A Fill-Unit Approach to Multiple Instruction Issue

Manoj Franklin
Dept. of Elect. and Computer Eng.
Clemson University
Clemson, SC 29634
mfrankl@eng.clemson.edu

Mark Smotherman
Dept. of Computer Science
Clemson University
Clemson, SC 29634
mark@cs.clemson.edu

Abstract

Multiple issue of instructions occurs in superscalar and VLIW machines. This paper investigates a third type of machine design, which combines the advantages of code compatibility as in superscalars and the absence of complex dependency-checking logic from the decoder as in VLIW. In this design, a stream of scalar instructions is executed by the hardware and is simultaneously compacted into VLIW-type instructions, which are then stored in a structure called a shadow cache. When a shadow cache line contains the instructions requested by the fetch unit, the scalar instruction stream is preempted and all operations in the shadow cache line are simultaneously issued and executed. The mechanism that compacts instructions is called a fill unit, and was first proposed for dynamically compacting microoperations into large executable units by Melvin, Shebanow, and Patt in 1988. We have extended their approach to directly handle data dependencies, delayed branches, and speculative execution (using branch prediction). This approach is evaluated using the MIPS architecture, and a six-functional-unit machine is found to be 52 to 108% faster than a single-issue processor for unrecompiled SPECint92 benchmarks.

Keywords: instruction-level parallelism, multiple operation issue, superscalar, VLIW.

1 Introduction

Multiple instruction issue is one way to exploit instruction-level parallelism and improve the performance of uniprocessors. Two basic approaches to multiple issue are superscalar and VLIW. This paper investigates a third approach, namely the use of a hardware assist to dynamically group instructions that can be issued together into long words; this allows later fetch of these long words and the multiple issue of the grouped instructions the next time the same piece of code is executed.

1.1 Approaches to Multiple Issue

In the superscalar approach, multiple instruction issue is an implementation feature and not an instruction set architecture (ISA) feature. The hardware fetches multiple instructions from a sequential instruction stream, decodes them in parallel, performs dynamic scheduling of the ready instructions, and attempts to issue multiple ready instructions every cycle. The advantage of this approach is that it provides code compatibility between all implementations of a particular instruction set (i.e., single-issue and different multiple-issue processors).

An ideal superscalar implementation establishes a large window of instructions among which to look for exploitable parallelism, and on each cycle performs dynamic scheduling (out-of-order execution) within the established window. Examples of this type of design can be found in the central window designs of [7] [12] and the Metaflow Lightning SPARC design [11]. However, many current superscalar implementations are more limited. The IBM Power-2, for example, decodes a maximum of eight instructions per cycle, dispatches up to six instruction per cycle, and can perform only a limited amount of out-of-order issue [13]. Approaches directed towards high clock rates are often more restricted in window size and order of issue; examples are the DEC Alpha 21064 and 21164, which decode two and four instructions per cycle, respectively, and provide only in-order issue.

In the VLIW approach, the compiler performs static scheduling and packs independent operations together as long word instructions. This eliminates the need for hardware dependency checking and makes a

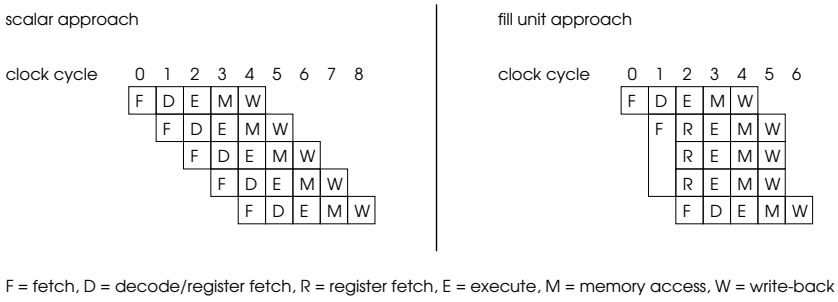

Figure 1: Fill unit operation.

tradeoff of building a relatively more complex compiler to exploit parallelism on relatively simple hardware. The VLIW instruction format exactly represents the functionality and number of resources on a given implementation, and thus code generated by the compiler is implementation-dependent; a different compiler scheduler and code generator must be provided for each distinct VLIW implementation. The Multiflow TRACE is an example VLIW machine with different models having either 14 or 21 operations placed together in one VLIW instruction.

Optimizing compilers targeted to specific implementations will continue to be an important part in gaining maximum performance for given programs (e.g., numerical codes). Although compiler-processor synergism has great potential, real-life systems include local networks of computers that are heterogeneous among implementations (even if homogeneous according to processor brand name) and cannot afford to store a multitude of processor-specific versions of the operating system and important applications. Rather, the file server will likely contain one generically-optimized binary version of each executable program. In this kind of environment, code compatibility will remain an important economic fact of life, and VLIW processors will not be a good choice.

1.2 The Fill Unit Approach

A possible third option to provide multiple issue, in-between the two extremes, is to provide hardware that monitors the instruction stream and groups multiple instructions into long-word instructions within the processor itself. This would allow a sequential instruction stream to be fed to the processor, with execution of parts of the code accelerated by wide issue whenever possible. At the same time it would preserve code compatibility. This is the approach pursued in this paper.

In 1988, Melvin, Shebanow, and Patt proposed a hardware assist to compact microoperations generated from sequentially-fetched instructions into a decoded instruction cache [10]. Their design was called a fill unit, and it worked as follows: an instruction prefetch buffer fetched instructions from memory or the normal instruction cache, and fed the corresponding microinstructions to the fill unit, which collected them together and placed them as a "multinode word" in a decoded instruction cache. Microoperation references were renamed into the address space of the decoded instruction cache, and the microoperations from a single instruction could be split across two multinode words. Filling stopped (i.e., a multinode word was "finalized") whenever a branch was encountered or no empty microoperation slots remained in the filled word. The microinstructions in a multinode word could have data dependencies, which would be resolved later by underlying dynamic scheduling hardware. The purpose of the fill unit in the original proposal was to give a larger piece of atomic work to the dynamic scheduler. A checkpoint-repair system was used for recovery [6], so that the machine could back up to the appropriate checkpoint and run in sequential mode whenever an exception occurred. Their proposal also included renaming of microarchitecture registers to encode forwarding requirements within and between multinode words, and it renamed ISA-level registers to improve performance. Branch prediction was proposed as a sequencing mechanism, separate from the fill unit. No experimental results were published, however.

This paper extends the fill unit concept to grouping RISC instructions that are independent or dependent in constrained ways into long words for later issue. This is in contrast to the original proposal of grouping arbitrarily-dependent microoperations into large units. No renaming is necessary for the instruction addresses, and the allowed dependencies are directly handled using no assumptions of underlying hardware beyond simple interlocks and cascaded functional units.

The paper is organized as 5 sections. The introduction has reviewed superscalar and VLIW concepts

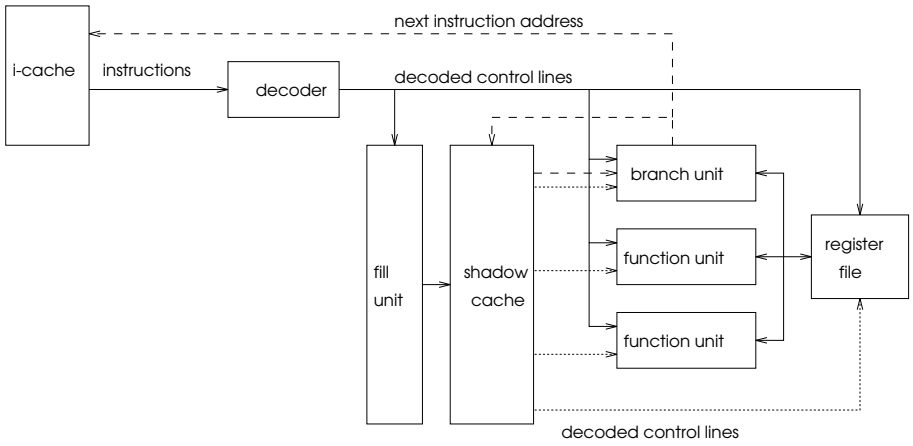

Figure 2: Block diagram of a processor with an instruction fill unit.

and placed the fill unit concept in perspective. Section 2 details the operation of the fill unit for various configurations of functional units, and Section 3 describes how the lines of a shadow instruction cache would be formatted for a particular instruction set (MIPS), especially in regard to handling the delayed branch. Section 4 presents an experimental evaluation of the design, and a summary and conclusions appear in section 5.

2 An Instruction Fill Unit

As described in the introduction, we use a fill unit to collect machine instructions that can always be issued together in the same cycle. In that sense, the long words formed by the fill unit resemble VLIW instructions.

2.1 General Operation

An instruction fill unit is a hardware assist that packs groups of fully or partially decoded sequential instructions together into a long word, and stores this instruction in a shadow instruction cache. These groups are the basis of multiple issue in the proposed implementation, as opposed to multiple decoding with hardware dependency checking found in superscalar processors, and these groups must be guaranteed to exclude dependencies that would result in incorrect results. Figure 1 conceptually shows how filled lines obtained from the shadow instruction cache can improve performance in a processor with a 5-stage pipeline. In the right half of the diagram, three instructions are supplied from the shadow cache and multiply-issued in cycle 1; this set of instructions preempts the decoding and issue of a single instruction along the normal scalar path. Note that by storing decoded fields in the shadow cache line, there is no need to (fully) repeat the decoding of instructions fetched from the shadow cache; however, register fetch must still occur prior to execution.

Figure 2 gives a block diagram showing the relationship between the instruction fill unit and the normal instruction cache and the functional units. The fill unit accepts decoded scalar instructions one-by-one as they are provided in program order from the normal instruction cache and decoder. It packs a group of decoded instructions into a buffer based on intra-group data dependencies and resource requirements, until the line is finalized (i.e., completed), at which point the buffer is written into the shadow cache and a new line begins to fill. A shadow cache line entry is identified by the address of the first instruction that was placed into that line. As depicted in Figure 2, the next-instruction address is generated by the branch unit (every cycle) and is sent to the normal instruction cache and the shadow cache. If the address matches that of a shadow cache entry, the shadow cache is given priority over the normal instruction cache and the (partially-) decoded control lines are supplied directly from the shadow cache.

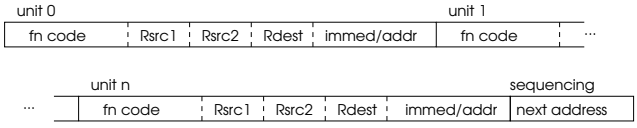

Figure 3: Generic shadow cache line format.

Figure 3 shows the general format of a shadow cache line in a processor with n functional units. Corresponding to each functional unit, there is a fixed set

of fields in the wide format. Because a variable number of decoded instructions can be supplied by a particular shadow cache line, each line must also include a next-address field and supply it to the branch unit. (Note that a program-counter-increment value is less desirable since the required address addition would delay the start of the next instruction fetch.) Filled lines are stored in the shadow cache only if they have multiple instructions filled, as it is more storage efficient to fetch single instructions from the normal instruction cache.

An exception that occurs within a group of multiply-issued instructions will cause the machine to flush the results of the group, enter a scalar-only execution mode, and serially reexecute the scalar instructions in the group up to the point of the exception [6].

2.2 Baseline Fill Unit Design

The design of a fill unit and the filling criteria are intimately tied to the functional unit configuration of the processor. Ideally, each shadow cache line would have as many instructions as allowed by the number of functional units in the processor. However, data dependencies, control dependencies, and resource collisions cause the average number of instructions placed into a shadow cache line to be less than this ideal. For explanation purposes, let us first consider a processor with the 6 functional units shown in Figure 4. The functional units include two load/store units (with store buffers to allow load bypass to the data cache), two integer units, a branch comparison unit, and a floating point unit.

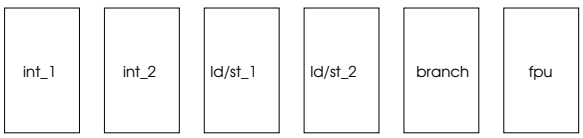

Figure 4: Basic execution units.

In the baseline fill unit case, the finalization (i.e., completion) of a shadow cache line occurs in the fill unit when a branch or RAW/WAW dependency is encountered, or when a resource request is made that cannot be satisfied. In filling a shadow cache line, two instructions with WAR dependencies are allowed to issue in the same cycle (as in a VLIW instruction) since all source registers are read in the second stage of the pipeline. Additionally, an RAW dependency from a value-producing instruction to a store instruction is allowed (assuming the presence of a store buffer, where the store can wait until its store value is ready), and both instructions can be issued in the same cycle. This arrangement of simultaneous issue of instructions with WAR dependencies and RAW dependency for store value is found in the superscalar Motorola 88110 [3].

The dependency checking performed by the fill unit does not require the same amount of logic as a medium- or wide-issue superscalar since the filling is done one instruction per cycle, and no reordering is done. Thus for six units, the fill unit needs only check the single incoming instruction against a maximum of five previous instructions. This could result in faster clock speeds or perhaps fewer pipeline stages than that of a superscalar processor, albeit with slightly reduced average issue rate.

Since the fill unit examines each scalar instruction as it attempts to fill a shadow cache line, the use of register file read ports can be carefully managed. That is, instead of a priority arbitration scheme to assign register read ports to ready instructions in program order as necessary in a superscalar (c.f. section 4.4 of Johnson [7]), ports can be assigned by the fill unit, and running out of ports can be a finalization condition. However, arbitration logic is still required among the functional units for access to write-back buses.

In this work, we omit register renaming since it would likely require an additional pipeline stage. We view this as a tradeoff between more complicated logic in the processor and the extra space needed for the shadow cache.

The next-instruction-address fields chain together a set of multiple-issue groups of instructions. If there is no branch or control-changing instruction in a wide instruction, this field is set to the address of the first instruction that was not filled into the line. A branch is encoded in the shadow cache line by two separate fields: a condition and a branch address. When the last instruction to be filled is a conditional branch, the next-instruction address field serves as the branch-untaken next-instruction address and the branch-address field serves as the branch-taken next-instruction address. When the last instruction to be filled is an unconditional branch, the branch-address field is unused and the branch target address is placed in the next-instruction address field.

Branching to an instruction within a previously filled line presents no logical problems. Since the shadow cache has no line with that starting address, the fetch will be made from the normal instruction cache, and a new fill line will start at the branch target. Because the same instructions at the branch target will appear twice in the shadow cache, the shadow cache tends to become somewhat less storage efficient. Similarly, if both the original filled line and the newly

filled line both are finalized by the same branch, a branch prediction scheme (e.g., a branch history table) that uses the scalar addresses of branch instructions and the starting addresses of branch-finalized shadow cache lines will now have multiple entries. We believe these effects to be minor since all affected lines must end at least by the next branch.

Note that self-modifying code is not well supported. A software coherency scheme for the normal instruction cache can invalidate the addresses of newly-written instructions individually, but these invalidated words might be packed into parts of several filled lines in the shadow cache. A method to determine an invalidate hit in the shadow cache requires that the address range of each line be checked. Since this is expensive, the approach chosen is to instead invalidate the entire shadow cache prior to execution of any self-modified code.

Because the two load/store units are sequentially assigned instructions in the shadow cache line, address collisions can be handled by forwarding logic for RAW and priority logic for WAW. A WAR collision is more problematic but can be handled by a bus lock for a read-modify-write. A page fault by one of the references results in OS invocation, but the shadow cache line itself can be refetched and retried after the fault is handled.

2.3 Extended Fill Unit Designs

Initial experiments with the baseline fill unit design indicated that shadow cache lines were finalized too quickly on general-purpose codes due to RAW dependencies among integer instructions. An important design technique in increasing the performance of general-purpose codes is the use of compound functional units such as the cascaded half-cycle integer ALUs (as done in the triple-issue TI SuperSPARC [2]) and fused 3-operand functional units [9]. (Special handling of dependent integer instructions is also an important part of the MIPS R4000 superpipeline design, where an ALU result can be produced every internal cycle [8].) A compound integer functional unit does not appear to impose cycle time limitations and allows dependent integer instructions to be issued from a shadow cache line in the same cycle.

Figure 5 shows the six basic execution units with a half-cycle integer ALU cascaded above the second integer ALU, the branch comparison unit, and the load/store units. (The cascaded units are conceptually identical to fused multiple-operand units.) For the second integer instruction, the source register read port specifiers must be extended by one bit each to provide a flag that indicates whether they should be

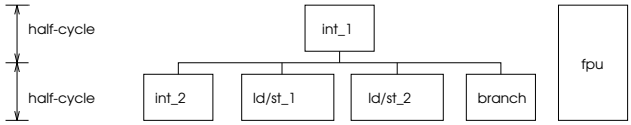

Figure 5: Bypass paths for cascaded execution units.

obtained from the register file or forwarded from the first integer execution unit. The branch condition test and the load/store address calculation can also have RAW dependencies from the first integer unit.

With regards to register file read ports, the forwarding requirements of dependent instructions will be recognized by the fill unit and read ports need not be assigned in these instances. Thus, it becomes less likely that a shadow cache line will be finalized due to lack of register file read ports.

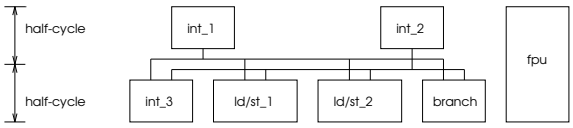

Figure 6: Bypass paths with three integer units.

Figure 6 shows a third integer ALU, which allows two integer results to be calculated in the first half-cycle and then used by the other units. Since forwarding can occur from either *int_1* or *int_2*, this adds a second bit to each forwarding flag.

Branch prediction using a branch address cache can be applied to any of these functional unit configurations to provide speculative execution. The predicted address is determined in parallel with instruction fetch and is applied to the shadow cache as well as the normal instruction cache. Loops can be well exploited by the fill unit design. However, recovery must be undertaken whenever a misprediction is made and will require one or more cycles to flush speculative results and possibly several cycles to execute in scalar mode up to and past the mispredicted branch.

3 A Case Study for the MIPS Architecture

This section determines a specific layout of a shadow cache line and required filling procedures by examining the MIPS architecture [8]. This demonstrates that binary compatibility can be achieved without requiring a full superscalar implementation.

3.1 MIPS Architecture

The MIPS architecture is representative of the class of streamlined architectures that have emerged recently; other architectures in this class have very similar traits [5]. The MIPS ISA only permits memory references in load/store instructions and defines 32 integer registers and 16 floating point registers. An important aspect of the MIPS architecture is the one cycle delay slot for all control changing instructions such as branches, jumps, and calls; i.e., the instruction following the jump or branch is always executed. Similarly, load instructions have a delay, or latency, of one cycle before the data being loaded is available to another instruction. The compiler fills in the delay slots of the loads as well as the delayed branches, and a nop instruction is used whenever a useful instruction cannot be moved into a given delay slot.

3.2 Finalization on a Delayed Branch

In designing a MIPS-compatible fill unit, we assume the six functional units as discussed in the Section 2.2; so, there are six major fields and one next instruction address field in each shadow cache line. The line can be fully or partially decoded. We choose the latter in this example to achieve storage efficiency. The function code fields are thus sized by the number of operations that can be requested of the particular unit; that is, the original MIPS instruction is decoded and then the function request is reencoded for each function unit. Register fields are set according to register file read port assignments; in this case 8 read ports require 3 bits of specification. Additionally the second integer unit, the load/store units, and the branch unit may have values forwarded from the first integer unit. Thus the port fields for these units have an extra bit to indicate forwarding.

The total shadow cache line length for the given assumptions is 259 bits, and thus requires approximately 33 bytes per line. For a third integer unit, an additional 38 bits is added to the shadow cache line to control the third integer unit along with an additional 8 bits for extra forwarding flags so that the total length is 305 bits, or approximately 39 bytes.

The architected delay slot for branches in the MIPS ISA presents a problem. A branch is a finalization condition in the designs in Section 2, and the delay slot instruction must either be filled in the same line (essentially as an extra part of the branch) or a new line should start and contain only the branch and the delay slot instruction. Figure 7 shows a code template for a delayed branch and the two manners of line filling just identified.

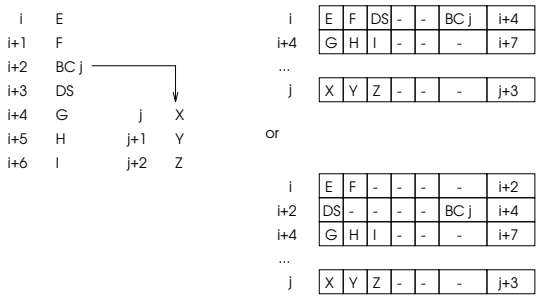

Figure 7: Delayed branch in MIPS.

In Figure 7, the instruction *BC j* is a conditional branch to address *j*, *DS* is the delay slot instruction, and the other capital letters represent non-branch instructions. The indices indicate instruction addresses. At the top right of the figure, the shadow cache lines are shown where the branch and delay slot instruction can be included in the filled line that started with instruction *E* at address *i*. The branch and delay slot instruction finalize that line, and other lines will be filled with instructions from the untaken path (i.e, the line with starting address *i+4*) and with instructions from the taken path (i.e, the line with starting address *j*). Note that the address *i+4* is given in the 30-bit next-address field of the line starting at *i* and that the address j is given in the 30-bit branch-target-address field. Dashes represent nops.

The lower right of Figure 7 shows the situation where the delay slot instruction could not be filled into the line starting at address *i*; thus a new line must be filled starting at address *i+2* and can contain only the branch and the delay slot instruction. Note that since the multiple issue of an instruction pair with a WAR dependency is allowed, there is no dependency condition that can prevent a branch and its delay slot instruction from being filled in the same line; it is instead a matter of whether there is a functional unit currently available for the delay slot instruction.

3.3 Filling Past a Delayed Branch

For MIPS programs, scheduling instructions into branch delay slots is a generic optimization. When this is combined with short basic blocks, finalization of the shadow cache lines can occur too quickly to take much advantage of wide issue. One alternative is to retain one branch per shadow cache line but to widen the line to include the sets of instructions along both the untaken path and the taken path, similar to a VLIW tree instruction [1] [4]. Only one set of instructions will be sent to the functional units, and

the selection will be based on a prediction returned from a branch prediction cache or a default heuristic. The underlying rule is that the instructions prior to the branch, if any, and the delay slot instruction must appear on both paths.

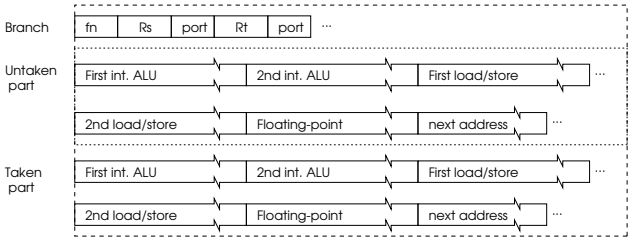

Figure 8: MIPS tree-like line format.

Figure 8 shows this type of wide line. In order to better depict the functionality of the tree-like line, the branch instruction fields apart from the target address field are moved to the top, and the five functional unit instruction fields appear grouped with an untaken address (i.e., normal next address as if the branch were untaken), and these five instruction fields appear again but grouped with a taken address (i.e., the next address as if the branch were taken). The tree-like line length for a two-integer-unit design is 436 bits (55 bytes), and for a three-integer-unit design is 526 bits (66 bytes).

In the normal filling of a wide line, both parts of the wide line are filled with the same instructions up to a branch. If possible, then the branch is filled into the single branch instruction field and the delay slot instruction is filled into both paths. The path not followed has its next address field set (i.e., the branch target address in case of an untaken branch or the address beyond the delay slot instruction for a taken branch), while the fill unit attempts to continue to fill instructions along the path followed by the current execution. The branch prediction cache is updated and will later return a taken/untaken predictor to control which set of instructions will be issued.

If a misprediction has been made, the wrong set of instructions will have been issued, so the functional units are flushed, the branch prediction cache is updated, and the wide line is reissued but with the alternate path. Since the alternate path may not have been traversed before, the set of instructions for this path may only include the instructions prior to the branch, if any, the branch, and the delay slot instruction. Thus, the fill unit can only issue up to the delay slot, but if the next instruction fetch occurs from the normal instruction cache, the fill unit backs up and attempts to continue filling the tree-like line with instructions from this new path. (If the shadow cache returns a wide line from this alternate path, then the filled record of a previous visit is available and multiple issue can occur without further filling.) Therefore, in the best case, there is a one cycle misprediction penalty and the machine continues in wide issue mode, rather than reverting to scalar mode for multiple cycles after the misprediction.

Figure 9 shows filled lines for the same code template as in Figure 7. The shadow cache line at the top right contains instructions from prior to the branch and along both paths (i.e., both paths have been visited previously); this corresponds to the dotted portion of the code template. Alternatively, if the delay slot instruction could not be filled along with branch, a second line is started and can include instructions from farther along each path (i.e., the lower right shadow cache lines in the diagram, again assuming that both paths had been previously visited).

4 Experimental Results

The previous sections described the fill unit and its design. In this section, we present the results of an empirical study of the fill unit using the MIPS architecture.

4.1 Simulation Tool, Benchmarks, and Performance Metrics

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

For benchmarks, we use the integer programs of the SPEC '92 suite. This is because we want to investigate speedups for general-purpose codes representative of the heterogeneous local area network that we described in the introduction. Data is presented for the following 6 benchmark programs: compress with input file in, eqntott, espresso with input file bca, gcc with input file stmt.i, sc with input files test.start and loada1, and xlisp with input file li-input.lsp, which are integer-intensive programs written in C. The programs were compiled using the MIPS C compiler using the optimization flags as distributed in the SPEC benchmark makefiles. All benchmarks were simulated to completion.

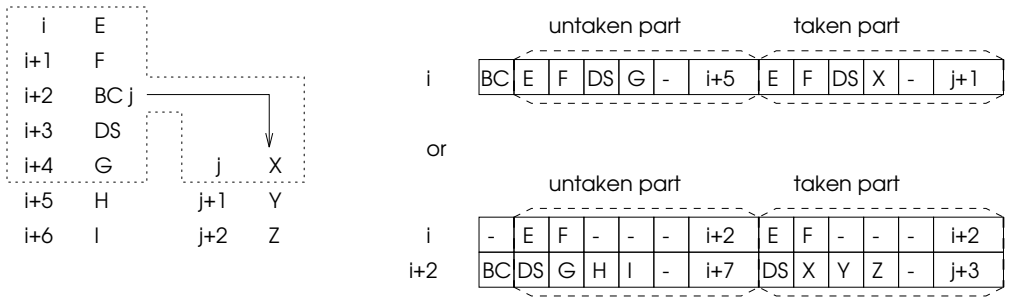

Figure 9: Filling a tree-like line.

For measuring performance, execution time is the sole metric that can accurately measure the performance of an integrated software-hardware computer system. Metrics such as instruction issue rate and instruction completion rate, are not very accurate in general because a compiler may have introduced many redundant operations. However, since the fill unit is a hardware assist, no software modifications are performed and the same instruction stream must be executed regardless of using a scalar design or the fill unit design. Thus the metric of average IPC (instructions per cycle) is appropriate and execution time directly scales with it.

All our studies are carried out with code compiled for a single-issue processor (MIPS R3000). The implications of this decision are: (i) no fill unit-specific optimizations were performed on the code, and (ii) the code is scheduled for a single-issue processor, which can have serious implications on fill unit performance. In that sense, the performance results presented here could be viewed as pessimistic.

4.2 Simulated IPC values

Table 1 shows the results of simulating the SPECint92 benchmark programs using a MIPS-compatible design with actual functional unit latencies, two-cycle loads, target prediction with one-cycle delay for incorrect predictions, 100% instruction and data cache hit rates, and 64k-entry shadow cache. The branch prediction method is the two-level adaptive scheme of Yeh and Patt [14] where the first-level table has 16k entries and a pattern size of six bits; the second-level table uses 3-bit saturating counters.

The first column of numbers represents the IPC of a scalar processor without counting any nops that are executed in the delay slots (counting these nops would allow the scalar processor to achieve its ideal of 1.0 IPC, but the fill unit results would be similarly inflated, e.g., compress would achieve an IPC of almost 3). The subsequent columns represent the different fill-unit designs discussed in Sections 2 and 3.

The second column of numbers in the table, labeled "Baseline", is the baseline fill unit, described in Section 2.2, that has a maximum of six instructions per cycle. Even though WAR and RAW-store dependencies are allowed, other data and control dependencies limit the filling effectiveness for the baseline design. Moreover, the performance of this and all other fill unit designs is governed by the loop structure of the benchmark. That is, code that is executed once and only once will have an IPC corresponding to scalar execution only. Code that is executed in a loop will be accelerated on the second and subsequent executions by the multiple issue capability provided by the shadow cache. From the benchmarks, eqntott apparently has the best loop effect and receives a 27% speedup.

The next three columns of numbers represent cumulative extensions to the baseline design but all use the simple line format; these designs correspond to the discussions in Section 2.3. In the third column of numbers, labeled "Dep. Issue", the simultaneous issue of dependent pairs of integer instructions is allowed and includes the calculation and use of store addresses. This significantly improves the performance of espresso and gcc, but actually decreases the IPC of eqntott. This decrease appears to be the result of a slightly reduced target prediction accuracy.

The fourth column of numbers, labeled "+Dep. Load Addr.", represents a design in which the dependent issue includes calculation and use of load addresses. The fifth column of numbers, labeled "+3rd Int. Unit", comes from the simulation of the seven-unit design that has a third integer unit (c.f. Figure 6). The compress and gcc benchmarks reach their maximum speedup with three integer units, at 93% and 75% respectively.

The final column of numbers is a two-integer-unit, dependent-issue design (building on the design reported in the fourth column of numbers) in which the line has been extended to include instructions from both paths of a branch. This corresponds to the design

Program	*Scalar*	*Baseline*	*+Dep. Issue*	*+Dep. Load Addr.*	*+3rd Int. Unit*	*Tree-like Line*
compress	0.89	1.07	1.15	1.59	1.72	1.56
espresso	0.87	0.99	1.15	1.23	1.30	1.32
eqntott	0.85	1.08	1.05	1.06	1.15	1.81
gcc (cc1)	0.87	1.05	1.22	1.27	1.52	1.35
sc	0.83	0.99	1.09	1.11	1.13	1.41
xlisp	0.79	0.97	1.00	1.00	1.01	1.30

Table 1: Simulated IPC for Perfect I and D Caches.

in Section 3.2. The four benchmarks, espresso, eqntott, sc, and xlisp, achieve their maximum speedups for this design, at 52%, 112%, 70%, and 65%, respectively. Eqntott especially exploits the tree-like line design, while compress and gcc appear to need a third integer unit to obtain best performance.

Next, let us look at the results with finite instruction and data caches, and different shadow cache sizes. Table 2 shows the results with 128k-byte instruction and data caches, and for shadow cache size varying from 2k entries to 64k entries. The functional unit configuration and shadow cache line design are same as those of the last column of Table 1; each shadow cache entry is 55 bytes wide. The first column gives the benchmarks, and subsequent columns give the IPC and shadow cache miss rate obtained for the different shadow cache sizes. It can be seen that the shadow cache has a high hit ratio even when it has only 2k entries (110k bytes in total size). When the number of shadow cache entries is increased to 32k, all programs except gcc have negligible shadow cache miss ratios.

5 Conclusions and Future Work

We have taken the idea of a fill unit, originally proposed by Melvin, Shebanow, and Patt, and applied it to the multiple issue of RISC instructions. In comparison to the original proposal, we have investigated the effect of issuing multiple dependent instructions using cascaded ALUs (or fused functional units), proposed a tree-like wide shadow cache line to handle delayed branches with minimal misprediction penalty, and used the normal instruction cache for any cases of single issue to increase the utilization of the shadow cache. We have chosen in the current work to omit renaming.

In comparison to current superscalar processor design, the fill unit has no need for complicated dependency-checking logic in the decoder to examine a large number of instructions each cycle, and there is no need for time- or transistor-consuming register file read port arbitration. The tradeoff for this is a large on-chip auxiliary memory structure, namely the shadow cache, and the logic necessary for the fill unit. Lines in the shadow cache can start at any instruction address and each line will contain multiple instructions. The possibility of overlapped lines exists but this presents no logical problems in instruction fetching or execution; rather, overlapping merely results in negligible dilution of the shadow cache and the branch prediction cache.

The results of our experiments with the MIPS architecture allow us to conclude that a fill unit approach to the multiple issue of RISC instructions is profitable. A design with six functional units has been shown to speed up eqntott by more than a factor of two without any recompilation. Other integer benchmarks saw a speedup between 52% and 93%.

Future work includes further refinements of filling past branches. One possibility is to dynamically eliminate branches and contain short 'if-then' code sequences completely within a shadow cache line. The instructions in the 'then' part would be made conditional, in the same manner that compilers use conditional moves when they are available in the ISA. For more general control flow, multiple speculative condition and prediction fields could be added to the shadow cache line to govern how different sets of instructions in the line should be issued. The value of register renaming as part of the fill unit and the effect of code scheduling should also be investigated.

Acknowledgements

We gratefully acknowledge the help and encouragement of Steve Melvin and Yale Patt during our work on this project. This work was supported in part by the NSF Research Initiation Award CCR-9410706.

References

[1] A. Aiken and A. Nicolau, "A Development Environment for Horizontal Microcode," *IEEE Transactions on Software Engineering*, vol. 14, no. 5, May 1988, pp. 584-594.

Program	*2k*		*4k*		*8k*		*16k*		*32k*		*64k*	
	IPC	miss	IPC	miss	IPC	miss	IPC	miss	IPC	miss	IPC	miss
compress	1.47	0.00%	1.48	0.00%	1.48	0.00%	1.48	0.00%	1.48	0.00%	1.48	0.00%
espresso	1.18	0.61%	1.20	0.34%	1.20	0.14%	1.20	0.00%	1.20	0.00%	1.20	0.00%
eqntott	1.79	0.17%	1.79	0.17%	1.79	0.17%	1.81	0.00%	1.81	0.00%	1.81	0.00%
gcc (cc1)	1.17	14.85%	1.25	7.29%	1.29	3.37%	1.31	1.69%	1.32	0.79%	1.35	0.21%
sc	1.27	2.70%	1.32	0.88%	1.37	0.34%	1.41	0.04%	1.41	0.00%	1.41	0.00%
xlisp	1.27	3.98%	1.29	0.78%	1.29	0.73%	1.30	0.00%	1.30	0.00%	1.30	0.00%

Table 2: Simulated IPC and Shadow Cache Miss Ratios for Different Shadow Cache Sizes

[2] G. Blanck and S. Krueger, "The SuperSPARC Microprocessor," *Proc. 37th COMPCON*, San Francisco, February 1992, pp. 136-141.

[3] K. Diefendorff and M. Allen, "Organization of the Motorola 88110 Superscalar RISC Microprocessor," *IEEE Micro*, vol. 12, no. 2, April 1992, pp. 40-63.

[4] K. Ebcioglu, "Some Design Ideas for a VLIW Architecture for Sequential Natured Software," in M. Cosnard, *et al.*, (eds.), *Parallel Processing (Proc. IFIP WG 10.3 Working Conference on Parallel Processing, Pisa, Italy)*, North Holland, 1988, pp. 3-21.

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

[6] W-M. Hwu and Y. Patt, "Checkpoint Repair for High Performance Out-of-Order Execution Machines," *IEEE Transactions on Computers*, vol. C-36, no. 12, December 1987, pp. 1496-1514.

[7] M. Johnson, *Superscalar Microprocessor Design.* Englewood Cliffs, NJ: Prentice-Hall, 1991.

[8] G. Kane and J. Heinrich, *MIPS RISC Architecture.* Englewood Cliffs, NJ: Prentice-Hall, 1992.

[9] N. Malik, R. Eickemeyer, and S. Vassiliadis, "Interlock Collapsing ALU for Increased Instruction-Level Parallelism," *Proc. Micro-25*, Portland, Decemeber 1992, pp. 149-157.

[10] S. Melvin, M. Shebanow, Y. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines," *Proc. Micro-21*, San Diego, Decemeber 1988, pp. 60-66.

[11] V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman, "The Metaflow Architecture," *IEEE Micro*, vol. 11, no. 3, June 1991, pp. 10-73.

[12] G.S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Transactions on Computers*, vol. 39, no. 3, March 1990, pp. 349-359.

[13] S. Weiss and J.E. Smith, *POWER and PowerPC.* San Francisco: Morgan Kaufmann, 1994.

[14] T-Y. Yeh and Y. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *Proc. ISCA 92*, Australia, May 1992, pp. 124-134.