Register Communication Strategies for the Multiscalar Architecture

T.N. Vijaykumar, Scott E. Breach and Guri S. Sohi
{vijay, breach, sohi}@cs.wisc.edu
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706

Abstract

This paper considers the problem of register communication in the Multiscalar architecture, a novel paradigm for exploiting instruction level parallelism. The Multiscalar architecture employs a combination of hardware and software mechanisms to partition a sequential program into tasks, and uses control and data speculation to execute such tasks in parallel. Inter-task register dependencies represent register communication in the architecture. The two primary issues in register communication for a Multiscalar processor are correctness and performance. Not only must proper values be directed from producers to consumers, these values must be sent as early as possible from producing tasks to consuming task in order to avoid execution stalls which may have a critical impact on overall performance. We present a simple model to ensure that register communication obeys the sequential semantics of the program. To this model, we apply a progression of hardware techniques (including register data speculation) and compiler analyses to alleviate stalls due to inter-task register communication. Finally, we perform an experimental evaluation of these hardware and compiler techniques on a Multiscalar processor configuration. The key result we obtained is that aggressive hardware support for register data speculation can be out performed by simpler hardware supplemented by compiler analyses.

1 Introduction

The Multiscalar architecture[1] [2] is a novel paradigm to exploit instruction level parallelism. Sequential programs are partitioned into code fragments called *tasks*, which are assigned to a collection of processing units connected via a uni-directional ring for communication. Each processing unit executes the instructions of its task until completion. The simultaneous execution of multiple tasks on multiple processing units results in the execution of multiple instructions per cycle. The architecture requires that the individual execution of each task as well as the aggregate execution of all tasks maintain the appearance of sequential program order.

Prediction unravels the control flow between tasks, and each predicted task is assigned to a processing unit for execution. For this collection of predicted tasks, execution may occur speculatively, but modification of architectural state can only occur non-speculatively. Upon completion, tasks are retired in program order to maintain sequential semantics. A combination of hardware and software mechanisms are used to ensure that control and data dependencies are honored as per the original sequential program specification [3] regardless of what transpires in the actual parallel execution.

As instructions in tasks execute, data values are produced and consumed within the same task and between different tasks, corresponding to intra-task and inter-task communication, respectively. These data values are bound to memory and register storage locations. In the case of memory storage, it is difficult to determine precisely the producers and consumers of data values since memory storage names are determined dynamically (via address calculations). On the other hand, in the case of register storage, it is straightforward to identify producers and consumers since all register storage names are known statically (via register specifiers).

Regardless of the type of storage involved, data values passed between instructions represent a critical factor in the implementation of the architecture, impacting two key aspects of program execution: correctness and performance. To ensure maintaining correctness, data values must be communicated from producing to consuming instructions as dictated by sequential semantics. To avoid constraining performance, data values must be communicated from producing to consuming instructions as soon as possible.

While correctness and performance need to be addressed for both intra-task and inter-task data communication, the issues associated with inter-task data communication represent a more pressing challenge, especially given that many of the issues for intra-task data communication have been dealt with rigorously in the context of scalar processors. Moreover, experience with the design of Multiscalar processors has shown that inter-task dominates intra-task data communication in terms of its impact on mechanisms to provide correctness and performance.

In this work, we focus on the communication of data values bound to register storage for two reasons (both related to the prevalence of the load-store model of computation). First, this type of communication is the most common. Second, it is the most amenable to analysis. To this end, we devote the rest of this paper to the investigation of a number of alternative register communication strategies for the Multiscalar architecture. The key idea in these strategies is to send register values from producers to consumers as early as possible to avoid execution delays (See Figure 1(a)).

In Section 2, we present the Multiscalar register model and correctness criteria. In Section 3, we describe a range of alternative register communication strategies. We begin with a simple base strategy that guarantees correctness; we progressively incorporate hardware and compiler mechanisms to achieve higher performance. In Section 4, we discuss related work. In Section 5, we evaluate each of these alternatives on a set of well-known benchmarks to observe the effects of each additional mechanism on overall performance. In Section 6, we offer concluding remarks.

2 Register Communication Model

The register file of a Multiscalar processor provides the appearance of a logically centralized register file, yet is implemented as physically decentralized register files, queues, and control logic [4]. Each processing unit has its own set of hardware registers; hence, each task has its own renamed version of the hardware registers. This approach allows a Multiscalar processor to exploit intra-task register communication locality within a single processing unit and to recover the precise architectural register state among multiple processing units in an efficient manner.

Before we proceed with the description of register communication, we must better define a Multiscalar task, since it plays an integral role in the overall model. Put succinctly, tasks are single entry, multiple exit partitions of the control flow graph of the program. It is important to realize that this definition places few restrictions on the internal control flow of a task. A task may be part of a basic block, a basic block, multiple basic blocks, a single loop iteration, an entire loop, or even a function call to name a few possibilities. Such flexibility has ramifications on register communication (as we describe in the sections to follow).

In the set of all architectural registers, there are two mutually exclusive and collectively exhaustive subsets: the set of registers that may be modified in the task, called the *ModSet,* and the set of registers that are guaranteed not to be modified in the task, called the *UnModSet*. Assume that during execution, every task eventually receives values for all the architectural registers from its predecessor and eventually sends values for all the architectural registers to its successor. (In an actual implementation, there are many hardware optimizations that reduce the bandwidth demands of register communication.)

When a register value arrives at a task, the hardware identifies the register either as a ModSet register or as a UnModSet register, by consulting the ModSet. The task binds the register value to its hardware register, regardless of whether it is a ModSet register or a UnModSet register. Any register value generated during the course of executing the instructions of the task is also bound to its appropriate hardware register.

The distinction between a ModSet and a UnModSet register is important only when the task sends the register value to successor tasks. For a UnModSet register, the same value that was received is propagated to successor tasks. For a ModSet register, the value that was received is stopped from propagating further; the hardware propagates whatever value is bound to the register when the last modification of the register is encountered in the task. Figure 1(b) illustrates the Multiscalar register model.

The compiler provides the ModSet to the hardware on a per task basis. (The UnModSet is the complement of the ModSet and therefore is not provided separately.) Besides determining the ModSet for each task, the compiler/hardware also has to identify at what points in the task each ModSet register may be sent to successor tasks. Informally, such points are the locations in the task beyond which the register is guaranteed not to be modified by the execution of the remaining instructions of the task.

This constraint on correctness may be relaxed so that each register may be sent multiple times, meaning the compiler/hardware may send the register from other basic blocks. However, the compiler/hardware must guarantee for each register on the ModSet that the last value to be sent for the register is the one dictated by sequential semantics. Using such an approach provides the capability to perform register data speculation during execution. To ensure sequential semantics, any task that receives a register it has already received earlier is squashed and restarted with the latter value.

The problem of identifying the ModSet is simple and straightforward; it is merely the union of the sets of registers that may be modified in any path through the task. Since tasks may have complex control flow within them, the problem of identifying at what points to send a ModSet register is somewhat more involved, especially if the ability to use register data speculation is exercised.

There are two correctness constraints on this process. First, every path that leads to an exit of the task must send all the registers in the ModSet to later tasks. Second, for each register in the ModSet, the compiler/hardware has to identify the basic block beyond which the register is guaranteed not to be modified subsequently in any path through the task and has to ensure that the value bound to the register in this basic block is the last one sent to later tasks.

The first constraint ensures that consuming tasks are not starved for a needed register value, irrespective of the path taken by the producing task. The second constraint guarantees that all tasks are provided the correct register values as per the sequential semantics of the program. Together, the two constraints guarantee forward progress and correctness. Neither constraint, however, precludes the use of register data speculation (as we describe in the next section).

3 Register Communication Strategies

In the previous section, we explained the register model for the Multiscalar architecture. Multiscalar tasks execute in parallel communicating register values from producers to consumers. In order to avoid stalling consumer tasks for

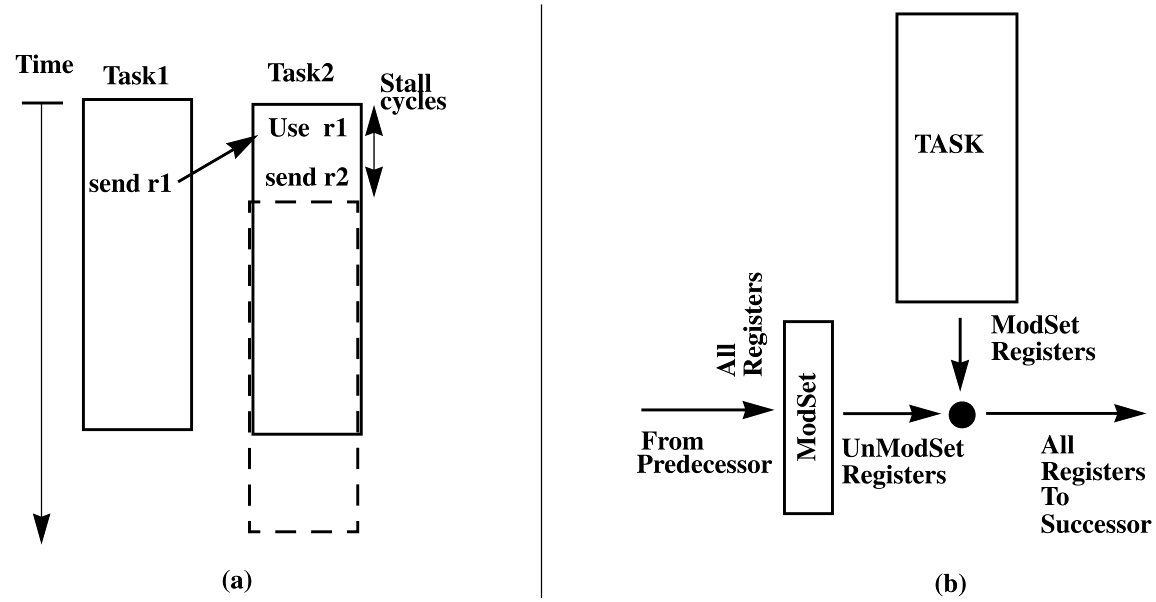

Figure 1: (a) Performance impact of register communication. Task2 needs register r1 but is stalled because Task1 sends r1 late. The delay cascades because, in turn, Task2 sends r2 late. (b) An abstraction of Multiscalar register communication. All of the registers arriving at a task are filtered by the ModSet so that the old values of the UnModSet registers are propagated and the new values of the Modset registers are sent, as and when generated by the task.

register values, it is crucial that register values from producer tasks are sent as soon as possible. In this section, we consider a progression of strategies to achieve high performance implementations of the Multiscalar register model.

3.1 Overview

The simplest strategy, called *End_Send*, to ensure correctness is to send the values of all the ModSet registers at the end of execution of the task. A possible improvement in performance over this simple strategy is to send the value of a ModSet register every time it is modified. This strategy is called *Eager_Send*. Since the value corresponding to the last modification of each register is sent last (and the hardware preserves the order among the different sends of the same register), this strategy preserves the semantics of the program. Although Eager_Send may send register values earlier than End_Send, Eager_Send may send the same register multiple times, resulting in squashing of subsequent tasks. If the last modification of each ModSet register is known, however, multiple sends of the same register (and the resultant squashes) can be avoided.

To this end, in the next strategy called *Last_Send*, the compiler identifies the last modification of each ModSet register and marks it explicitly for communication. The Last_Send strategy avoids squashes by sending a register only after its last modification is known. Nevertheless, in the presence of control flow, it is conservative and may delay sending register values. The last strategy, known as *Spec_Send*, speculatively sends register values corresponding to last modifications from "high frequency" paths and resorts to squashes if any "low frequency" paths modify a previously sent register. Spec_Send attempts to avoid both delaying register values due to infrequent paths and also excessive squashing due to incorrect speculation.

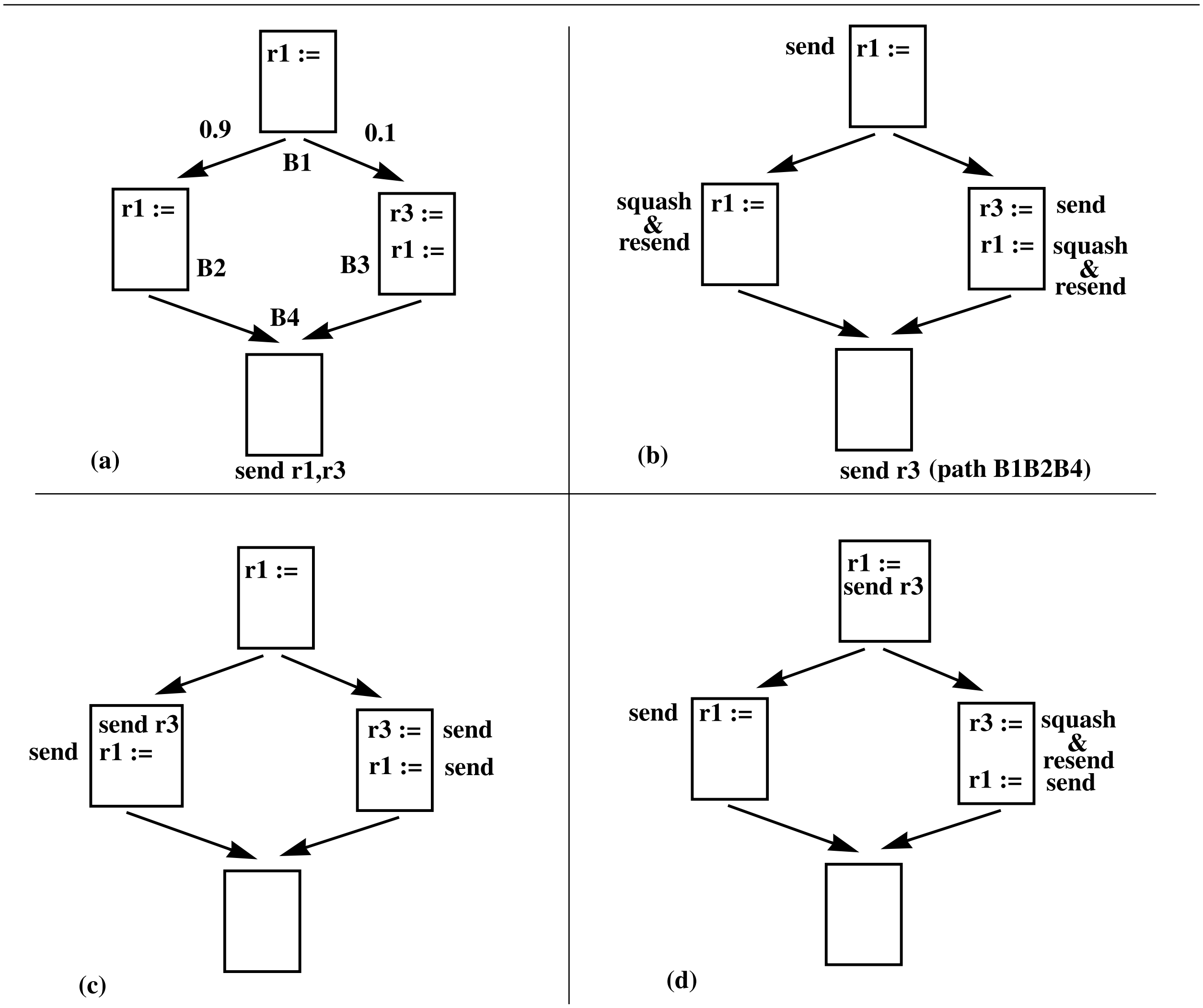

Figure 2: An example illustrating the four strategies. The task comprises basic blocks B1, B2, B3 and B4. The edge B1B2 is taken with probability 0.9 and the edge B1B3 is taken with probability 0.1. The ModSet for this task contains registers r1 and r3. (a) End_Send: Both r1 and r3 are sent at the end. (b) Eager_Send: r1 is sent in B1 but since there is a define of r1 in B2 and B3, r1 causes a squash and is sent again. If B2 is taken then r3 is sent at the end otherwise r3 is sent in B3. (c) Last_Send: r1 and r3 are sent from B2 and B3. Sending r3 from B2 may require an extra instruction and in the other cases the existing instructions may be annotated to send their destination registers. (d) Spec_Send: Since B2 is more frequent than B3, r3 is speculatively sent from B1 and r1 is sent from B2 and B3 like Last_Send. If B3 is taken then r3 causes a squash and is sent again. Sending r3 from B1 may require an extra instruction, and in the other cases the existing instructions annotated like Last_Send.

Before going into the details of the various strategies, let us illustrate them through an example. Consider the task in Figure 2 comprising basic blocks B1, B2, B3, and B4. Let us assume that the edge B1B2 is taken with probability 0.9, and the edge B1B3 is taken with probability 0.1. In End_Send (Figure 2(a)), both r1 and r3 are sent at the end of execution, causing subsequent tasks that need r1 or r3 to wait. Eager_Send (Figure 2(b)) sends r1 as soon as it is defined in B1. Since there is a define of r1 in both B2 and B3, r1 is sent again, causing a squash. If B3 is taken then r3 is sent as soon as it is defined, otherwise it is sent at the end. Thus, Eager_Send sends r3 earlier than End_Send if B3 is taken but incurs extra squashes due to sending incorrect values of r1.

In Last_Send (Figure 2(c)), both r1 and r3 are sent from B2 or B3, since they are the last modifications of the registers. Since B2 does not define r3, an extra instruction may be inserted to send r3. Last_Send avoids the squashes incurred by Eager_Send by sending only the correct values of r1; it sends r3 earlier than Eager_Send via an explicit instruction. Spec_Send (Figure 2(d)) speculatively sends r3 from B1, since B3 is infrequently taken, and the frequent path of B1B2B4 does not define r3. If B3 is taken then r3 is sent again, causing a squash. As is the case for Last_Send, r1 is sent from B2 or B3. Spec_Send sends r3 even earlier than Last_Send by taking advantage of speculation. In the frequent case of B2 being taken, r3 is sent earlier from B1 via an explicit instruction, but in the infrequent case of B3, this send of r3 results in a squash.

3.2 Details

Let us now describe the details in the realization of each of these four strategies. End_Send and Eager_Send may be implemented with no compiler support and no ISA modification. Last_Send and Spec_Send require compiler analyses to determine the last modification of each register in the presence of complex control flow. The analyses required may be formulated in terms of existing data flow frameworks. (We present a few data flow equations instead of the complete framework in the sections following. We assume in these data flow equations that only the last modification of a register in a basic block is considered for the analyses.) In addition, both strategies may require ISA modification in order to efficiently convey information from the compiler to the hardware.

3.2.1 End_Send

End_Send can be implemented with relatively simple hardware and does not require any compiler or ISA support. Once all instructions of the task have executed, all registers in the ModSet may be sent (one by one). Unfortunately, this strategy is likely to provide low performance since register values needed by instructions in a consuming task must wait until all instructions in the producing task have executed. In addition, it does not take advantage of register data speculation. Nevertheless, this strategy is used as a base for each of the other strategies, since registers in the ModSet that are not modified during the execution of the task or that cannot be sent, due to control flow, until the end of the execution of the task are easily handled.

3.2.2 Eager_Send

Eager_Send builds upon the End_Send base strategy by allowing the hardware to send any modification of a register, as a part of the execution of the associated instruction; it does not require any compiler or ISA support. The hardware for this scheme may be quite complex, since it must track the modification of all registers on the ModSet and ensure that the last modification is the last value to be sent for the register. Assuming instructions are allowed to complete execution out-of-order (regardless of whether instructions issue in-order or out-of-order), it may be necessary to inspect all instructions in the dynamic window to ensure registers are sent in the proper order if sends are not forced to occur in the specified program order.

Nevertheless, this strategy is more likely to provide high performance as compared to the base strategy. The advantage of this strategy is that a value may be sent as soon as the register is produced, thereby reducing register wait time for the value to be consumed. Unfortunately, this form of register data speculation has no information about the last modification of registers and therefore is by nature uncontrolled. As such, the disadvantage of this strategy is that multiple values for the same register may be sent by the producing task often, thereby causing frequent squashes of the consuming tasks. Such squashes may increase the cost of incorrect execution enough to negate any decrease in the register communication delay.

3.2.3 Last_Send

This strategy uses compiler support to provide static analyses of register dependence and may require changes in the ISA. It is capable of employing register data speculation, but this addition is reserved for the Spec_Send strategy. The key issue in Last_Send (and in Spec_Send, since it is a variation of Last_Send) is, for each ModSet register, to determine in the presence of complex control flow the basic blocks after which the register is guaranteed not to be modified, and to generate sends for the register in the earliest such basic block down any path through the task. Figure 3 lists the data flow equations for Last_Send. NoMoreDef determines the basic blocks after which a register is guaranteed not to be modified down any path from this point in the task. Send identifies the earliest basic block down any path through the task among such basic blocks.

$$NoMoreDef(i, r) = \prod_{j \in children(i)} \{NoMoreDef(j, r) \cap \neg BBDef(j, r)\}$$

$$Send(i, r) = \left\{ \sum_{j \in parents(i)} \neg(NoMoreDef(j, r)) \right\} \cap NoMoreDef(i, r)$$

INITIAL VALUES: $NoMoreDef(i, r) = TRUE$

Figure 3: Data flow equations for Last_Send: BBDef(i,r) is true if register r is modified in basic block i. NoMoreDef(i,r) is true if there is a guarantee that register r is not modified in any successor of basic block i in the current task. Send(i,r) is true if NoMoreDef(i,r) is true and register r has not been sent in at least one path from the entry of the task to basic block i.

Informally, NoMoreDef(i,r) is true if beyond basic block i, register r is not modified in the task. BBDef(i,r) is true if register r is modified in basic block i. The equation asserts that for every child of basic block i, if there is no modification of the register beyond the child, and if the child does not modify the register, then there is no modification of the register beyond basic block i. A key property of NoMoreDef is that if it is true for a basic block, then it is true for all of its successors in the task.

Informally, Send(i,r) is true if register r is not sent in at least one path from the entry of the task to basic block i, and if NoMoreDef(i,r) is true. The equation asserts that if there is no modification of the register beyond basic block i, and if NoMoreDef is false for at least one of the parents of basic block i, then the register is sent from basic block i.

Send indicates the first basic block, for a given path from the entry of the task, where NoMoreDef changes from false to true. This basic block is the earliest in the given path where the register may be sent. Because a task may have multiple overlapping paths from its entry, there may be multiple first basic blocks (one for each distinct path) where NoMoreDef changes from false to true. As a result, on a given path Send may be true for multiple basic blocks along the path (corresponding to places where other paths overlap). In this case, after the earliest send of the register is encountered, all later sends of the register are redundant and are ignored by the hardware.

3.2.4 Spec_Send

With the analyses of Last_Send, a register is not sent until it can be guaranteed that no further modification of the register can occur. This guarantee implies that an infrequently executed basic block that may modify a register late in the task, must delay sending from a frequently executed basic block that modifies the register early in the task. For the Spec_Send strategy, the early modification of the register may be sent so long as the late modification of the register is sent later if needed. These sends must be distinguished from the redundant sends described above so as not to be ignored. The same complex hardware that ensures registers are sent in the proper order for the Eager_Send strategy may be required if sends are not forced to occur in the specified program order.

This strategy is similar to the Eager_Send strategy. Not surprisingly, the advantage of this strategy is that in the frequent case when the late modification of the register does not occur, the register is sent as soon as possible. The disadvantage of this strategy is that in the infrequent case when the late modification of the register does occur, multiple values for the same register may be sent by the producing task, as in the Eager_Send strategy, thereby causing squashes of the consuming tasks. However, unlike Eager_Send, this strategy does have information about the last modification of a register and may perform register data speculation in a controlled fashion. Given accurate profiling information about the expected characteristics of program execution, Spec_Send stands a far better chance than Eager_Send of using register data speculation profitably.

Figure 4 lists the data flow equations for Spec_Send. The key difference between the analyses for Last_Send and Spec_Send is that Spec_Send elides the “low frequency” defines of registers, but otherwise computes SpNoMoreDef and SpSend exactly as Last_Send computes NoMoreDef and Send. NoIgnore, distinguishes those sends which cannot be ignored, corresponding to the “low frequency” defines elided in SpSend.

Informally, SpNoMoreDef(i,r) is true if beyond basic block i, register r is not modified in “high frequency” basic block i in the task. HiFreqDef(i,r) is true if register r is modified in basic block i, and if the execution frequency of the basic block is greater than a fraction, SpThreshold, of the frequency of the task entry. The equation asserts that for every child of basic block i, if there is no “high frequency” modification of the register beyond the child, and if the child does not modify the register with “high frequency”, then there is no “high frequency” modification of the register beyond basic block i.

Informally, NoIgnore(i,r) is true if basic block i modifies register r with “low frequency”, and if SpNoMoreDef(i,r) is true. The purpose of NoIgnore is to identify the sends that cannot be ignored for registers elided due to the “low frequency” of the basic block that performs the modification. Note that NoIgnore may cause multiple sends of the same

$$SpNoMoreDef(i, r) = \prod_{j \in children(i)} \{SpNoMoreDef(j, r) \cap \neg HiFreqDef(j, r)\}$$

$$SpSend(i, r) = \left\{ \sum_{j \in parents(i)} \neg SpNoMoreDef(j, r) \right\} \cap SpNoMoreDef(i, r)$$

$$HiFreqDef(i, r) = BBDef(i, r) \cap HiFreqNode(i)$$

$$HiFreqNode(i) = Freq(i)/Freq(entry) > SpThreshold$$

$$NoIgnore(i, r) = SpNoMoreDef(i, r) \cap BBDef(i, r) \cap \neg HiFreqNode(i)$$

INITIAL VALUES: $SpNoMoreDef(i, r) = TRUE$

Figure 4: Data flow equations for Spec_Send. HiFreqDef(i,r) is true if register r is modified in basic block i and the execution frequency of the basic block is greater than a fraction (SpThreshold) of the frequency of the entry of the task. SpNoMoreDef(i,r) is true if there is a guarantee that register r is not modified in any "high frequency" successor of basic block i in the task. SpSend(i,r) is true if SpNoMoreDef(i,r) is true and register r has not been sent in at least one path from the entry of the task to basic block i. NoIgnore(i,r) is true if i is a "low frequency" basic block that modifies the register r.

register, resulting in multiple squashes. Such multiple squashes may be avoided by restricting the computation of NoIgnore, but we do not discuss this option further here.

3.3 Conveying Information from Software to Hardware

The compiler solves the system of data flow equations to determine the basic blocks for which Send or SpSend and NoIgnore are true. If such a basic block contains an instruction that modifies the register, then the instruction is annotated to *forward* the register. Otherwise, an extra instruction, called a *release* instruction, is inserted at the top of the basic block to send the register. As register data communication is fairly dense, performing sends by annotating existing instructions as forwards is imperative, since performing sends with releases requires adding extra instructions which may inevitably impact the critical path of execution through the program. In Figure 5, there are two examples of tasks with register communication annotations corresponding to the Last_Send strategy.

3.4 Implications of Dead Registers and Register Assignment

If a register is dead beyond a task, then the register value need not be sent to successor tasks. Dead register information may be used to reduce register bandwidth demand as well as avoid unnecessary release instructions. Dead register information may be conveyed to the hardware by overloading the ModSet. Registers that are dead beyond a task are included in the ModSet of the task, regardless of whether they are defined in the task or not; yet, no forward or

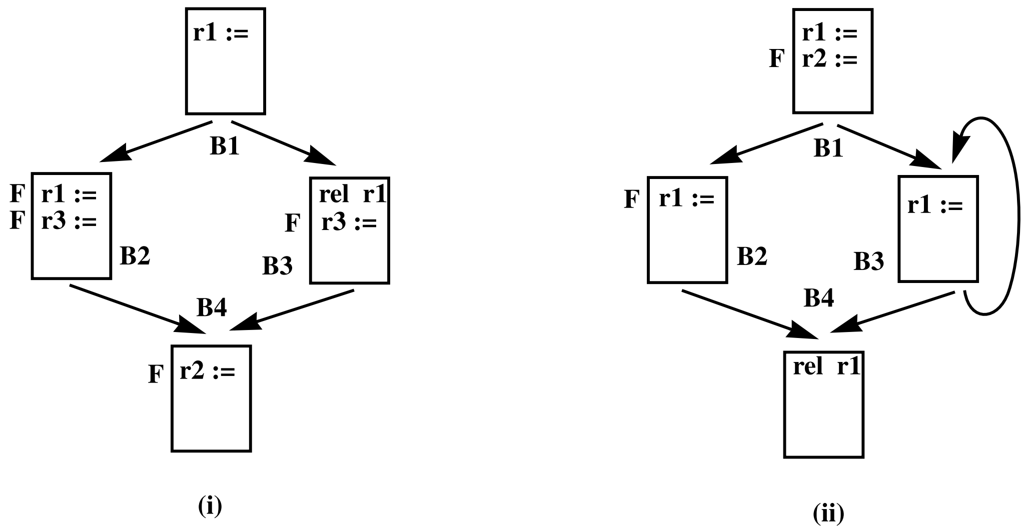

Figure 5: Annotation of Register Communication using Last_Send. Two examples of tasks with register communication annotations are shown. Forwards of registers are indicated by F's beside the instructions. In example (i), in the path B1B2B4, r1 is last modified in B2 and is forwarded from B2 and in the path B1B3B4, r1 is not modified past B1 and is released in B3. r3 is last modified in B2 and B3, in paths B1B3B4 and B1B2B4 respectively, and is forwarded from them. r2 is modified only in B4 and is forwarded from there. In example (ii), r1 is last modified in B2 in the path B1B2B4 and is forwarded from B2. Since there is a loop in the path B1B3B4, r1 is guaranteed not to change only on exit from the loop and so is released in B4.

release is performed for these registers. If the register is defined in the task, then the value is used within the task, but is not propagated to successor tasks. If the register is not defined in the task, then the value from a predecessor task is used within the task, but is not propagated to successor tasks. Since the register is dead, no successor task need wait for the register value. Both Last_Send and Spec_Send can take advantage of such dead register information.

Anti-dependencies introduced by register assignment may delay the sending of a register (See Figure 6); the use of round-robin assignment of registers may mitigate this problem. Figure 6(a) shows a task with four basic blocks. Pseudo-register r1 is dead in B4 but is live out of B3. Pseudo-register r2 is live out of B4. Pseudo-registers r1 and r2 have non-overlapping live ranges and hence can be assigned the same physical register. Figure 6(b) shows such a register assignment. For the Last_Send strategy, this assignment causes $1 to be sent only in B3. A round-robin assignment, as shown in Figure 6(c), avoids delaying the send of $1 for Last_Send by assigning r2 to another physical register. The Eager_Send strategy would cause incorrect sends and squashes for the assignment in Figure 6(b), but the round-robin assignment would not incur any such squashes.

4 Related Work

Data flow analyses have been applied to a large number of problems in compiler optimizations. The basic register communication problem falls under the category of simple *backward flow* problems[5]. Compilers of data parallel languages like Fortran D [6] and HPF [7] generate data communication for distributed memory machines. But these works do not include any speculative communication.

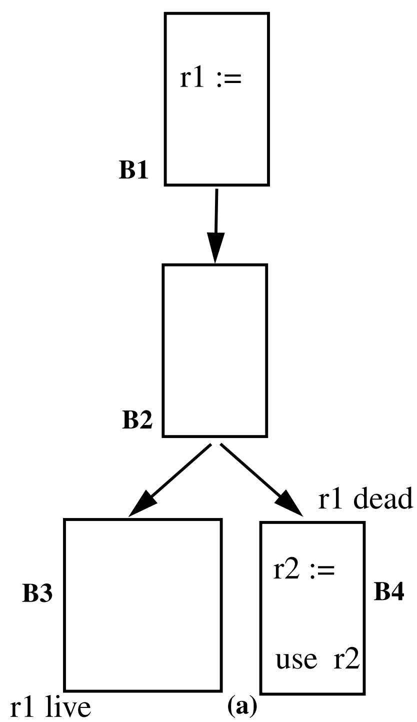

Figure 6: An example of register assignment delaying register send. (a) A task with 4 basic blocks before register allocation. r1 and r2 are pseudo-registers. r1 is dead at the beginning of B4 but live at the end of B3. (b) Both r1 and r2 are assigned the same register $1, resulting in the release of $1 in B3. (c) r1 is assigned $1 and r2 is assigned $2. $1 is forwarded in B1, much earlier than B3; $2 is forwarded independent of the forward of $1.

Annotating instructions with information from compiler analyses has been implemented in Torch[8]. Static speculation have been applied to superscalar, VLIW and superpipelined machines. Techniques like guarding[9][10], percolation scheduling [12], trace scheduling [13], boosting [15], hyperblock scheduling [16], superblock scheduling [17], sentinel scheduling[11] and modulo scheduling[12] employ varying degrees of static speculation.

Utilizing speculative execution in the compiler constitutes two issues: (i) decision of when to perform computation under speculation, and (ii) recovery from incorrect speculation. The first issue applies to compiling speculative register communication in the Multiscalar architecture as well. The second issue is addressed by a combination of hardware and compiler in the Multiscalar architecture. The compiler inserts code (in the form of “resends”) to detect any incorrect static speculation and the hardware performs recovery. Unlike most VLIW machines, no bookkeeping code need be inserted in the program but recovery may be more expensive since larger numbers of instructions may be squashed.

5 Experimental Evaluation

We have evaluated the register communication strategies for the Multiscalar architecture described in the previous sections using a compiler derived from the Gnu C Compiler (*gcc)* and a simulator that we have developed to model the characteristics of a Multiscalar processor. Our compiler is configured to produce code for the MIPS1 ISA without any delay slots. The code generation phase of gcc has been modified to annotate the assembly output with Multiscalar specific information. The Gnu Assembler (*gas)* and the Gnu Linker (*gld*) have also been modified to pass these annotations to the output binary. The binary generated by our compiler is run on our simulator which faithfully models the behavior of a Multiscalar processor on a cycle-per-cycle basis and produces output result files for verification. All the simulations reported in this paper were run to completion, and the outputs were verified against the reference outputs.

5.1 Framework

To put the problem of compiling register communication in perspective, we start out with an overview of the Multiscalar compiler. The responsibilities of the compiler include partitioning sequential programs into tasks that are likely to have few dependencies between each other, maintaining correctness by specifying control and data dependencies between the tasks to the hardware, and improving performance by streamlining control and data dependencies between the tasks. Corresponding to these requirements, the problem of compiling for the Multiscalar architecture involves: (i) devising heuristics to obtain suitable tasks, (ii) determining inter-task data (both memory and register) communication and inter-task control flow to maintain correctness, and (iii) scheduling inter-task data (both memory and register) communication to mitigate performance loss.

The organization of our compiler, which is derived from *gcc*, is shown in Figure 7. After a series of traditional phases like jump optimizations, loop optimizations, and common sub-expression elimination, the Multiscalar compiler partitions the program into tasks. For the purpose of this paper, tasks may be assumed to be sub-graphs of the control flow graph, with a single entry point and an arbitrary number of exit points. We have implemented simple, greedy heuristics to partition programs into tasks. The heuristics can be augmented by user hints to enable evaluation of different partitioning schemes. After the program is partitioned into tasks, the compiler performs optimizations specific to the Multiscalar architecture like loop restructuring. Register allocation and instruction scheduling is performed after this phase in the usual manner. At the final code generation phase, the compiler annotates the assembly code with inter-

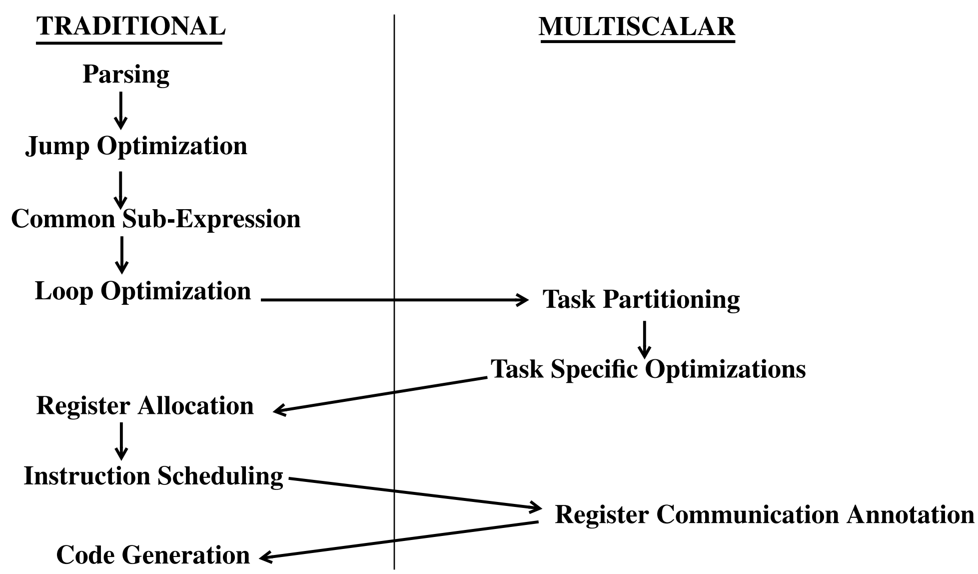

Figure 7: Organization of the compiler for the Multiscalar architecture.

Benchmark Program	*Benchmark Input*	*Dynamic Instructions*	*Total cycles in-order*	*Total cycles out-of-order*
compress	in	70.24 mil	98.17 mil	87.05 mil
xlisp	7 queens	222.48 mil	313.76 mil	266.53 mil
gcc	integrate.i	69.39 mil	86.17 mil	75.83 mil

Table 1: Base case (End_Send) statistics of benchmarks.

task register communication and control flow information. In this paper we restrict our attention to the parts of the compiler that deal with register communication.

5.2 Results

All experiments were run with the simulator configured as a Multiscalar processor with 4 processing units. The compiler was configured to produce tasks with at most 4 targets. The control flow predictor used a path based scheme which selects from 4 targets per prediction and maintains a 15 bit path history register indexing into 32k entries. In addition, it includes a 64 entry return address stack. Each processing unit was configured with 32k of 2-way set associative instruction cache in 64 byte blocks with 1-cycle hits and 10-cycle misses. The data cache was 8-way interleaved on the low order bits of the block address, for a total of 64k of direct mapped storage in 16 byte blocks with 2-cycle hits and 10-cycle misses. Each 8k bank of the interleaved data cache was configured with a 32 entry address resolution buffer[13] to handle memory disambiguation. Both loads and stores were non-blocking. A 128-bit wide split transaction bus connects all caches to the memory. Each processing unit can send at most one register value per cycle to the next processing unit, with a transfer latency of one cycle. Each processing unit was given 1-way, out-of-order or in-order issue characteristics and a 32 entry re-order buffer.

The performance achieved by the various schemes described in Section 3 depends upon many factors: average task size, overhead of extra release instructions, and effectiveness of each scheme in avoiding register stalls to name the most significant. Table 1 identifies the benchmarks[1] used in this study and reports the performance for End_Send. All benchmarks were compiled with the -O2 level of optimization flag. Profiling was performed to provide basic block frequency counts, as needed (with different profile and evaluation inputs). Table 2a shows the average dynamic size of the tasks executed in each of the benchmarks. Table 2b reports the overhead of release instructions inserted by the compiler (for Last_Send and Spec_Send). Figure 8 and Figure 9 show the speedups obtained by the different register communication strategies, Eager_Send, Last_Send, and Spec_Send over the base strategy of End_Send, for 4 processing units with 1-way out-of-order and in-order issue characteristics, respectively

In Table 1, he columns titled “Total Cycles” contain the number of cycles taken to execute each of the benchmarks for the given input for a Multiscalar processor of 4 processing units with 1-way in-order and out-of-order issue characteristics. Since End_Send sends all the ModSet registers at the end of each task, subsequent tasks must wait for register values incurring heavy performance loss. The out-of-order issue configuration performs better (13%-17%) than the in-order configuration due to its ability to tolerate the delays in register communication for End_Send.

1. To the referees: we intend to simulate all of the SPEC 92 and SPEC 95 benchmarks.

Table 2a indicates that, on average, the tasks of these benchmark programs are fairly small. In particular, xlisp is

Benchmark Program	*Dynamic task size*
compress	14.37 instrs
xlisp	7.66 instrs
gcc	12.30 instrs

Table 2a: Average task size.

Benchmark Program	*Number of releases*
compress	2.7%
xlisp	3.9%
gcc	2.3%

Table 2b: Overhead of releases for Last_Send.

smaller than the others due to frequent function calls in the program. This case is the result of the current greedy task selection heuristic used by the compiler which partitions tasks at function call sites. Table 2b shows the number of release instructions executed by each benchmark program, for Last_Send, as a percentage of the total number of dynamic instructions executed. The overhead due to release instructions depends on the size of the tasks, since a larger task encapsulates more register live ranges than a smaller one, reducing the amount of communication performed in the program. To further reduce overhead, these runs also include the dead register optimization discussed in Section 3.4. Overall, avoiding extra release instructions by annotating existing instructions and eliminating the communication of dead registers are effective means of keeping this execution overhead meager.

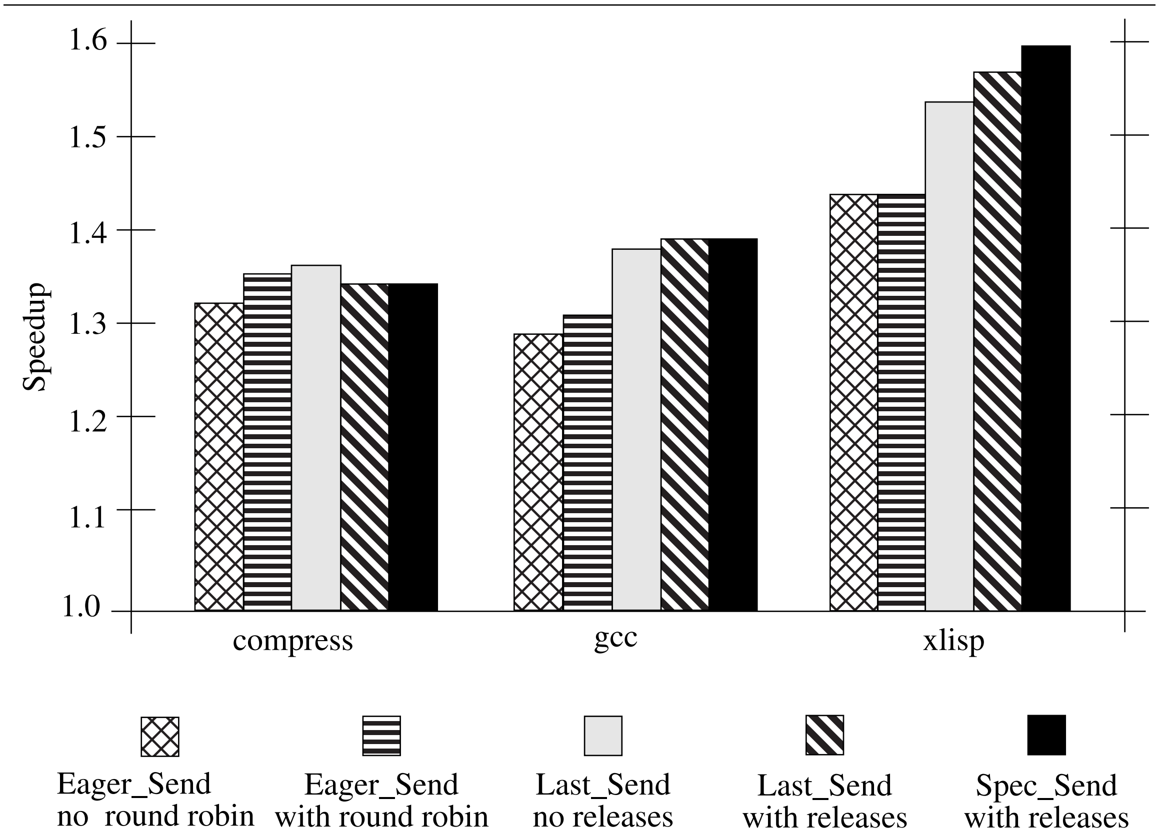

Figure 8: Speedups for out-of-order processing units.

Figure 8 and Figure 9 indicate speedups of the various register communication strategies with 1-way out-of-order and

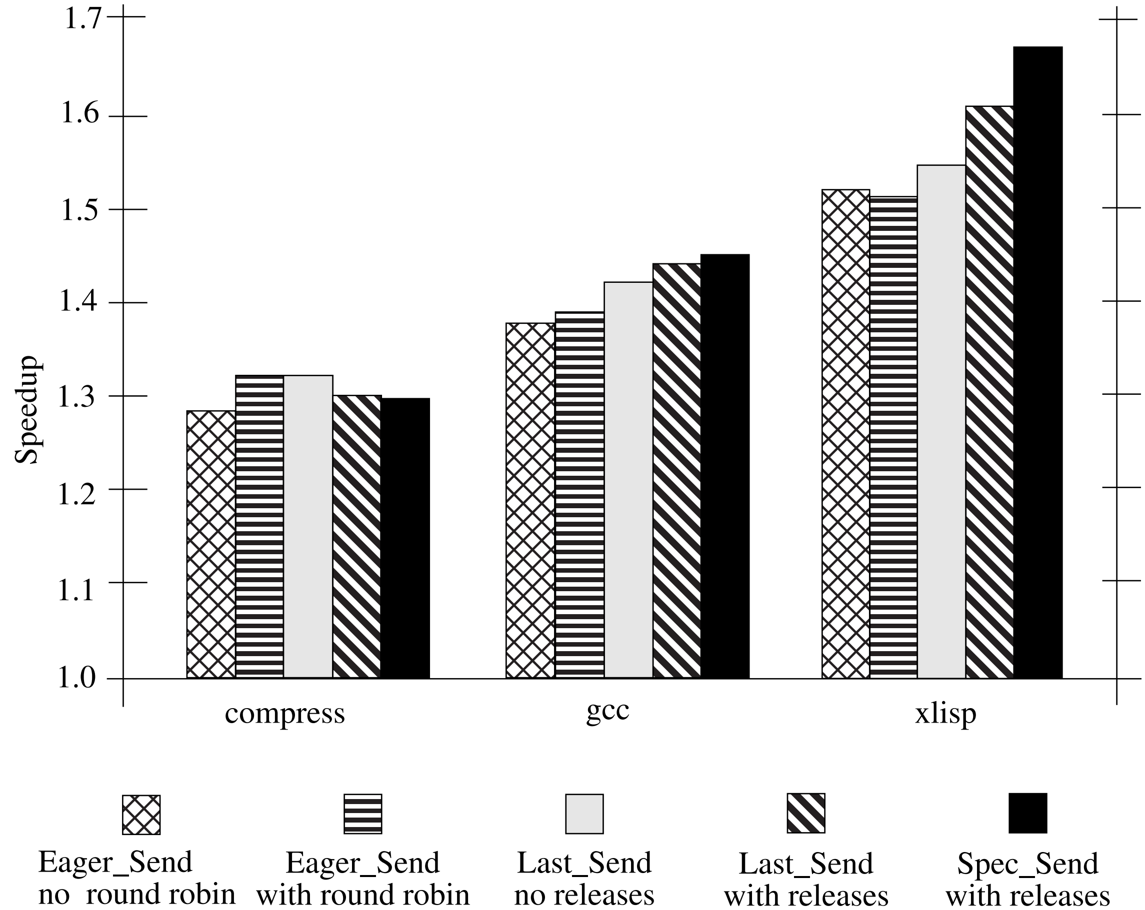

Figure 9: Speedups for in-order processing units.

in-order processing units, respectively. In order from left to right, the bars show speedups for Eager_Send without round robin register assignment, Eager_Send with round robin assignment, Last_Send without explicit release instructions, Last_Send with explicit release instructions, and Spec_Send with explicit release instructions. A stack-like register assignment (which is the default in gcc) is done if round robin assignment is not used. Last_Send without explicit releases is compiled by annotating instructions with forward bits only; any release of registers is done implicitly by the hardware at the end of task execution. For Spec_Send, the value of SpThreshold was placed at 0.9.

Considering out-of-order runs first, the best performance is achieved only when compiler assist is applied. In particular, changing the register assignment scheme to round robin gives an improvement of 4% in compress and 2% in gcc over Eager_Send without round robin assignment. For both gcc and xlisp, the biggest improvement in performance is seen when the compiler generates explicit communication (as in Last_Send) instead of the hardware sending each register every time it is defined (as in Eager_Send). Though compress improves by only 1%, gcc and xlisp improve by as much as 8% and 9%, respectively over Eager_Send with round robin. Last_Send avoids the squashes incurred with Eager_Send by being conservative, implying that it may be better to stall for register values than to speculatively send the values earlier but incur squashes on incorrect speculation. The use of explicit releases in Last_Send contribute another 1% for gcc and 3% for xlisp. In compress, however, the overhead of release instructions offsets any improvement in register communication resulting in performance loss of about 2% over Last_Send without releases. Finally, using Spec_Send improves xlisp by 3% over Last_Send due to the fact that the compiler is able to employ controlled register data speculation so as to avoid the negative effect of excessive squashes for uncontrolled, as in Eager_Send. Overall, compiler assist contributes performance improvements of 6% in compress, 10% in gcc and 12% in xlisp, over Eager_Send without round robin assignment. These improvements are significant considering the capability of latency-tolerant out-of-order processing units.

The in-order configuration follows the trends of the out-of-order configuration closely. The use of round robin register assignment boosts the performance of Eager_Send by 6% in compress and 1% in gcc. However, it degrades performance in xlisp by 1%. Though Last_Send without releases does not have a noticeable effect on compress, both gcc and xlisp improve by 4%. The use of release instructions contributes another 2% to gcc and 6% to xlisp, respectively. As was the case in the out-of-order configuration, releases cause a degradation of performance in compress by 2%. Using Spec_Send contributes another 1% to gcc and 6% to xlisp with no noticeable impact on compress. Again, the controlled register data speculation of the Spec_Send strategy provides better performance than the uncontrolled register data speculation of Eager_Send. Overall, compiler assist contributes performance improvements of 6% in compress, 6% in gcc, and 13% in xlisp, over Eager_Send without round robin assignment. These improvements indicate that substantial performance improvements are possible even for latency-intolerant in-order processing units.

6 Summary

Critical paths in program execution for a Multiscalar processor often involve the communication of register values produced by one task and consumed by another. Accordingly, register data communication represents a critical factor in an implementation of the Multiscalar architecture. In this work, we considered two aspects of this important issue: correctness and performance. To specify correctness criteria as per sequential semantics, we described an abstract model of Multiscalar register communication. To achieve high performance, we developed a progression of hardware and compiler techniques that reduce the communication delay implied by the abstract model, sometimes using register data speculation for this purpose. In addition, we discussed the analyses the compiler has to carry out and the information that it has to convey to the hardware to realize a range of register communication strategies.

We evaluated the practicality and the effectiveness of the techniques by implementing them in the GNU C compiler and our detailed simulator of a Multiscalar processor. We simulated on our cycle-by-cycle simulator the execution of a number of the SPEC92 benchmarks compiled by our compiler to evaluate the strategies we developed; we compared these strategies to a base case in which register communication is performed at the end of task execution. The experimental results indicate that aggressive hardware with no compiler support to detect the last modification of a register within a task improves performance by 28% to as much as 53% for in-order units and 30% to 45% for out-of-order units. The addition of static compiler analyses that detects the last modification of registers and informs the hardware through annotation of existing instructions and insertion of extra instructions (where needed) improves performance by 32% to as much as 67% for in-order processing units and 36% to 59% for out-of-order processing units.

As a result of this work, we make two key observations. First, we have found that register data speculation can be an important component in a high performance register communication mechanism for a Multiscalar processor. However, if such register data speculation is used in an uncontrolled fashion, gains in performance may be eroded by losses due to incorrect execution. On the other hand, controlled register data speculation can avoid this pitfall, but requires accurate information about the expected characteristics of program execution. Second, we have found that conveying information from the software to the hardware is a valuable tool to improve register communication. However, if providing such information involves inserting extra instructions, it must be done carefully. While annotating existing instructions does not affect the critical path through a program, inserting extra instructions likely does. Consequently, it is possible that such a decrease in register communication delay time may not be enough to offset a likely increase in the critical path.

Acknowledgements

This work was supported in part by NSF Grants CCR-9303030 and MIP-9505853, ONR Grant N00014-93-1-0465, and by 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.

References

[1] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Conference Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 58–67. Association for Computing Machinery, May 1992.

[2] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin-Madison, November 1993.

[3] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Conference Proceedings of the 22nd Annual International Symposium on Computer Architecture*. Association for Computing Machinery, June 1995.

[4] S. Breach, T. Vijaykumar, and G. Sohi. The anatomy of the register file in a multiscalar processor. In *Conference Record of the 25th Annual International Symposium on Microarchitecture*, pages 181–190, San Jose, CA, November 1994. Association for Computing Machinery.

[5] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[6] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for fortran d on mimd distributed-memory machines. In *Conference Proceedings of the International Conference on Supercomputing*, July 1992.

[7] C. Koelbel, D. Loveman, R. Schreiber, J. G. Steele, and M. Josel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[8] M. Smith, M. Lam, and M. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Conference Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 344–354. Association for Computing Machinery, May 1990.

[9] P.-T. Hsu and E. Davidson. Highly concurrent scalar processing. In *Conference Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 386–395. Association for Computing Machinery, June 1986.

[10] D. Pnevmatikatos and G. Sohi. Guarded execution and branch prediction in dynamic ilp processors. In *Conference Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 120–129. Association for Computing Machinery, April 1994.

[11] S. Mahlke, W. Chen, W. Hwu, B. Rau, and M. Sclansker. Sentinel scheduling for VLIW and superscalar processors. In *Conference Proceedings of the Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 238–247. Association for Computing Machinery, October 1992.

[12] B. Rau, M. Schlansker, and P. Tirumalai. Code generation schema for modulo scheduled loops. In *Conference Record of the 25th Annual International Symposium on Microarchitecture*, pages 158–169, Portland, OR, December 1992. Association for Computing Machinery.

[13] M. Franklin and G. S. Sohi. ARB: A hardware mechanism for dynamic memory disambiguation. *IEEE Transactions on Computers, forthcoming*.