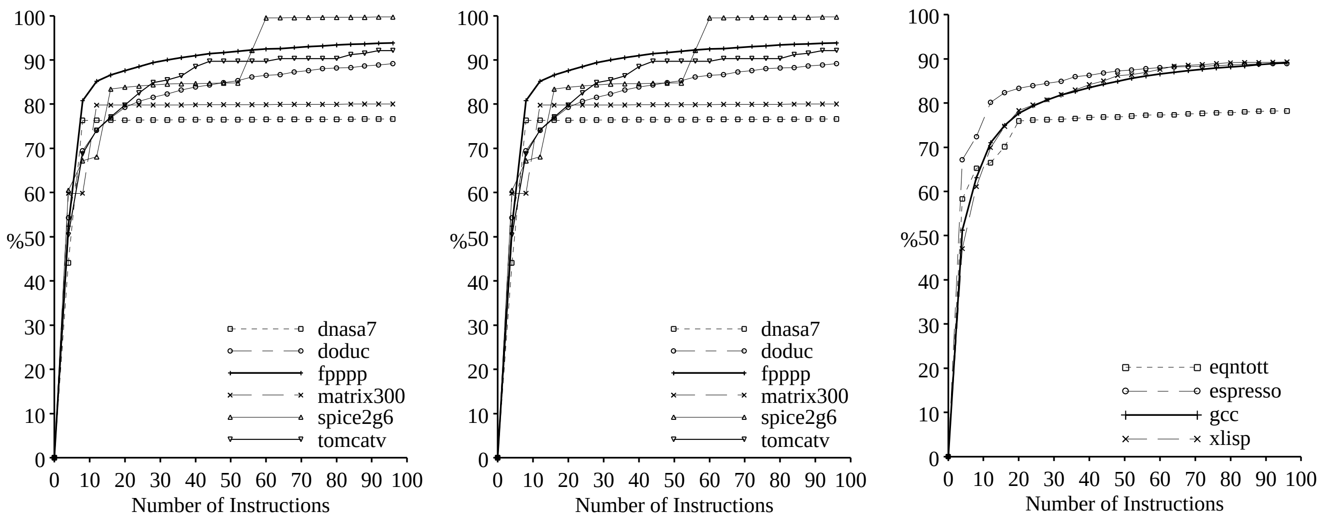
3.3.1. Age of Register Operands

Figure 1 presents the cumulative percentage distribution of the age of register operands, measured in terms of dynamic instructions or the dynamic window size. In Figure 1 (also Figures 2 and 3), the first and second graphs show the data for the FP and integer registers, respectively, of the FP benchmarks, and the third graph shows the data for the integer registers of the integer benchmarks. The X-axis denotes the dynamic window size and the Y-axis denotes the cumulative percentage. For example, in Figure 1(i), 80% of all FP register instances read in **matrix300** were created in the last 10 instructions executed.

From Figure 1, we can assess the potential reduction in register read traffic that can be attained by data forwarding. For example, by buffering the results of the last 30 instructions (of all types) in the execution unit, we can reduce over 75% and 55-95% of the read traffic to

Table 2: Degree-of-Use Distribution of Register Instances

Benchmarks	Floating-Point Register Instances						Integer Register Instances					
	Percentage with Degree of Use					Average Degree of Use	Percentage with Degree of Use					Average Degree of Use
	0	1	2	3	≥4		0	1	2	3	≥4	
eqntott							0.89	71.34	17.54	9.47	0.76	1.86
espresso							3.67	72.30	17.66	3.74	2.63	1.48
gcc							6.26	67.37	15.51	4.45	6.41	1.69
xlisp							4.27	66.14	12.42	10.20	6.97	1.84
dnasa7	0.00	99.83	0.02	0.03	0.12	1.31	0.67	2.36	16.29	64.36	16.33	3.28
doduc	1.46	84.00	9.51	1.94	3.09	1.36	10.31	44.35	26.52	10.13	8.69	2.93
fpppp	0.16	91.09	6.15	1.14	1.46	1.16	1.34	10.12	83.45	0.46	4.63	3.09
matrix300	0.00	99.92	0.00	0.00	0.08	1.25	15.29	61.54	7.71	0.12	15.35	1.92
spice2g6	0.21	79.85	19.22	0.16	0.56	1.22	4.04	73.38	12.08	3.56	6.94	1.68
tomcatv	0.00	86.43	8.30	1.49	3.77	1.26	0.12	24.99	37.54	27.40	9.96	3.22



(i) FP Operands of FP Benchmarks (ii) Integer Operands of FP Benchmarks (iii) Integer Operands of Integer Benchmarks

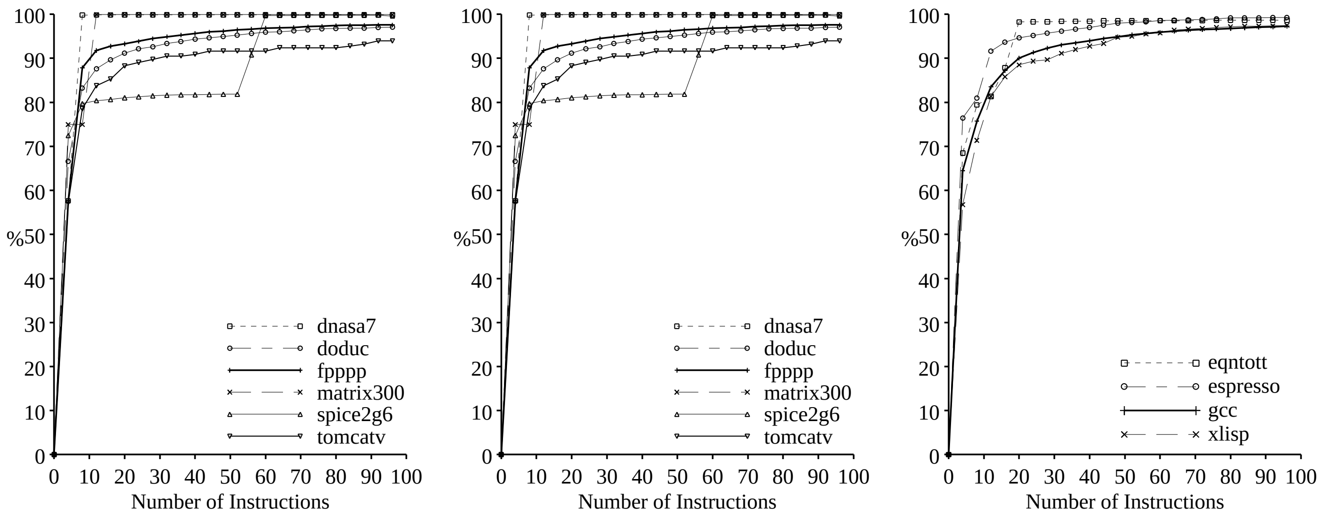
Figure 1: Cumulative Percentage Distribution of the Age of Register Operands

the FP and integer register files, respectively, for the FP benchmarks, and over 75% of the register read traffic for the integer benchmarks. More buffering would help somewhat, but not significantly, because a fair amount of the read traffic is due to register instances created more than 100 instructions ago. Our experience suggests that this remaining read traffic is mainly due to a small number of frequently-used values, such as loop invariants. Even though loop invariants form a small percentage of all register instances, they contribute to a significant percentage of register reads, since each loop-invariant instance may be read several hundred times.

3.3.2. Useful Lifetime of Register Instances

Figure 2 presents the cumulative percentage distribution of the temporal distance between the creation and last use (*i.e.*, the useful lifetime) of register instances. The information in Figure 2 can be used to determine the amount of register write traffic that could be eliminated, if we have perfect knowledge about the last use of each register instance, and if we use that knowledge to delay writes to the register file. For example, if we delay writing back the result of an instruction until 30 more instructions enter the execution unit, and we have perfect knowledge about the last use of each register instance, in more than 80% of all cases (except for the integer registers of `tomcatv`), we will have encountered the last use of the register instance and therefore the register instance need not be written back, resulting in more than 80% reduction in register write traffic.

It is interesting to see how the data in Figures 1 and 2, and Table 2 are related. Figure 2 shows that for all benchmarks except `tomcatv`, only about 10% of the register instances are not used up within 100



(i) FP Instances of FP Benchmarks (ii) Integer Instances of FP Benchmarks (iii) Integer Instances of Integer Benchmarks

Figure 2: Cumulative Percentage Distribution of the Useful Lifetime of Register Instances

instructions of being created. Earlier in Figure 1, we noticed that only 70-80% of the operands were generated within the previous 100 instructions. Combining these two facts together, we can see that about 10% of the register instances contribute to 20-30% of the source operands of all instructions. This information lends support to our earlier observation (from Table 2) that a few, and only a few, of the register instances are used more than a few times, and that the ones that fall into this category are used a large number of times. Consider, for example, **`matrix300`**. In Figure 1(i), the curve for **`matrix300`** stays nearly constant at 80%, indicating that about 20% of the FP operands are coming from at least 100 instructions away in the past. In Figure 2(i), the curve for **`matrix300`** stays constant at 99.92%, *i.e.*, about 99.92% of the FP register instances are used up within 100 instructions of their generation. We can infer from this that, for **`matrix300`**, about 0.08% of the FP register instances account for 20% of the FP register read traffic. (This is also true for the integer registers, but the resolution in Table 2, *i.e.*, the breakdown of the ≥ 4 category, is not sufficient to show this.)

3.3.3. Lifetime of Register Instances

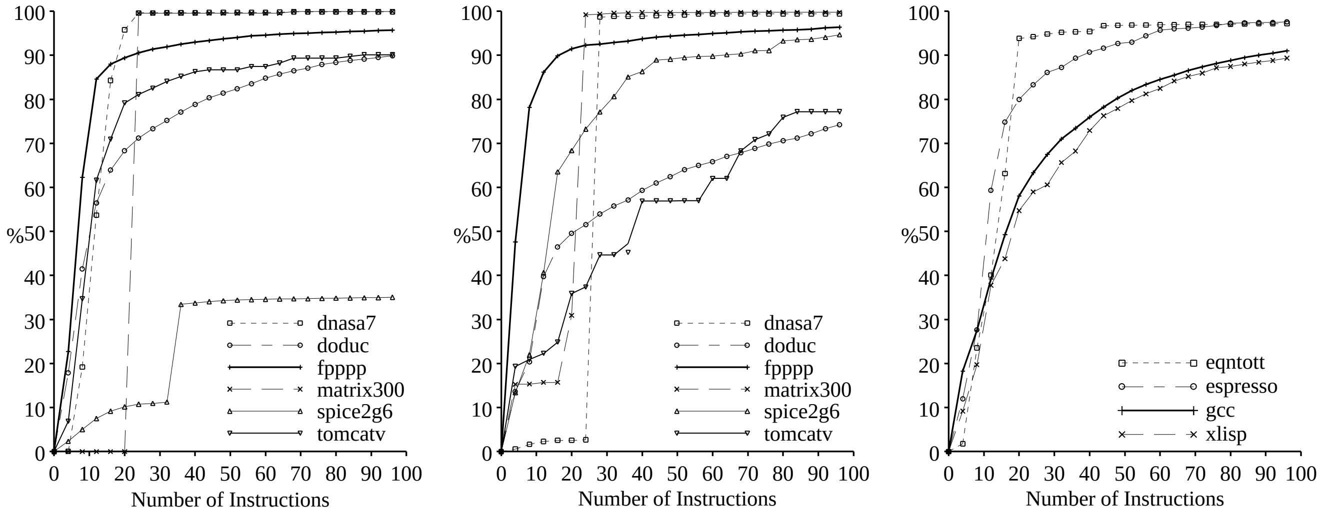
Figure 3 presents the cumulative percentage distribution of the lifetime of register instances. We can infer from Figure 3 that, if an instruction's result is not written until 25-30 more instructions have entered the dynamic window, we can suppress over 50% of the register writes in almost all cases, saving over 50% of the write traffic.

3.4. Effect of Compiler and Instruction Set

So far we have ignored the compiler issue. Whereas our specific results are applicable solely to the MIPS R2000, with the MIPS C and FORTRAN compilers (version 1.31), we expect to see similar trends with other load/store architectures because many of the results are a manifestation of the nature of the programs themselves, and the way code is written for such machines, regardless of the inter-operation communication mechanism. Most programs are written in an imperative language for a sequential machine having a limited number of ISA-visible registers for storing intermediate values.

The dyadic/monadic nature of instruction sets, and dependencies caused by a limited number of registers are factors that could change the results, albeit slightly. Depending upon the number of ISA-visible registers, and the register allocation algorithms used by different compilers (such as minimizing the number of live registers or lifetimes of register instances), the lifetime values would change, but not as much as to void the results of our study. For instance, if most of the instructions create a new register instance, then the *average* lifetime of an instance will be roughly equal to the number of registers. (To be precise, the *average* lifetime in terms of dynamic instructions is upper bounded by the number of registers divided by the fraction of instructions that produce a new instance.)

The average value of the degree of use of register instances depends on the dyadic/monadic nature of instruction set and the dynamic instructions encountered. Our results could change somewhat if the instruction set architecture is changed. To add confidence to our results, we compiled the C benchmarks with the GNU C compiler also, and collected the statistics. We observed very similar results with the GNU C compiler.



(i) FP Instances of FP Benchmarks (ii) Integer Instances of FP Benchmarks (iii) Integer Instances of Integer Benchmarks

Figure 3: Cumulative Percentage Distribution of the Lifetime of Register Instances

4. An Application

Having discussed some important register traffic characteristics, let us now turn our attention to how one might use them to develop alternate inter-operation communication mechanisms. In this section, we discuss how some of the characteristics were used in the development of the multiscalar processing model [9] for exploiting fine-grain parallelism, and the distributed inter-operation communication mechanism for the multiscalar model. The objective of this section is not to advocate a particular processing model or inter-operation communication mechanism, but only to illustrate how some of the results of Section 3 can be put into effective use. Indeed, many different schemes could be developed based on these results and the execution models in which the schemes are used. For example, single-use instances can be exploited by means of compound functional units.

The central idea behind the multiscalar inter-operation communication mechanism is to exploit the dataflow properties present in programs by providing multiple versions of the register file. This helps to keep the number of read and write ports per register file much less than that needed in conventional VLIW and superscalar processors. We shall briefly describe the multiscalar processing paradigm below; more detailed information can be found in [9].

4.1. The Multiscalar Processing Paradigm

In section 3, we saw that *most of the register operands are generated in the immediate past, and that most of the register instances are used up almost immediately after creation. This means that instructions of close proximity, quite likely, are dependent*. Notice that this does not necessarily mean that instructions farther apart are independent; one could argue that because near-neighbor instructions are dependent, then by transitivity, these dependencies could propagate forward (and backward) from any given instruction in the stream such that all instructions could, in fact, be dependent. We shall see why this need not be the case. An inspection of the dynamic dataflow graph of many sequential programs reveals that there exists a large amount of theoretically exploitable instruction-level parallelism [4, 8], *i.e.*, a large number of computation nodes that can be executed in parallel, provided a suitable processor model with a suitable inter-operation communication mechanism exists. In that case, if instructions of close proximity are dependent, *then many of the farther-apart instructions should be independent*. This means that most of the parallelism can be found only further down in the dynamic instruction stream. The obvious way to get to that parallelism is to use a large window of dynamic instructions.

Using a large enough dynamic window is not sufficient by itself; the hardware required to extract independent instructions and to enforce dependencies in a large window typically involves wide associative searches, and is non-trivial. There should be some means of decentralizing the critical resources in the system. If instructions of close proximity are most likely dependent, and instructions farther apart are independent, then we can consider a block of instructions (most likely having dependencies) as a single subwindow, and exploit fine-grain parallelism by overlapping the execution of multiple subwindows. This is the central idea behind the multiscalar paradigm. Instead of considering a consolidated large dynamic window, the principle of temporal locality of creation and use of register instances is applied, and the large window is split into smaller subwindows (c.f. Figure 4). This helps to decentralize the critical resources in the system.

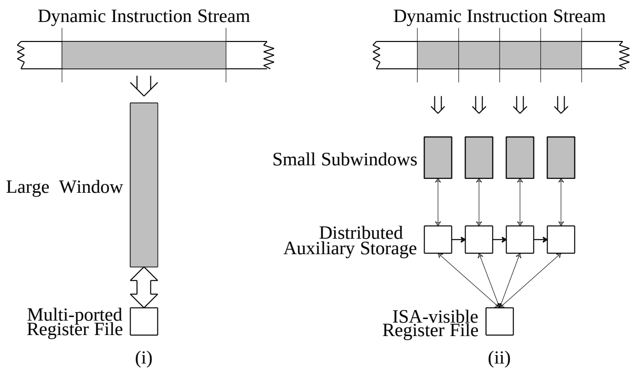


Figure 4: (i) A Single Large Window
(ii) Split into Small Subwindows

The multiscalar processor consists of several independent, identical *stages*, each of which is equivalent to a typical datapath found in modern processors. The stages conceptually form a circular queue, with hardware pointers to the head and tail of the queue. These pointers are managed by a control unit, which also performs the task of assigning subwindows to the stages. The active stages, the ones from the head to the tail, together constitute the large dynamic window of operations, and they contain subwindows, in the sequential order in which they appear in the dynamic instruction stream. In any given cycle, up to a fixed number of ready-to-execute instructions begin execution in each of the active stages. When all the instructions in the stage at the head have completed execution, the stage is committed, and the head pointer is moved forward to the next stage.

4.2. Distributed Inter-Operation Communication Mechanism

Let us now look into the inter-operation communication mechanism of the multiscalar processor. In the multiscalar processor, at any one time, there could be more than 100 active operations, many of which may start execution simultaneously. Clearly, a centralized register file cannot handle the amount of register traffic needed to support such a large number of active operations; a decentralized mechanism is essential.

For decentralizing the communication mechanism, we again utilized the results of section 3. In particular, we utilized the result that a significant number of register instances are used up and eventually overwritten soon after they are created. This means that if we have a local register file for each stage, much of the communication in a stage can be handled by the local register file itself. Only the last updates to the registers in a stage need be passed on to the subsequent stages. Furthermore, most of these last updates need not propagate beyond one or two stages, because a new instance would soon be created for those registers. The local register files, which we call *future files*, work similar in spirit to the *future file* proposed in [18] for implementing precise interrupts in pipelined processors. Thus we exploit the *temporal locality of creation and use of register instances* to design a good decentralized register file structure that ties well with the multiscalar execution model. The distributed system also helps to maintain precise state at each subwindow boundary, which significantly eases recovery actions in times of incorrect branch prediction.

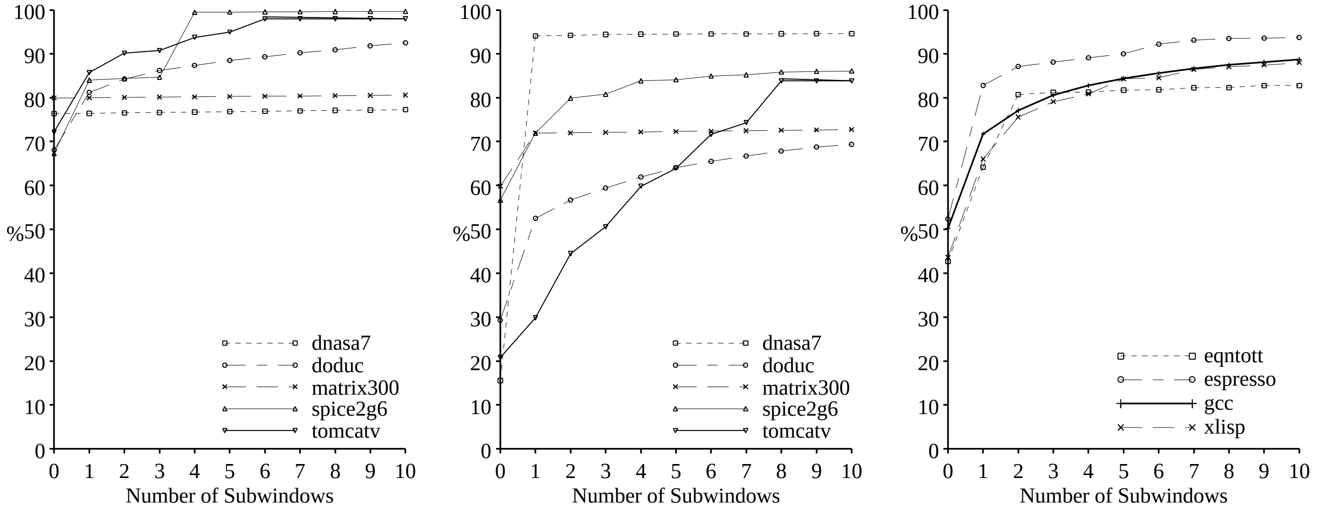
4.3. Traffic Handled by Local Register Files

To study the efficacy of the distributed register file in the multiscalar model, we conducted several simulation experiments. It is worthwhile to see how much of the read traffic is being handled by the local register files, and how much of the rest are generated in subwindows close by in the past. If many of the operands come from subwindows immediately in the past, it means that the register traffic flowing across subwindow boundaries is less, and "localized".

Figure 5 shows the cumulative percentage distribution of the register operands based on the number of subwindows away in the past they were created, for a multiscalar implementation with subwindow size of up to 32 instructions. In Figure 5, the data points falling on the Y-axis (*i.e.*, the points corresponding to "Number of Subwindows = 0") depict the percentage of operands that are generated in the same subwindow. Needless to say, the graphs of Figure 5 closely follow those in Figure 1. It can be seen from Figure 5 (i) that for the FP benchmarks, a large portion of the FP operands (ranging from 68-80%) are generated in the same subwindow. For the FP benchmarks' integer operands, about 50-95% of the operands come from at most 3 subwindows in the past. Recall that integer operands do not form a substantial portion of the read traffic for the FP benchmarks (c.f. Table 1).

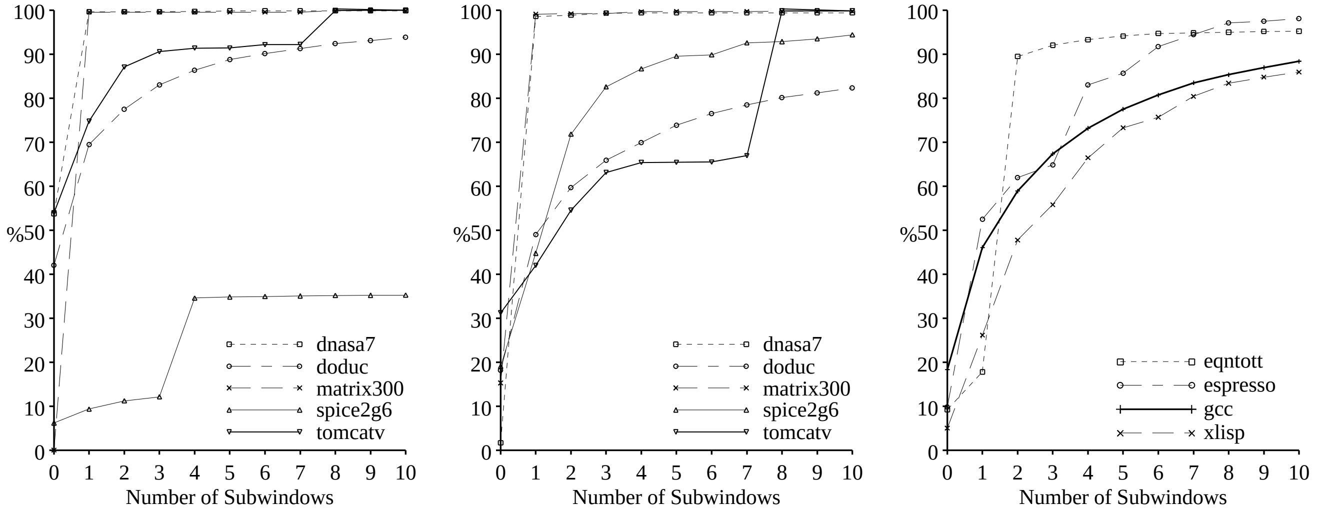
Figure 6 shows the cumulative percentage distribution of the lifetime of register instances in terms of dynamic subwindows of the multiscalar implementation. Notice that the graphs of Figure 6 closely follow those of Figure 3. For a dynamic subwindow size of up to 32 instructions, except for **`dnasa7`**, a large portion of the FP instances (ranging from 70% to 99.92%) are used up either in the same stage in which they were created or in the subsequent stage. For the integer write traffic also, the write traffic reduction is substantial.

The distributed register file thus provides sufficient bandwidth to meet the demands of the multiscalar processor, demands that would have been difficult to meet with a centralized register file. The efficacy of the distributed future file system for the multiscalar model is further substantiated by the high sustained issue rates we obtained in our simulation studies of the multiscalar model, some of which were reported in [9].



(i) FP Operands of FP Benchmarks (ii) Integer Operands of FP Benchmarks (iii) Integer Operands of Integer Benchmarks

Figure 5: Cumulative Percentage Distribution of the Age of Register Operands in the Multiscalar Implementation



(i) FP Instances of FP Benchmarks (ii) Integer Instances of FP Benchmarks (iii) Integer Instances of Integer Benchmarks

Figure 6: Cumulative Percentage Distribution of the Lifetime of Register Instances in the Multiscalar Implementation

5. Summary and Conclusions

As processors start issuing several instructions per clock cycle, the inter-operation communication bandwidth must be improved commensurately. If all inter-operation communication is carried out through a centralized register file, the register file can easily become a bottleneck. Alternate mechanisms are needed to support the bandwidth demands of fine-grain parallel processors capable of sustaining an issue rate of more than 4 or 5 instructions per cycle. To provide a foundation for the investigation of alternate mechanisms, we studied the register file usage in a load/store architecture (the MIPS R2000) using the SPEC benchmarks.

We saw that most of the register instances are used only once; techniques to exploit this property merit further investigation. The average number of times a register value is used is about 2. A few register instances are used heavily, and these correspond to constant FP values and values used in address manipulation and loop control. We also saw that many of the register instances are used for the last time within 20 instructions, and almost all are used up within 100 instructions.

Our results suggest that decentralized storage in the execution unit, to forward data from one instruction to another, has significant potential as an inter-operation communication mechanism, and in reducing the bandwidth required of a centralized register file. For example, by delaying writes to the register file by 25-30 instructions, and by buffering these values in the execution unit, we can reduce more than 80% of the read traffic, and more than 50% of the write traffic to the register file.

We saw how some of these results were used in the development of the multiscalar model. In the multiscalar implementation, multiple versions of the register file were used, giving a decentralized inter-operation communication mechanism. With a subwindow size of up to 32 instructions, a significant portion of the register read traffic is handled by local register files. Also, most of the register instances do not propagate beyond 3-4 subwindows, indicating that the distributed register file system does not present a bottleneck to performance.

Admittedly, many more schemes could be developed to exploit register traffic characteristics. We believe that a decentralized inter-operation communication mechanism is indispensable to the design of a fine-grain parallel processor, and researchers and designers should exploit localities of communication, as evidenced in this paper, to construct inter-operation communication mechanism (perhaps with the user-appearance of a centralized ISA-visible register file) suitable for future fine-grain parallel processors.

Acknowledgements

We thank Todd Austin for his comments. Financial support for this work was provided by NSF grant CCR-8919635 and by an IBM Graduate Fellowship.

References

[1] ‘‘Special Issue,’’ *IBM Journal of Research and Development*, vol. 34, pp. 23-36, January 1990.

[2] J. R. Goodman *et al*, ‘‘PIPE: a Decoupled Architecture for VLSI,’’ *Proc. 12th Annual Symp. on Computer Architecture*, pp. 20-27, June 1985.

[3] M. Annaratone *et al*, ‘‘The Warp Computer: Architecture, Implementation and Performance,’’ *IEEE Transactions on Computers*, vol. C-36, pp. 1523-1538, December 1987.

[4] M. Butler *et al*, ‘‘Single Instruction Stream Parallelism Is Greater than Two,’’ *Proc. 18th Annual Int’l Symp. on Computer Architecture*, pp. 276-286, 1991.

[5] R. P. Colwell *et al*, ‘‘A VLIW Architecture for a Trace Scheduling Compiler,’’ *IEEE Trans. on Computers*, vol. 37, pp. 967-979, August 1988.

[6] R. P. Colwell *et al*, ‘‘Architecture and Implementation of a VLIW Supercomputer,’’ *Proc. Supercomputing ’90*, pp. 910-919, 1990.

[7] S. Borkar *et al*, ‘‘Supporting Systolic and Memory Communication in iWarp,’’ *Proc. 17th Annual Int’l Symp. on Computer Architecture*, pp. 70-81, 1990.

[8] T. M. Austin and G. S. Sohi, ‘‘Dynamic Dependency Analysis of Ordinary Programs,’’ *Proc. 19th Annual Int’l Symp. on Computer Architecture*, pp. 342-351, 1992.

[9] M. Franklin and G. S. Sohi, ‘‘The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism,’’ *Proc. 19th Annual Int’l Symp. on Computer Architecture*, pp. 58-67, 1992.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990.

[11] E. Hokenek, R. K. Montoye, and P. W. Cook, ‘‘Second-Generation RISC Floating Point with Multiply-Add Fused,’’ *IEEE J. of Solid-State Circuits*, vol. 25, pp. 1207-1213, October 1990.

[12] R. Jolly, ‘‘A 9-ns 1.4 Gigabyte/s, 17-Ported CMOS Register File,’’ *IEEE J. of Solid-State Circuits*, vol. 26, pp. 1407-1412, October 1991.

[13] G. Kane, *MIPS R2000 RISC Architecture*. Englewood Cliffs, New Jersey: Prentice Hall, 1987.

[14] K. Murakami, N. Irie, M. Kuga, and S. Tomita, ‘‘SIMP (Single Instruction Stream / Multiple Instruction Pipelining): A Novel High-Speed Single-Processor Architecture,’’ *Proc. 16th Annual Symp. on Computer Architecture*, pp. 78-85, 1989.

[15] B. R. Rau, D. W. L. Yen, W. Yen, and R. Towle, ‘‘The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs,’’ *IEEE Computer*, vol. 22, pp. 12-35, January, 1989.

[16] R. M. Russel, ‘‘The CRAY-1 Computer System,’’ *CACM*, vol. 21, pp. 63-72, January 1978.

[17] J. E. Smith, ‘‘Decoupled Access/Execute Architectures,’’ *Proc. 9th Annual Symp. on Computer Architecture*, pp. 112-119, April 1982.

[18] J. E. Smith and A. R. Pleszkun, ‘‘Implementing Precise Interrupts in Pipelined Processors,’’ *IEEE Trans. on Computers*, vol. 37, pp. 562-573, May 1988.

[19] A. Wolfe and J. P. Shen, ‘‘A Variable Instruction Stream Extension to the VLIW Architecture,’’ *Proc. ASPLOS-IV*, pp. 2-14, April 1991.