Register Connection: A New Approach to Adding Registers into Instruction Set Architectures

Tokuzo Kiyohara
Media Research Laboratory
Matsushita Electric Industrial Co., Ltd.
Kadoma-shi, Osaka, 571 Japan

Scott Mahlke, William Chen, Roger Bringmann
Richard Hank, Sadun Anik, Wen-mei Hwu
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

Code optimization and scheduling for superscalar and superpipelined processors often increase the register requirement of programs. For existing instruction sets with a small to moderate number of registers, this increased register requirement can be a factor that limits the effectiveness of the compiler. In this paper, we introduce a new architectural method for adding a set of extended registers into an architecture. Using a novel concept of connection, this method allows the data stored in the extended registers to be accessed by instructions that apparently reference core registers. Furthermore, we address the technical issues involved in applying the new method to an architecture: instruction set extension, procedure call convention, context switching considerations, upward compatibility, efficient implementation, compiler support, and performance. Experimental results based on a prototype compiler and execution driven simulation show that the proposed method can significantly improve the performance of superscalar processors with a small or moderate number of registers.

1 Introduction

Designing high-performance processors often involves exploiting instruction-level parallelism (ILP). An example of such an approach, pipelining, has been widely used, and many pipelined designs are capable of executing nearly one instruction per cycle. Further performance improvement can be achieved either by executing more than one instruction per cycle, or by increasing the depth of pipelining. Superscalar and VLIW processors fetch, decode, and execute more than one instruction per cycle by providing multiple functional units and datapaths. Superpipelined processors divide the pipeline into smaller segments that have less delay, allowing the clock cycle to be shortened and more instructions to overlap with each other.

In order to assist the hardware to achieve performance objectives, compilers for superscalar, VLIW, and superpipelined processors use optimization and code scheduling techniques to exploit ILP. The code optimization techniques for these processors create additional temporary variables to eliminate data and control dependences among instructions. The code scheduling techniques reorder instructions so that instructions that are close to each other tend to be independent of each other. Both can greatly improve the effectiveness of a processor that exploits ILP. However, they also tend to increase the number of variables that are simultaneously live at each point of program execution. For instruction set architectures with small register files, such as the Intel i80X86 and the Motorola 680X0, with 8 and 16 registers respectively, these simultaneously live variables cannot be accommodated in registers. As a result, some of the variables have to be *spilled* to memory. Extra memory loads must be executed before using these spilled variables and extra stores must be executed after modifying them. Spilling tends to add to the latency of computation and consume memory access bandwidth, which reduces the effectiveness of the optimization and scheduling techniques.

A straightforward solution to a shortage of registers is to increase the number of registers in the instruction set architecture. However, major difficulties exist with this approach. In the case of designing a new instruction set, the number of bits required to select among registers may be too large for a given instruction format. For thirty two bit instruction formats, supporting more than thirty two registers imposes a strict limit on the number of bits available to opcodes and constants. For existing architectures, the sizes of the opcodes and constants are already fixed, leaving no room for indexing into an enlarged register file.

In this paper, we introduce a method referred to as *Register Connection (RC)* to add a set of *extended registers* to an architecture. We will refer to the registers in the original architecture as *core registers*. The novel aspect of the RC method is that rather than explicitly moving data between the core and the extended registers, it specifies a small set of opcodes to dynamically *connect* the register indices to a large set of registers. When a register index is connected to a register, all accesses using the register index are automatically directed to the appropriate register of the enlarged register file.

The concept of connecting registers without data movement enables an efficient implementation of our proposed method. The basic idea is to have all the processor function units directly access a enlarged register file. A translation is performed by keeping track of the connection between the addressable registers in the instruction set and the larger number of registers available in the architecture. With this translation, instructions with small register indices end up accessing the large register file before they are issued into the function units. By systematically scheduling the connect instructions, one can achieve a performance level similar to that with a large register file for an architecture with RC support.

Although the basic idea of RC is simple, there are important technical issues involved in applying RC to an instruction set. The rest of this paper is structured to address these issues. Section 2 describes the architectural support for RC. Section 3 discusses register allocation issues. Section 4 addresses upward compatibility. Section 5 reports experiments on the performance advantage of RC using a prototype compiler and execution-driven simulation. Finally, concluding remarks are offered in Section 6.

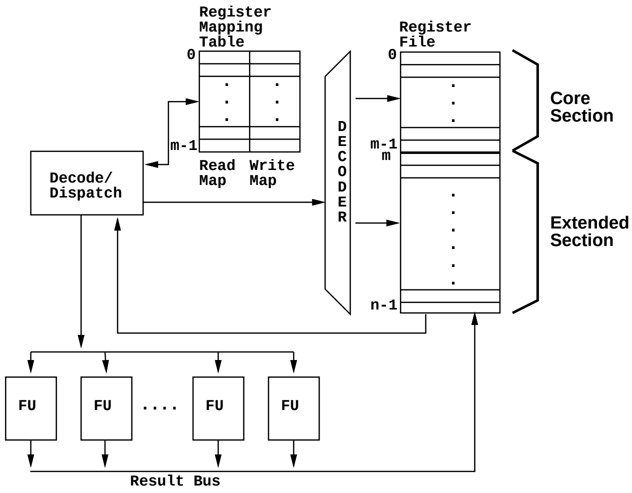

Figure 1: Superscalar processor supporting register connection.

2 Architecture Support for Register Connection

2.1 Design overview

RC requires several architectural extensions to increase the number of registers in existing instruction sets. The base architecture to which these extensions are applied is a generic, pipelined, superscalar processor with an m-entry register file. A summary of the changes to the base architecture to support RC is presented in Figure 1. First, the base register file is replaced by an n-entry register file, $n > m$. The enlarged register file consists of two logical components, the core section and the extended section. The core section contains the first m registers and corresponds to the original register file of the base architecture. The extended section contains the remaining $n - m$ registers that have been added to the architecture.

The second extension is the addition of an m-entry register mapping table. The register mapping table is used to map between the m addressable registers in the instruction set and the n registers available in the extended architecture. In order to provide for this mapping, register accesses in the base architecture are converted to indirect accesses through the register mapping table. Therefore, in the extended architecture, each register access consists of the following steps: a register number specified in the operand field of a machine instruction is used to index into the register mapping table. The register mapping table is then accessed to obtain the physical register number. Finally, the register file entry specified by the physical register number is accessed. The register mapping table used for RC is similar to the mapping used in the PDP-11 to map a smaller addressable memory space into a larger physical memory [1].

To make register mapping more flexible, each mapping entry contains both a read map and a write map. The read/write map specifies the physical register to be utilized when the register is specified as a source/destination register. Separate read and write maps allow more efficient use of a limited number of register mapping table entries. This flexibility becomes more important for smaller values of m in the base architecture.

The final extension is a modified decode/dispatch stage in the processor pipeline. Since registers are accessed indirectly, two accesses are required to fetch each register source operand of an instruction. First, the register mapping table is accessed to determine the physical register numbers for all registers utilized by the instruction. Second, the register file is accessed to obtain the register source operand values. A possible side effect of RC is an increased time to perform decode/dispatch. This may require an additional pipeline stage to perform decode/dispatch to prevent an increase in cycle time.

connect-use $Ri1$, $Rp10$
connect-use $Ri2$, $Rp7$
connect-def $Ri3$, $Rp6$
$Ri3 \leftarrow Ri1 + Ri2$

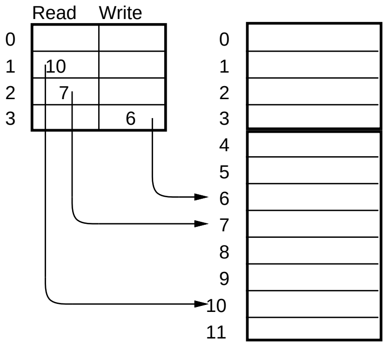

Figure 2: Connection instruction example.

2.2 Connection instructions

To utilize the enlarged register file, two instructions referred to as *connect-use* and *connect-def* are added to the instruction set. The *connect-use* and *connect-def* instructions change the register mapping information within the register mapping table. Both instructions take two input operands: a register mapping table index Ri and a register number Rp of the physical register file. *Connect-use* inserts the register number into the read map entry referenced by the register mapping table index. All subsequent reads using Ri are redirected to Rp. Similarly, *connect-def* updates the write map entry, and redirects subsequent writes using Ri to Rp.

The functionality of *connect-use* and *connect-def* can be illustrated with the code sequence in Figure 2. The core section has only four registers and the extended section adds another eight registers. The *connect-use* and *connect-def* instructions redirect the accesses made by the add instruction to the extended section of the register file. With the redirection, the add instruction will access $Rp10$ and $Rp7$ for its input operands and deposit its results into $Rp6$.

To reduce the number of connect instructions, it is possible to combine two connects into a single instruction provided the instruction size is large enough. There are three possible combinations: *connect-use-use*, *connect-def-use*, and *connect-def-def*. By incorporating these three new multiple-connect instructions instead of *connect-use* and *connect-def*, a more compact code schedule can be obtained. [1] The functionality of the multiple-connect instructions remain the same as *connect-use* and *connect-def*; however, the number of operands is four instead of two.

[1]For illustration purposes, the *connect-use* and *connect-def* model is used for clarity. However, for the experimental results, the *connect-use-use*, *connect-def-use*, and *connect-def-def* model is used.

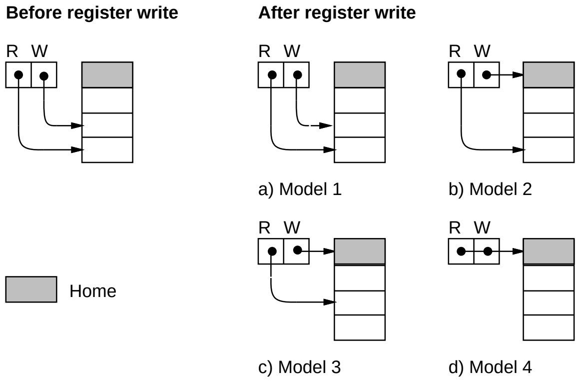

Figure 3: Four RC models. a) The register map is unchanged after a register write. b) The write map is reset to the home location after a register write. c) The write map is reset to the home location after a register write and the read map is replaced by the previous write map entry. d) Both the write and read maps are reset to the home location after a register write.

2.3 Alternative techniques for automatic register connection

In order to reduce the number of connect instructions, other instructions can be allowed to perform automatic register connection as a side effect. In this section, four alternative models of automatic register connection are discussed: (1) no reset, (2) write reset, (3) write reset with read update, and (4) read/write reset.

The first model provides for no automatic register connection. Therefore, the register mapping information may only be changed by explicit connect instructions. The other three models perform varying degrees of automatic register connection after an execution of a register write. In all models, only the register mapping table entry corresponding to the destination register is altered. Other strategies for automatic register connection for the source registers are possible; however, they are not considered in this paper. Figure 3 shows the updated register connection information after a register write for the four models.

Model two attempts to avoid an extra *connect-def* instruction by relocating the write map after a register write. When writing into a register pointed to by Rix, Rix_{write} (write map of Rix) is reset to Rpx (referred to as the *home location* of Rix) for subsequent writes. However, to read the written value, a subsequent *connect-use* for Rix is still required.

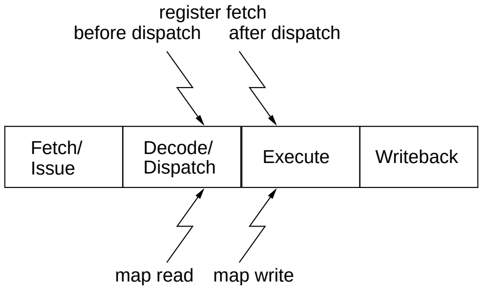

Figure 4: Example pipeline configuration with two variations.

In addition to adjusting the write map done with model two, model three also modifies the read map in an attempt to eliminate an extra *connect-use* instruction. In this model, when writing into a register pointed to by Rix, the automatic reset mechanism sets Rix_{read} to Rix_{write}, and Rix_{write} to Rpx. The automatic adjustment of connections provides the result of the execution for subsequent reads of Rix, and avoids the destruction of the data saved in Rix by subsequent writes of Rpx. Model three is chosen for implementation and performance simulation in this paper. The compiler algorithm to utilize this automatic reset model is discussed in Section 3.

Model four emphasizes the free use of the registers in the core section. When writing into a register pointed to by Rix, both the read map and the write map entries of Rix are set to Rpx. Future reads and writes of Rix are redirected to Rpx without extra connect instructions.

2.4 Zero-cycle execution latency of connect instructions

Since the RC mechanism does not require actual data movement, *connect-use* and *connect-def* can be implemented with zero-cycle execution latency. In order for these instructions to have zero-cycle latency, the implementation must allow them to affect the register accesses of instructions issued at the same cycle. This requires some forwarding logic to update the register accesses with the information contained in the connect instructions issued at the same cycle.

The forwarding that is performed varies slightly with the pipeline configuration. A simple four-stage pipeline to illustrate the necessary forwarding is shown in Figure 4 with two variations: register fetch is performed before dispatch or after dispatch. The register mapping table for both variations of this model is read late during the decode stage and updated at the beginning of the execute stage. Therefore, all connect instructions are ensured to update the register mapping table so that instructions in the next cycle can read the correct value. However, any instructions

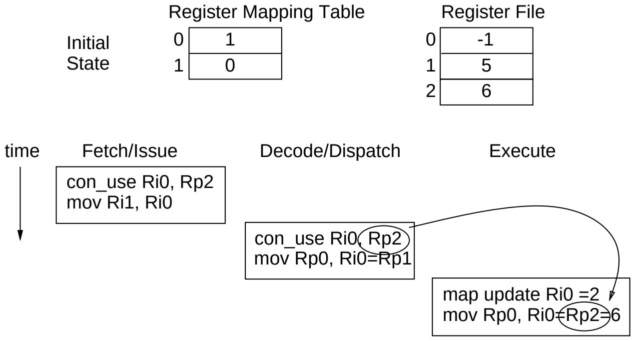

Figure 5: Example of forwarding when register fetch is performed after instruction dispatch.

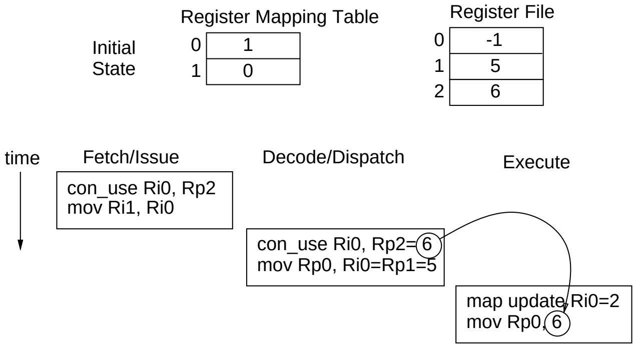

Figure 6: Example of forwarding when register fetch is performed before instruction dispatch.

which utilize connect instructions issued at the same cycle will obtain stale data from the register mapping table.

When register fetch is performed after dispatch, *connect-use* and *connect-def* must forward updated physical register numbers to other instructions during dispatch. Therefore, the correct physical registers are always either fetched during register fetch or available for writeback of the results. This forwarding is illustrated in the example shown in Figure 5. In this example, a 2-entry register mapping table and a 3-entry physical register file with the initial states shown in the figure are assumed. A *connect-use* which updates map location 0 is issued at the same time as a move instruction which utilizes map location 0 as a source operand. During decode, the move instruction reads the register mapping table, however stale data is obtained (Mapping table entry 0 contains a 1 rather than the desired value of 2). The *connect-use*, therefore, forwards the updated physical register number to the move instruction during dispatch, so the correct physical register contents are read during the execution stage.

When register fetch is performed before dispatch, the forwarding mechanism for *connect-def* instructions is not changed. However, *connect-use* instructions must forward the data value of the correct register to other instructions rather than the updated physical register numbers because register fetch has already been performed. The previous example illustrating the modified forwarding is shown in Figure 6. In the decode stage, the move instruction obtains the wrong data for $Ri0$ since map location 0 is stale. In order to properly forward the correct data to the move prior to execution, *connect-use* instructions are required to read the physical register contents which the read map is being set to. Thus in this example, *connect-use* reads the contents of $Rp2$ during the decode stage. During the dispatch stage, a simultaneous use of $Ri0$ is detected, and the correct data is forwarded from the *connect-use* to the move.

The timing constraints of these forwarding mechanisms must be addressed for any processor design. In the case of *connect-def*, there is likely sufficient time to perform to proper updates since the destination register is not required until a late stage in the pipeline. In the case of *connect-use* instructions, there is a more strict timing requirement. For this paper, it is assumed there is sufficient timing freedom prior to instruction dispatch to accommodate the necessary forwarding; however, this may not be the case for all implementations. In Section 5, the performance degradation in the case where an extra pipeline stage is required for RC support is evaluated.

2.5 Comparison with previous work

At the architecture level, the extended registers are similar to the T registers in the CRAY-1 architecture [2]. The similarity is that both techniques provide additional registers to hold more values than may be addressed in the instruction set. However, explicit data movement is necessary in the CRAY-1 architecture to utilize data in the T registers. In contrast, the RC method requires no explicit data movement to utilize data in the core and extended sections of the register file. Dynamic connection through the register mapping table is utilized to access both core and extended sections of the register file.

The register mapping table used in RC is similar to the mapping table used in the IBM RS/6000 [3]. In the RS/6000 the mapping table is utilized to perform dynamic register renaming. Each instruction accesses the mapping table to determine the appropriate physical register to access. In comparison, the RC method provides explicit instructions to modify the mapping table. Therefore, the extended registers are exposed to the code optimizer, code scheduler, and register allocator for use.

3 Register Allocation

The RC method requires a number of changes to the register allocation process [4]. The register allocator now has a much larger register file available with the addition of the extended registers. However, the decision to place a variable in a core register over an extended register depends upon the architectural model of register connection and the register allocation method chosen.

The register allocation method we chose attempts to place the most important variables into the core registers, while storing the less important variables in the extended registers or memory. This method is similar to the caller/callee save convention used in many compilers. The automatic reset RC models can naturally take advantage of this method since its reset mechanism can eliminate many connect instructions. The no-reset model treats all physical registers uniformly. However, if the number of active variables is larger than the number of mapping indices at any time, a large number of connect instructions may be required. The above allocation method automatically minimizes the number of connection instructions by maximizing the use of the core registers.

Once register allocation is complete, appropriate connect instructions must be inserted to enable instructions to access the variables allocated to extended registers. This can be accomplished by emulating the register mapping table and either selecting the index entry currently pointing to the physical register as its index or selecting the least important index as the new index. Consider the code sequence shown below that uses model three (Section 2.3). The core register file size is eight registers (R1-R8). Two variables have been allocated to extended registers, R9 and R10.

	connect_use Ri6,Rp9
1) R2 ← R2 + R9	1) Ri2 ← Ri2 + Ri6
2) R10 ← R3 + 1	*connect_def* Ri7,Rp10
3) R4 ← R10 + R5	2) Ri7 ← Ri3 + 1
	3) Ri4 ← Ri7 + Ri5

If we assume that the register maps for registers R1-R8 are currently pointing to their home locations, the code sequence requires two *connect instructions*. A *connect-use* is required prior to instruction 1 to allow reading of Rp9, in which case we use the register *read* map of Ri6. Also, the destination of instruction 2 was assigned to extended register Rp10, requiring a *connect-def* to set the register *write* map of Ri7 to Rp10. Note that a *connect-use* is not required prior to instruction 3 since the register *read* map of Ri7 is set to the register *write* map as a side affect of writing into the register. The selection of the register map entry used to access an extended register is arbitrary; however, with proper selection, the register allocator can attempt to minimize the artificial dependences introduced by these instructions and maximize the code motion opportunity available to the scheduler.

4 Upward Compatibility

One important reason for extending an architecture with the RC method instead of adding more registers to the operand fields in the instruction set, is to ensure upward compatibility with existing program binaries. On the surface, the RC method should trivially satisfy the upward compatibility requirement. Since the programs compiled for the original architecture will not contain any *connect-use* and *connect-def* instructions, all of the core registers will remain connected to their home locations throughout the program execution. The register access operations will operate as if there were no extended register file. However, there are three situations which must be addressed to completely ensure upward compatibility - subroutine calls, context switching, and trap and interrupt handling.

4.1 Subroutine calls

A typical approach to saving one of the extended registers across subroutine calls is to perform a *connect-use* to the extended register and then store the contents to memory. If the register is not subsequently reconnected to its core register, it is possible for the called subroutine to incorrectly access a register that it treats as a callee save register. For example, assume map entry 5 is connected to extended register 30 prior to the subroutine call so that it can be saved to memory. At the beginning of the called subroutine, core register 5 is saved since it is being treated as a callee-save register. Since the register is still connected to register 30, the wrong register contents are saved. Any subsequent write to register 5 will set the read map to correctly point to register 5. Prior to exiting the subroutine, register 5 must be restored from memory. Unfortunately, the contents of register 30 would be restored to register 5, introducing a possible program error.

This problem could be prevented by first performing a *connect-use* to register 5 and then saving its content. However, programs compiled for the original architecture cannot take advantage of this. It could also be prevented by requiring the calling subroutine to re-connect the core registers. In the worst case scenario, this could introduce one instruction for every core register. A more efficient solution to this problem is to make the *jsr* instruction also reset the map to point to the original core registers. The hardware to perform the reset of the register map is required by the architecture to ensure correct mapping of the core registers after power-up initialization.

A similar problem can also occur when returning from a subroutine. It is possible that a register is connected to an extended register to compute a returned value from the subroutine. Any *connect-use* instructions would be live across the subroutine return. Thus, reading a caller-save core register may actually access the extended set and introduce a program error. This problem can be eliminated by also requiring the *rts* instruction to reset the register map.

4.2 Context switches

A subtle issue of upward compatibility arises in the case of context switching. Programs compiled to use the RC extension require the connection information to be maintained across any context switch point. Therefore, core registers, extended registers and the connection information should be saved and restored. For programs compiled for the original architecture, only core registers need to be saved and restored, although saving and restoring extended registers and connection information would still result in correct operation. Therefore, there is an opportunity to avoid saving the extended registers and the connection information for programs compiled for the original architecture. This optimization would require a flag in the process status word to mark the program as either for the original architecture or for the extended architecture. The context switching routine can use this bit to choose different formats of the process context representation in the process control blocks.

4.3 Traps and Interrupts

Traps and interrupts are slightly more complicated than subroutine calls since they occur outside the control of the program. To permit access to registers, the method discussed for handling context switches can be used. However, traps and interrupts are typically used to implement time critical device drivers and perform instruction emulation. If any register is used, it must be saved and restored. However, to access the correct physical register, the map entry must also be saved, connected to the correct register and finally restored. The addition of the connect instructions could cause a severe performance penalty for device drivers that require few registers. A simple alternative to this approach is to bypass the register map for traps and interrupts. This can be accomplished by adding a register map *enable* flag to the processor status word. A trap or interrupt would disable this flag. Any subsequent register accesses would go directly to the core registers. The return from exception or interrupt condition will restore the original processor status word, which will automatically re-enable the register map.

If the trap or interrupt require more than the core registers, the register map can be re-enabled by writing to the processor status word. The register map

Instruction	Latency	Instruction	Latency
INT ALU	1	FP ALU	3
INT multiply	3	FP conversion	3
INT divide	10	FP multiply	3
branch	1/1-slot	FP divide	10
memory load	2 or 4	memory store	1

Table 1: Instruction latencies.

thus used must be saved, reset and restored prior to the return.

5 Experimental Results

5.1 Compiler support

In order to conduct meaningful experimental evaluation of the RC method, the required register allocation, code scheduling, and code generation support for RC have been implemented in the IMPACT-I compiler. All benchmark programs are compiled with full-scale classical and instruction-level parallelization code optimizations [5]. The register allocator uses a graph coloring algorithm that utilizes profile information in its priority calculations. All compiler optimizations are verified by executing the output code on a DEC-3100 workstation.

For the original architecture, the compiler generates spill code needed to access variables spilled out to memory. For the extended architecture, the compiler manages the register file through the register mapping table and generates *connect-use* and *connect-def* instructions to access variables in the register file. In addition, the compiler generates save and restore code for the registers at procedure call interfaces. The code scheduler is designed to take advantage of the zero-cycle latency of the connect instructions as illustrated in Section 2.4. For all core register file sizes, four integer registers are reserved as spill registers and one integer register is reserved for Stack Pointer.

5.2 Architecture assumptions

The instruction set used in all experiments is the MIPS R2000 instruction set extended with additional branch opcodes to allow general operand comparison and to facilitate static branch prediction. In the experiments, the size of the number of core integer registers is varied from 8 to 64, and the number of the core floating-point registers is varied from 16 to 128 to study the effect of the register file size. Double precision floating point variables use two floating point registers.

In experiments with integer benchmarks, RC support is evaluated only for the integer register file while a fixed floating point register file of 64 entries is assumed. Conversely, for experiments with floating point benchmarks, RC support is evaluated only for the floating point register file while a fixed integer register file of 64 entries is assumed. Furthermore, in experiments with RC support, the register file is assumed to contain a total of 256 registers with the size of the core section specified in the experiment. The size of the extended section is therefore the difference between the core size and 256. In experiments without RC support, the register file contains only the specified number of core registers.

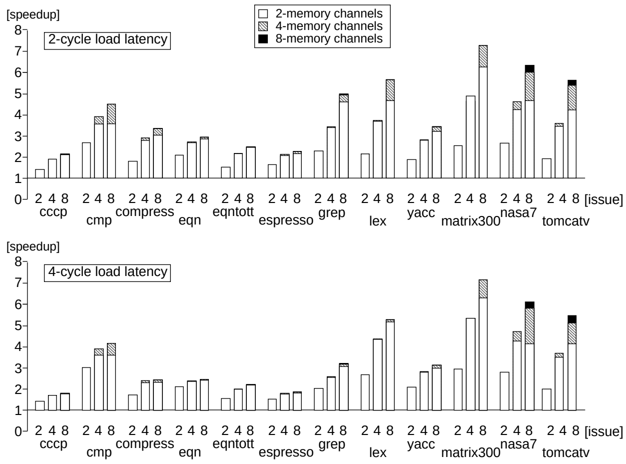

Figure 7: Speedup for processors with an unlimited number of registers, varying issue rates and memory channels.

The underlying microarchitecture is assumed to have deterministic instruction latencies (see Table 1) and CRAY-1 style register interlocking [2]. Given an issue rate, all combinations of instruction patterns are allowed to be executed in parallel assuming homogeneous pipelined function units. The only exception is that memory accesses are restricted to a subset of the function units in the experiments. For 2-issue and 4-issue models, there are two memory channels and in the 8-issue model there are four memory channels.

5.3 Results

The performance of the RC mechanism is evaluated using nine integer and three floating-point benchmarks. The integer benchmarks are *cccp, cmp, compress, eqn, eqntott, espresso, grep, lex* and *yacc,* and the floating-point benchmarks are *matrix300, nasa7* and *tomcatv.* The execution time of each benchmark, assuming a 100% cache hit rate, is derived using execution-driven simulation. The base configuration for the speedup calculations is a single-issue processor with an unlimited number of registers using conventional compiler scalar optimizations.

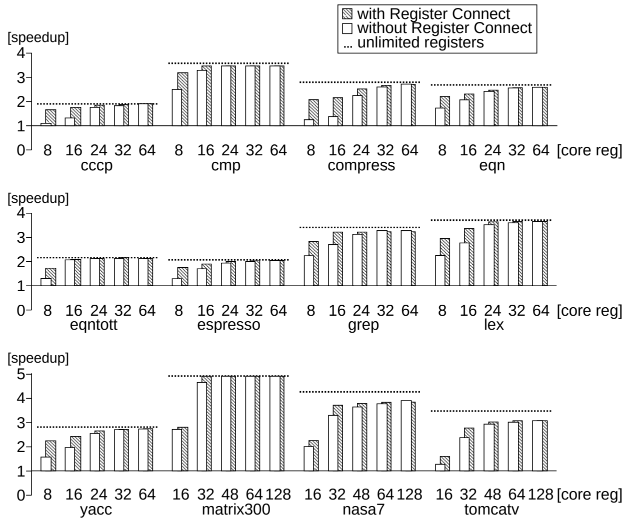

Figure 8: Speedup for a 4-issue processor with 2-cycle load latency and varying number of core registers.

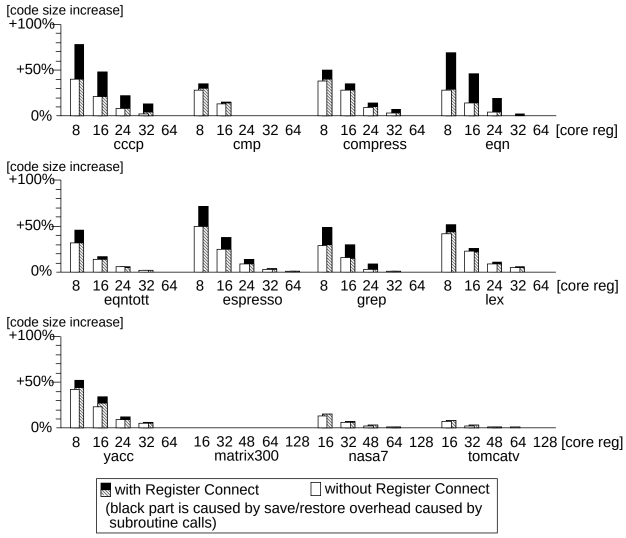

Figure 9: Percentage of code size increase due to spill code for a 4-issue processor with 2-cycle load latency and varying number of core registers.

The speedup for processors with unlimited registers, varying issue rates and memory channels is shown in Figure 7. The goal of the RC method is to approach the unlimited register performance with the combination of a small number of core registers.

Effect of the number of core registers

Figure 8 shows the benefit of the RC method for processors with varying numbers of core registers. The white bars show the speedup of the model without RC support, (referred to as the without-RC model). The shaded bars show the speedup of the model with RC support, (referred to as the with-RC model). All results in this figure assume a 4-issue processor and 2-cycle load latency. The dotted lines show the speedup of the model with the unlimited number of integer registers.

For the integer benchmarks, 32 and 64 core registers for both with-RC and without-RC models achieve almost the same performance level as the unlimited register case. The performance degradation of both models starts in the 24-register case and becomes more severe in the 16-register case. For floating-point benchmarks, performance degradation starts around 32 registers and becomes more severe for 16 registers. All benchmarks run with a small number of core registers demonstrate a large performance advantage using the with-RC model over the without-RC model.

Figure 9 presents the percentage increase of code size after register allocation. The white bars show the percentage of code size increase for the without-RC model. The shaded/black bars show the percentage increase for the with-RC model. The black part corresponds to the percentage increase caused by save/restore of the extended registers before and after procedure calls.

As expected, with 32 and 64 core registers, the code size increase of both models for integer benchmarks is very small: approximately 10% or less. The code size expansion for both models starts with the 16 register case, which corresponds to the extra spill code or connect instructions inserted by the compiler. Although the code size increase of the with-RC model is significantly more than the without-RC model, the with-RC model achieves higher performance.

Effect of issue rate and load latency

Figures 10 and 11 illustrate the benefit of the RC method for different load latencies and instruction issue rates. The white bars show the speedup of the without-RC model with 16 core integer registers for the integer benchmarks and 32 core floating-point registers for the floating-point benchmarks. The shaded bars show the speedup of the with-RC model with the same number of registers. The dotted lines correspond to the speedup achievable using an unlimited number of registers.

The performance improvement due to the RC method is more significant for higher issue rates, especially in the 8-issue case. The RC method reduces the overhead caused by spill code. This overhead is attributed to the spill load latency and the scheduling restrictions imposed by dependences between spill registers. For higher issue rate processors, the impact of these two factors on the scheduled code and the instruction level parallelism is more significant. Fur-

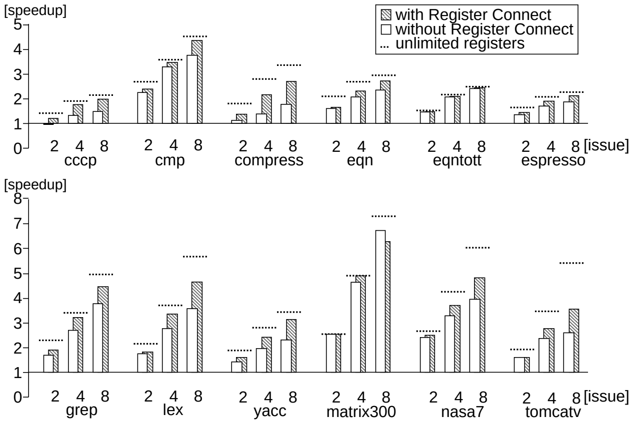

Figure 10: Speedup for 2-cycle load latency, 16 core integer registers for integer benchmarks, 32 core floating-point registers for floating-point benchmarks, and varying issue rate.

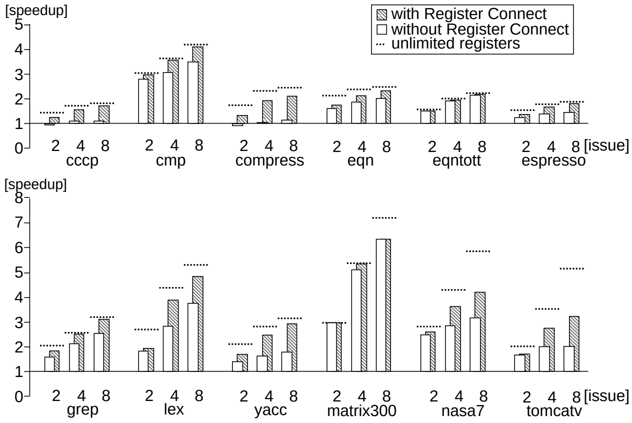

Figure 11: Speedup for 4-cycle load latency, 16 core integer registers for integer benchmarks, 32 core floating-point registers for floating-point benchmarks, and varying issue rate.

thermore, in the 8-issue case, the number of empty instruction slots at each clock cycle increases because of the limited instruction level parallelism in the benchmarks. This allows the compiler to hide the adverse effects of the code expansion due to the RC method.

The shorter load latency increases the efficiency of the spill code, so the performance improvement achieved by the RC method is less for two cycle load latency than for four cycle load latency. Nevertheless, there is sizable benefit for both latencies.

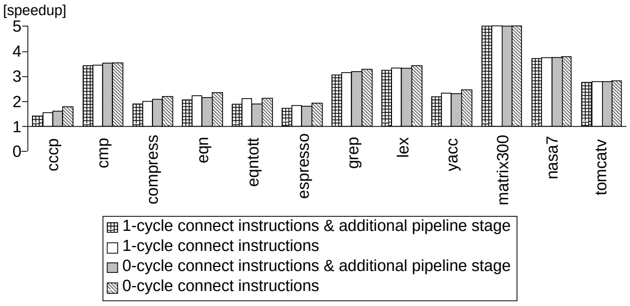

Figure 12: Speedup for an 4-issue processor, 2-cycle load latency, 16 core integer registers for integer benchmarks, 32 core floating-point registers for floating-point benchmarks, and varying architecture support and pipeline implementation.

Effects of different RC implementations

Figure 12 compares the performance of four possible implementation scenarios for a 4-issue processor with 2-cycle load latency. The most efficient implementation considered is zero-cycle latency connect instructions implemented within an existing processor pipeline. The zero-cycle latency connect instruction and additional pipeline stage scenario considers adding a pipeline stage for accessing the register mapping table, and implementing forwarding to instructions issued in the same cycle. The one-cycle latency implementation of the connect instruction does not require forwarding. Similarly, the additional pipeline stage is evaluated with one-cycle latency connect instructions. The results show that there is very little performance loss when the RC method cannot be implemented within an existing pipeline with zero-cycle latency. This makes the RC method a feasible improvement even for high speed implementations.

Effect of a limited number of memory channels

The number of memory channels can greatly affect the processor implementation cost. Figure 13 shows the effect of increasing the the number of memory channels from two to four for a 4-issue processor with 2 and 4-cycle load latency. The white bars show the speedup of the without-RC model and the shaded bars show the speedup of the with-RC model for two memory channels. The upper solid bars show the additional gain in speedup by increasing the memory channels to four for the without-RC model. The dotted lines show the speedup with an unlimited number of registers and two memory channels.

Figure 13 shows that for a 4-issue processor, the benefit of increasing the number of memory channels from two to four is much less than the benefit of implementing the RC method for two memory channels. This demonstrates that the RC method improves performance not only by reducing the frequency of memory accesses but also by providing a more efficient mechanism.

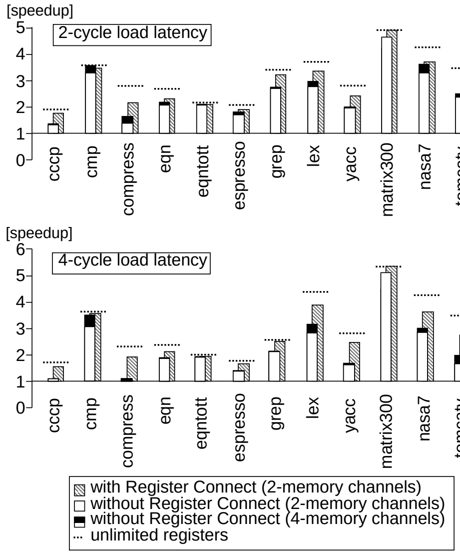

Figure 13: Speedup, varying the number of memory channels and register rename channels, for 4-issue processor with 2 and 4-cycle load latency.

6 Conclusion

The effectiveness of compilers for superscalar and superpipelined processors can be limited by the size of the register files in existing architectures. In this paper, we have introduced the Register Connection (RC) method to add a large number of registers into an architecture. We have shown that the RC method does not require any change to the format of existing instructions. It allows the compiler to take advantage of a large register file which can be conveniently accessed through a small register mapping table. We have also addressed the issues regarding procedure calls and context switches. Overall, the RC method can be added to an existing architecture in an upward compatible manner.

An implementation of the RC method has been described. By overlapping the execution of the connect instructions with the instruction dispatch logic in superscalar processors, one can achieve zero-cycle effective execution latency for the connect instructions. As a result, the connect instructions can affect the immediate subsequent instructions issued in the same clock cycle. This makes it extremely inexpensive for instructions to access the extended register file.

Experimental evaluation shows that the RC method improves the performance of superscalar processors with 16 or fewer core registers. The performance improvement increases with the issue rate. A four-issue processor with 16 core integer registers and 240 extended registers, and a 2 cycle load latency can achieve 90% of the performance of an equivalent processor with an unlimited number of core registers. This performance result shows that the RC method is a very promising technique to extend existing instruction set architectures for high performance superscalar implementation. As new code parallelization methods become available, we expect that the RC method will become beneficial for architectures with 32 or more registers.

Acknowledgements

The authors would to thank John Gyllenhaal and Grant Haab, along with all members of the IMPACT research group for their comments and suggestions. Special thanks to the anonymous referees whose comments and suggestions helped to improve the quality of this paper significantly. This research has been supported by JSEP under Contract N00014-90-J-1270, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, Matsushita Electric Industrial Co. Ltd., Hewlett-Packard, and NASA under Contract NASA NAG 1-613 in cooperation with ICLASS. Scott Mahlke is also supported by fellowship provided by Intel Foundation.

References

[1] Digital Equipment Corporation, Marlboro, Massachusetts, *Microcomputers and Memories*, 1982.

[2] R. M. Russell, "The Cray-1 computer system," *Communications of the ACM*, vol. 21, pp. 63–72, January 1978.

[3] G. F. Grohoski, "Machine organization of the IBM RISC System/6000 processor," *IBM Journal of Research and Development*, vol. 34, pp. 37–58, January 1990.

[4] G. J. Chaitin, "Register allocation and spilling via graph coloring," in *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction*, pp. 98–105, June 1982.

[5] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellete, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, January 1993.