University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS) | Department of Computer & Information Science

December 2004

RENO: A Rename-Based Instruction Optimizer

Vlad Petric
University of Pennsylvania

Tingting Sha
University of Pennsylvania

Amir Roth
University of Pennsylvania, amir@cis.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation
Petric, Vlad; Sha, Tingting; and Roth, Amir, "RENO: A Rename-Based Instruction Optimizer" (2004). *Technical Reports (CIS)*. Paper 33.
http://repository.upenn.edu/cis_reports/33

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-04-28.

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/33
For more information, please contact repository@pobox.upenn.edu.

RENO: A Rename-Based Instruction Optimizer

Abstract
The effectiveness of static code optimizations--including static optimizations performed "just-in-time"--is limited by some basic constraints: (i) a limited number of logical registers, (ii) a function- or region-bounded optimization scope, and (iii) the requirement that transformations be valid along all possible paths.

RENO is a modified MIPS-R10000 style register renaming mechanism augmented with physical register reference counting that uses map-table "short-circuiting" to implement dynamic versions of several well-known static optimizations: move elimination, common subexpression elimination, register allocation, and constant folding. Because it implements these optimizations dynamically, RENO can overcome some of the limitations faced by static compilers and apply optimizations where static compilers cannot. RENO has many more registers at its disposal--the entire physical register file. Its optimizations naturally cross function or any other compilation region boundary. And RENO performs optimizations along the dynamic path without being impacted by other, non-taken paths. If the dynamic path proves incorrect due to mispeculations, RENO optimizations are naturally rolled back along with the code they optimize.

RENO unifies several previously proposed optimizations: dynamic move elimination [14] (RENO_{ME}), register integration [24] (RENO_{CSE}), and speculative memory bypassing (the dynamic counterpart of register allocation) [14, 21, 22, 24] (RENO_{RA}). To this union, we add a new optimization: RENO_{CF} a dynamic version of constant folding. RENO_{CF} extends the map-table from *logical -- register* --> [*physical -- register*] to *logical -- register* --> [*physical -- register* : *displacement*]. RENO_{CF} uses this extended map-table format to eliminate register-immediate additions--which account for a surprisingly high fraction of the dynamic instructions in SPECint and MediaBench programs--and fuse them to dependent instructions. The most common fusion scenario is the fusion of a register-immediate addition to another addition, e.g., a memory address calculation. RENO_{CF} implements this fusion essentially "for free" using 3-input adders.

The RENO mechanism is works solely with physical register names and immediate values; it does not read or write the physical register file or use any non-immediate values for any purpose. This isolated structure allows us to implement RENO within a two-stage renaming pipeline.

Cycle-level simulation shows that RENO can dynamically eliminate or fold 22% of the dynamic instructions in both SPECint2000 and MediaBench, respectively; RENO_{CF} itself is responsible for 12% and 16%. Because dataflow dependences are collapsed around eliminated instructions, RENO improves performance by averages of 8% and 13%. Alternatively, because eliminated instructions do not consume issue queue entries, physical registers, or issue, bypass, register file, and execution bandwidth, RENO can be used to absorb the performance impact of a significantly scaled-down execution core.

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-04-28.

This technical report is available at ScholarlyCommons: http://repository.upenn.edu/cis_reports/33

RENO: A Rename-Based Instruction Optimizer

Vlad Petric, Tingting Sha, Amir Roth
{*vladp, shatingt, amir*}*@cis.upenn.edu*

December 9, 2004

Abstract

The effectiveness of static code optimizations—including static optimizations performed "just-in-time"—is limited by some basic constraints: (i) a limited number of logical registers, (ii) a function- or region-bounded optimization scope, and (iii) the requirement that transformations be valid along all possible paths.

RENO is a modified MIPS-R10000 style register renaming mechanism augmented with physical register reference counting that uses map-table "short-circuiting" to implement dynamic versions of several well-known static optimizations: move elimination, common subexpression elimination, register allocation, and constant folding. Because it implements these optimizations dynamically, RENO can overcome some of the limitations faced by static compilers and apply optimizations where static compilers cannot. RENO has many more registers at its disposal—the entire physical register file. Its optimizations naturally cross function or any other compilation region boundary. And RENO performs optimizations along the dynamic path without being impacted by other, non-taken paths. If the dynamic path proves incorrect due to mispeculations, RENO optimizations are naturally rolled back along with the code they optimize.

RENO unifies several previously proposed optimizations: dynamic move elimination [14] (RENO_{ME}), register integration [24] (RENO_{CSE}), and speculative memory bypassing (the dynamic counterpart of register allocation) [14, 21, 22, 24] (RENO_{RA}). To this union, we add a new optimization: RENO_{CF} a dynamic version of constant folding. RENO_{CF} extends the map-table from $logical-register \rightarrow [physical-register]$ to $logical-register \rightarrow [physical-register : displacement]$. RENO_{CF} uses this extended map-table format to eliminate register-immediate additions—which account for a suprisingly high fraction of the dynamic instructions in SPECint and MediaBench programs—and fuse them to dependent instructions. The most common fusion scenario is the fusion of a register-immediate addition to another addition, e.g., a memory address calculation. RENO_{CF} implements this fusion essentially "for free" using 3-input adders.

The RENO mechanism is works solely with physical register names and immediate values; it does not read or write the physical register file or use any non-immediate values for any purpose. This isolated structure allows us to implement RENO within a two-stage renaming pipeline.

Cycle-level simulation shows that RENO can dynamically eliminate or fold 22% of the dynamic instructions in both SPECint2000 and MediaBench, respectively; RENO_{CF} itself is responsible for 12% and 16%. Because dataflow dependences are collapsed around eliminated instructions, RENO improves performance by averages of 8% and 13%. Alternatively, because eliminated instructions do not consume issue queue entries, physical registers, or issue, bypass, register file, and execution bandwidth, RENO can be used to absorb the performance impact of a significantly scaled-down execution core.

1 Introduction

Superscalar execution cores execute instructions reading, writing, and forwarding operands, but also performing a significant amount of book-keeping: register renaming, buffer (issue queue, register file) allocation, dynamic wakeup/select scheduling, and buffer de-allocation. High performance implementations demand both high execution bandwidth and low execution latency as well as high-capacity, high-bandwidth book-keeping.

Map-Table	Raw Instruction	Renamed Instruction	Map-Table Action	Executed Operation
r1→p1, r2→p2	add r1, r2, r3	add p1, p2, p3	r3→p3	(p1+p2)→p3
r1→p1, r2→p2, r3→p3	move r3, r2	move p3, p4	r2→p4	(p3+0)→p4
r1→p1, r2→p4, r3→p3	load r4, 8(r2)	load p5, 8(p4)	r4→p5	MEM[p4+8]→p5

Map-Table	Raw Instruction	Renamed Instruction	Map-Table Action	Executed Operation
r1→p1, r2→p2	add r1, r2, r3	add p1, p2, p3	r3→p3	(p1+p2)→p3
r1→p1, r2→p2, r3→p3	move r3, r2	—	**r2→p3**	—
r1→p1, **r2→p3**, r3→p3	load r4, 8(r2)	load p5, 8(**p3**)	r4→p5	MEM[**p3**+8]→p5

Figure 1: **TOP:** Conventional processing. **BOTTOM:** Dynamic move elimination (RENO$_{ME}$)

Previous research showed that the level of indirection provided by register renaming can be exploited to achieve the effect of executing register moves without actually executing them [14]. On most processors, register move are pseudo-instructions that expand to register-immediate additions with an immediate of zero. As the top part of Figure 1 shows, a conventional processor allocates a new physical register to a *move* and executes an "add-to-zero" operation. The bottom part of the figure shows the same instruction sequence on a processor augmented with dynamic move elimination. This processor recognizes the move idiom and achieves the same end result by mapping the destination register of the move to the same physical register as its source, $p3$, move and discarding the instruction itself.

Dynamic move elimination has two primary benefits. First, the execution latency of move itself–admittedly only one cycle–is removed from the dataflow critical path of the program. The conventional processor in our example executes the *add* in cycle 1, *move* in cycle 2, and *load* in cycle 3. The move-elimination enabled processor executes *add* in cycle 1 and *load* in cycle 2. Because *add* and *move* effectively "share" a physical register ($p3$), the *load*'s dependence on the *move* effectively "short-circuits" and becomes a dependence on the *add*. When the *add* completes, it wakes up the dependent *load* directly. Second, and perhaps more importantly, the bandwidth and buffer cost of the *move* is removed from contention with remaining instructions. Eliminated moves do not consume physical registers or execution buffers, and do not contend for scheduling slots, bypasses and register file read and write ports. Other researchers have subsequently co-opted the same map-table short-circuit trick to eliminate dynamically redundant instructions [24] or perform speculative memory bypassing [14, 21, 22, 24].

In this paper we present a different, unified view of these techniques, that of different aspects of a single mechanism which uses "short-circuit" register renaming to implement the dynamic counterparts of well-known static optimizations. We call this mechanism **RENO (RENaming Optimizer)**. Dynamic move elimination is **RENO**$_{ME}$, the counterpart of static move elimination. Register integration [24]—one particular implementation of redundant instruction elimination and speculative memory bypassing—is **RENO**$_{CSE+RA}$, the combination of the dynamic counterparts of dynamic counterparts of static common-subexpression elimination and register allocation. The static versions of these optimizations are inherently limited by: (i) a small register namespace, (ii) separate, function-level compilation, (iii) conservative information about memory dependences, and most critically (iv) the requirement that any transformation be correct along all possible static paths. The dynamic RENO versions: (i) can use the much larger physical register namespace, (ii) do not see function or other static compilation-block boundaries, (iii) can optimize based on dynamically available or speculative memory dependence information, and (iv) need only worry about the correctness of the optimization along the current dynamic path. If the path turns out to be mis-speculated or memory dependence information turns out to be wrong, the wrong instructions are rolled back and RENO optimizations are rolled back with them.

In addition to presenting this unified view, we also add another optimization to RENO: **RENO**$_{CF}$, the dynamic counterpart of constant folding. Previous optimizations work within the confines of the conventional logical-register to physical-register mapping, i.e., $l \rightarrow [p]$. RENO$_{CF}$ extends this mapping to $l \rightarrow [p : d]$; the interpretation of this mapping is the sum of the register value and a displacement d. RENO$_{CF}$ eliminates register-immediate additions in the same way that RENO$_{ME}$ eliminates register moves. In Figure 2, we reconsider our example instruction sequence where the *move* is replaced by an

Map-Table	Raw Instruction	Renamed Instruction	Map-Table Action	Executed Operation
r1→p1, r2→p2	add r1, r2, r3	add p1, p2, p3	r3→p3	(p1+p2)→p3
r1→p1, r2→p2, r3→p3	addi r3,4,r2	addi p3, 4, p4	r2→p4	(p3+4)→p4
r1→p1, r2→p4, r3→p5	load r4, 8(r2)	load p5, 8(p4)	r4→p5	MEM[p4+8]→p5

Map-Table	Raw Instruction	Renamed Instruction	Map-Table Action	Executed Operation
r1→[p1:0], r2→[p2:0]	add r1, r2, r3	add [p1:0], [p2:0], p3	r3→p3	(p1+p2)→p3
r1→[p1:0], r2→[p2:0], r3→[p3:0]	addi r3, 4, r2	—	**r2→[p3:4]**	—
r1→[p1:0], r2→**[p3:4]**, r3→[p3:0]	load r4, 8(r2)	load p5, 8(**[p3:4]**)	r4→p5	MEM[(**p3+4**)+8]→p5

Figure 2: **TOP:** Conventional processing. **BOTTOM:** Dynamic constant folding (RENO$_{CF}$)

register-immediate addition $addi\ r3,4,r2$. At the top of the figure, we see that a conventional processor treats the *addi* like any other instruction, allocating a new physical register to it ($p4$) and executing the operation $(p3+4) \rightarrow p4$. Shown at the bottom of the figure, RENO$_{CF}$ eliminates the *addi* by setting the extended mapping $r3 \rightarrow [p3:4]$, and renames the *load* to $load\ p5,8([p3:4])$. Notice, while the latency of *addi* has been removed from the dataflow graph, the *load* address calculation now has to perform a 3-input addition ($(p3+4)+8$) rather than a 2-input one. RENO$_{CF}$ exploits the fact a 3-input adder can be implemented by prepending two gate levels to a 2-input adder using the carry-save technique. In effect, RENO$_{CF}$ defers register-immediate additions and then "fuses" them to dependent operations, exploiting the fact the most common fusion scenario—addition to addition—can be performed in "zero cycles" using the 3-input adder technique.

The RENO mechanism is based solely on physical register names—it does not read or write register values for any purpose—and can be implemented comfortably within a 2-stage renaming pipeline. RENO$_{CF}$ requires two straightforward changes to the execution engine.

The folding of register-immediate additions may seem like a narrow optimization. However, these instructions—which are used in programmatic increments, address calculation, stack frame management, and loop control—account for an average of 12% and 17% of all dynamic instructions in SPECint2000 and MediaBench [17], respectively. RENO$_{CF}$ also enables much simpler implementation of RENO$_{CSE+RA}$, i.e., register integration. It turns out that the primary benefit of RENO$_{CSE+RA}$ is the elimination of loads. The elimination of ALU operations provides a much smaller direct benefit but is required to serve as the dataflow "glue" that connects load eliminations. We observed however that the primary glue components are in fact register-immediate additions. Because of this synergy and because RENO$_{CSE+RA}$ is a table-driven technique whereas RENO$_{CF}$ is not, we propose and advocate a configuration in which RENO$_{CF}$ eliminates register-immediate additions and RENO$_{CSE+RA}$ focuses on eliminating loads. This configuration collapses an average of 22% of the dynamic instructions in both SPEC and MediaBench—very nearly the 25% a full-blown RENO$_{CSE+RA}$ configuration would collapse—but with much smaller tables and far fewer table lookups.

We implement RENO in a cycle-level Alpha AXP simulator. On a 4-way superscalar, dynamically scheduled processor, RENO yields performance improvements of 8% and 13% on SPECint and MediaBench, respectively. Despite these performance improvements, RENO's best attribute may be its ability to reduce consumption of execution resources like physical registers, issue queue entries, ALUs, and bypass paths. If the primary concern is engineering "complexity" rather than performance, RENO can achieve the same performance as the 4-way baseline, but with 30% fewer physical, 1 fewer ALU and associated issue, bypass, and register file bandwidth, and a 2-cycle scheduler.

This paper:

- Unifies several previously proposed techniques into a single framework called RENO, which uses map-table short-circuits to implement dynamic analogs of traditional static optimizations.

- Proposes RENO$_{CF}$, a renaming-based implementation of dynamic constant folding. RENO$_{CF}$ is the first name-based technique we are aware of that can optimize instructions that produce values that are not already in the physical register

file, i.e., which cannot simply share an existing register. It accomplishes this using an extended map table and a limited form of dynamic operation fusion.

- A simulation-driven performance evaluation of RENO.

The next section describes the formulation of each of the RENO optimizations, focusing on RENO$_{CF}$. Section 3 describes RENO's implementation, including detailed descriptions of the renaming pipeline. Section 4 presents a simulation-driven performance evaluation.

2 RENO Optimizations

RENO is a modified register renamer that collapses instructions out of the dynamic instruction stream. To do this, RENO uses a single simple trick: physical register sharing. RENO looks for instructions whose output values it can "prove" already exist somewhere in the physical register file. When it finds such an instruction, RENO sets the map table entry of its output register to point to the physical register that contains (or will contain) the proper value. The instruction itself is removed from the execution core in terms of latency (the eliminated instruction's would-be consumers are short-circuited to read the existing physical register), bandwidth (the collapsed instruction does not consume a scheduling or writeback slot), and buffer capacity (the collapsed instruction is not allocated an issue queue entry or a physical register).

The dynamic counterparts of several different static optimizations, move elimination, common subexpression elimination and register allocation, can be formulated as RENO optimizations. All use the same basic map table manipulations and register sharing framework. The differences between one optimization and another have to do with the types of instruction collapsed the machinery that detects collapsing opportunities. One of our contributions is the demonstration that constant folding can fit into this framework, and can synergize with its previously described components.

2.1 RENO$_{ME}$: Move Elimination

Dynamic move elimination [14] is the oldest and simplest RENO-style optimization. RENO$_{ME}$ adds only one component to the basic register sharing machinery: a circuit that can identify register moves. Figure 1 showed how move elimination is implemented using register sharing.

2.2 RENO$_{CSE+RA}$: Common Sub-expression Elimination and Register Allocation

Just as the previously-proposed dynamic move elimination is RENO$_{ME}$, the previously-proposed register integration [24] is essentially RENO$_{CSE+RA}$, the dynamic formulation of common sub-expression elimination and register allocation.

Register integration effectively treats the physical register file as a value cache. Just like a compiler maintains a table of "available expressions" which it uses to recognize redundancies, register integration maintains a table that describes which values are currently available in the physical register file. The integration table (IT) contains tuples of the form $<opcode/imm, [p_{in1}], [p_{in2}] \rightarrow [p_{out}]>$; each tuple describes one physical register in terms of the register dataflow of the instruction that created the value. When renaming an instruction, the IT is searched (using a hashing scheme, not associatively) for tuples that match the operation the current instruction will perform, i.e., same opcode/immediate and same input physical registers. If a match is found, the current instruction is considered redundant and is bypassed—no surprise here—by mapping its output to $preg_{out}$.

The top of Figure 3 shows an example of RENO$_{CSE}$, common subexpression elimination as implemented by register integration. The instruction sequence appears less contrived when one imagines other instructions interceding between the ones shown. The first *load* is non-redundant. It is allocated the new register $p3$, executed and also assigned an IT entry that describes the value it is computing into $p3$. The signature or tag of this entry is the operation and input register $p1$ used in

Map-Table	Raw Instruction	Renamed Instruction	Map-Table Action	Executed Operation	IT Entry Created
r1→p1,	load r3,8(r1)	load p3,8(p1)	r3→p3	MEM[p1+8]→p3	<**load/8,-,p1→p3**>
r1→**p1**, r3→p3	**load** r4,**8**(r1)	—	**r4→p3**	—	—
r1→p1, r3→p3, r4→p3	add r3,r3,r1	add p3,p3,p6	r1→p6	(p3+p3)→p6	<add/-,p3,p3→p6>
r1→**p6**, r3→p3, r4→p3	**load** r3,**8**(r1)	load p8,8(p6)	r3→p8	MEM[p6+8]→p8	<load/8,-,p1→p3>

Map-Table	Raw Instruction	Renamed Instruction	Map-Table Action	Executed Operation	IT Entry Created
sp→p8, r1→p1, r2→p2	store r2,8(sp)	store p2,8(p8)	—	p2→MEM[p8+8]	<**load/8,-,p8→p2**>
sp→p8, r1→p1, r2→p2	addi sp,-16,sp	addi p8,-16,p9	sp→p9	(p8-16)→p9	<**addi/16,p9,-→p8**>
sp→p9, r1→p1, r2→p2	add r1,r1,r2	add p1,p1,p3	r2→p3	(p1+p1)→p3	<add/-,p1,p1→p3>
sp→**p9**, r1→p1, r2→p4	**addi** sp,**16**,sp	—	**sp→p8**	—	—
sp→**p8**, r1→p1, r2→p4	**load** r2,**8**(sp)	—	**r2→p2**	—	—

Figure 3: **TOP:** Common subexpression elimination (RENO$_{CSE}$). **BOTTOM:** Speculative memory bypassing (RENO$_{RA}$)
.

the computation. The second load is redundant with the first load. RENO$_{CSE}$ detects this redundancy because $r1$ has not changed since the first load and so the signature of the current *load*—*load* with immediate offset 8 reading physical register $p1$—matches the signature of the IT entry created by the first load. The now familiar collapsing operation is performed by setting the mapping $r4 \rightarrow p3$, the first and second loads now "share" physical register $p3$. Redundant, collapsed instructions do not create IT entries because an identical entry must already exist. The subsequent *add* is non-redundant and follows the same steps as the first load. The interesting thing about this *add* is that it overwrites register $r1$. This makes the third load in the sequence (fourth instruction overall) non-redundant with the first two. The signature of this instruction—load with immediate 8 reading physical register $p6$—rightfully does not match the signature in the entry created by the first load.

Register integration's dataflow-style tuples naturally implement a dynamic version of common-subexpression eliminiation. These same tuples—used in a slightly different way—can also implement another common idiom, speculative memory bypassing [14, 21, 22, 24]—the short-circuiting of producer-store-load-consumer chains to producer-consumer chains—for stack store-load pairs. This optimization is the dynamic counterpart of register allocation, which is why within RENO, we refer to it as RENO$_{RA}$. Stack communication is the product of function-level compilation and the limited number of architectural registers which forces spilling. Speculative memory bypassing exploits the fact that there are more physical registers than logical registers, so that spilling out of the physical register file is less frequent.

Register integration implements speculative memory bypassing by the use of reverse IT entries. The bottom part of Figure 3 contains an example. Notice, the stack store (first instruction) doesn't create an IT entry for a future redundant store $<store/8, p2, p8\rightarrow>$, but rather for the anticipated corresponding load (last instruction) $<load/8, -, p8\rightarrow p2>$. This entry is created by switching the opcode from *store* to *load* and placing the input data register $p2$ in the output position. Stack pointer decrements (second instruction) create similar reverse entries for future stack pointer increments (fourth instruction) to allow speculative memory bypassing to bootstrap itself across function calls.

Register integration [26] is a speculative technique, it requires that eliminated loads be re-executed pre-retirement using the data cache read/write store retirement port. In practice, the performance impact of re-execution—which is "dependence-free"—is low.

2.3 RENO$_{CF}$: Constant Folding

The RENO optimization introduced in this paper is a dynamic implementation of constant folding, RENO$_{CF}$. Constant folding differs from move elimination, common-subexpression elimination, and register allocation in that it collapses instructions that compute new values. Moves perform no actual computation. Nor do loads that are collapsed by speculative memory bypassing — these are effectively "memory moves". Common-subexpression elimination collapses instructions that compute, but the values these instructions produce are not new. Because of this difference, RENO$_{CF}$ requires a change in the way register

Map-Table	Raw Insnstruction	Renamed Instruction	Map-Table Action	Executed Operation
r1→[p1:0]	addi r1, 5, r2	—	**r2→[p1:5]**	—
r1→[p1:0], **r2→[p1:5]**	addi r2, 6, r4	—	**r4→[p1:11]**	—
r1→[p1:0], r2→[p1:5], **r4→[p1:11]**	or r4, r1, r8	or **[p1:11]**, [p3:0], p6	r8→[p6:0]	(**(p1+11)**|p3)→p6

Figure 4: Constant folding in RENO$_{CF}$

renaming works. This change in turn forces RENO$_{CSE+RA}$ to be modified. As we will see, however, this modification does pay for itself. Not only does RENO$_{CF}$ greatly expand the scope of dynamic instruction elimination and register sharing optimizations, it can also offload a lot of work from RENO$_{CSE+RA}$ without requiring table lookups.

A new map table format. We have already seen how RENO$_{CF}$ folds register-immediate additions (Figure 2). A conventional map table maps logical registers to physical registers, e.g., $r1 \rightarrow [p1]$. A RENO$_{CF}$ map table maps logical register to physical-register/displacement pairs, e.g., $r1 \rightarrow [p1{:}4]$. RENO$_{CF}$ folds register-immediate additions by annotationg the addition in this new map table format and then fusing it to any subsequent instruction that attempts to use the result.

In theory, annotation and subsequent fusion could be applied to any instruction. In practice, however, RENO$_{CF}$ is limited to folding register-immediate additions. The reasons for this have to do with the complexity of both the extended map table format and the fused functional units which will ultimately accept the deferred operation. We examine these reasons in detail.

Folding register-register operations is hard. First, it should be easy to see why RENO$_{CF}$ does not fold register-register instructions. The chief problem goes beyond map table format and functional units. If RENO$_{CF}$ were to fold register-register operations, it would create fused operations with three register inputs. This would require additional register file ports, bypass paths, and scheduler tag matching hardware. But that isn't the end of it. If RENO$_{CF}$ could fold dependent register-register instructions, it would create fused instructions with four register inputs, then five, and so on. In the absurd limit, RENO could fold all the instructions in a program and execute the resulting expression on a monstrous compound ALU. By limiting itself to register-immediate instructions, RENO$_{CF}$ ensures that fused operations have at most two input registers and that the scheduler, register file, and bypass network, are not further complicated.

The required fused functional units should be practical. When choosing which register-immediate instructions to allow RENO$_{CF}$ to fold, the next criteria we apply is fusability. Because a folded instruction will subsequently be fused to a dependent instruction, it is important that the fused functional unit be feasible to build. For instance, a load is technically a register-immediate instruction—it has only one register input—but it would not be practical to add a load port to the input path of every other functional unit in the machine. For a similar reason, it would not be practical to fold register-immediate multiplies or divides. In fact, the only fusable instructions perform simple single-cycle operations like additions, shifts, and logical functions. These operations could be fused to other operations relatively cheaply. In certain cases, it may even be possible to perform "zero-cycle" fusion, e.g., an addition and a dependent logical operation could be performed in a single cycle or two dependent additions could be performed in a single cycle using a carry-save three-input adder.

Folded instruction sequences should have a fixed map table format. In addition to being fusable, additions, shifts, and logical operations are good candidates for folding because they are also trackable. An operation is trackable if there is a fixed format extended map table representation that can express collapsed operation chains of arbitrary length. This definition extends to groups of operations as well. A trackable operation group has a fixed map table format that can represent arbitrary collapsed chains of any combination of operations from within the group. A few examples should clarify this definition and its importance.

Consider addition. A single, fixed map table entry format $[p{:}d]$ can represent the result of an arbitrary chain of dependent register-immediate additions: $(p+d_1)$ is represented as $[p{:}d_1]$, $((p+d_1)+d_2)$ is represented as $[p{:}(d_1+d_2)]$, $(((p+d_1)+d_2)+d_3)$ is represented as $[p{:}((d_1+d_2)+d_3)]$, and so on. This is because addition is associative, i.e., $((p+d_1)+d_2) == (p+(d_1+d_2))$. This allows RENO to perform the immediate-immediate portion of the operation itself—in other words to "fold the constants"—and defer only a single register-immediate operation. An example of such folding is shown in Figure 4.

Map-Table	Raw Instruction	Renamed Instruction	Map-Table Action	Executed Operation	IT Entry Created
r1→[p1:0]	addi r1,4,r1	—	**r1→[p1:4]**	—	—
r1→[p1:4]	load r3,8(r1)	load p2,8([p1:4])	r3→[p2:0]	MEM[(p1+4)+8]→p3	<load/8,-,**[p1:4]**→[p2:0]>
r1→[p1:4], r3→[p2:0]	load r4,8(r1)	—	r4→[p2:0]	—	—

Figure 5: Common subexpression elimination (RENO$_{CSE}$) and constant folding (RENO$_{CF}$) together.

Shifts and logical functions are also associative because of identities like $((p << d_1) << d_2) == (p << (d_1 + d_2))$ and $((p|d_1)|d_2) == (p|(d_1|d_2)$. There are also identities that allow any number of shifts and adds to be represented in a single map table entry that performs one shift and one add. Suppose we had a map table entry [p:s:d], which had the interpretation $((p << s) + d)$. Any combination of shifts and adds could be represented in this format by applying the identities $(((p << s) + d) + d_1) == ((p << s) + (d + d_1))$ and $(((p << s) + d) << s_1) == ((p << (s + s_1)) + (d << s_1))$. Next, we could add logical operations into the mix as well. However, there is a good reason for not doing so. More general formats represent more complex functions which would be less fusable. For instance, we may end up with a single cycle functional unit if we fuse an adder to another adder. We are less likely to end up with a single cycle unit if we have to fuse a shifter-adder to an adder. We restrict constant folding to register-immediate additions because additions are much more common—and much more useful as dataflow "glue"—than either shifts or logical operations.

2.4 Reformulating RENO$_{ME}$ and RENO$_{CSE+RA}$ to accomodate RENO$_{CF}$

Because RENO$_{CF}$ changes the map-table representation from $r \rightarrow [p]$ to $r \rightarrow [p{:}d]$, the two existing optimizations must be updated to account for this change. RENO$_{ME}$ is updated trivially. Recall, most machines implement moves as register-immediate additions with an immediate of zero. RENO$_{CF}$ therefore subsumes RENO$_{ME}$; it does not distinguish between zero and non-zero immediates.

For RENO$_{CSE+RA}$, we extend IT tuple format and attach a displacement to each register name. The new format is $<opcode/imm, [p_{in1}{:}d_{in1}], [p_{in2}{:}d_{in2}] \rightarrow [p_{out}{:}d_{out}]>$. Interestingly, this modification and the simultaneous actions of RENO$_{CF}$ do not impact either common-subexpression elimination or speculative memory bypassing. RENO$_{CSE+RA}$ integration recognizes two instructions as redundant if they have the same register dataflow, i.e., they read values created by the same dynamic instructions. If a redundant instruction depends on a RENO$_{CF}$ collapsed instruction, then it must also be that the instruction with which it is redundant also depends on the same collapsed instruction. Figure 5 illustrates.

The more significant interaction between RENO$_{CSE+RA}$ and RENO$_{CF}$ involves the overlap in the types of instructions the two optimizations target. Both RENO$_{CSE+RA}$ and RENO$_{CF}$ target ALU operations. RENO$_{CSE+RA}$ can collapse all kinds of ALU operations—including shifts and logical operations and register-register additions—but requires those operations to be redundant and uses table lookups to identify redundancies. RENO$_{CF}$ eliminates only register-immediate additions, but does not require these additions to be redundant and does not use table lookups.

As we alluded to earlier, it turns out that the primary benefit of RENO$_{CSE+RA}$ is the elimination of loads, which are more costly—in terms of both latency and bandwidth consumption—than ALU operations. RENO$_{CSE+RA}$ collapses ALU operations not for their own sake but rather to expose more load elimination opportunities. The prime example of this is RENO$_{RA}$'s reverse entries for stack-pointer decrements (see the bottom of Figure 3) that allow stack-pointer increments to be collapsed, which in turn allows speculative memory bypassing—a load elimination optimization—to be applied across function calls. Incidentally, stack-pointer increments and decrements happen to be register-immediate additions, and we have found that many other ALU operations that enable load elimination are also register-immediate additions. This is not a coincidence, as register-immediate additions are used in address calculations for many memory access idioms.

Given this phenomenon, we have found that a cheap and effective strategy is to repartition elimination responsibilities among RENO$_{CSE+RA}$ and RENO$_{CF}$. Specifically, we use RENO$_{CF}$ to handle ALU operations while RENO$_{CSE+RA}$ focuses on loads. This dramatically reduces the size (by 50%) and bandwidth requirements (by 56%) of the IT while maintaing peak

or near-peak collapsing rates and occasionally even improving them. The key here is that removing ALU tuples from the IT reduces contention with load tuples and can result in more load elimination.

3 RENO Implementation

In this section, we describe several aspects of the RENO implementation.

3.1 Physical Register Reference Counting

All RENO optimizations, even RENO$_{ME}$ require physical register sharing which in turn relies on a physical register reference counting scheme. The scheme tracks the number of times each physical register is used an output, i.e., mapped to an architectural register or in-flight instruction. We do not track the number of times a given physical register is used as an instruction input; "use" counts have been used to drive optimizations that are orthogonal to sharing like dead code elimination [5] and early register reclamation [19]. Our reference counting semantics naturally extend traditional free list semantics: allocations—and RENO "sharing operations"—become increments; e-allocations become decrements.

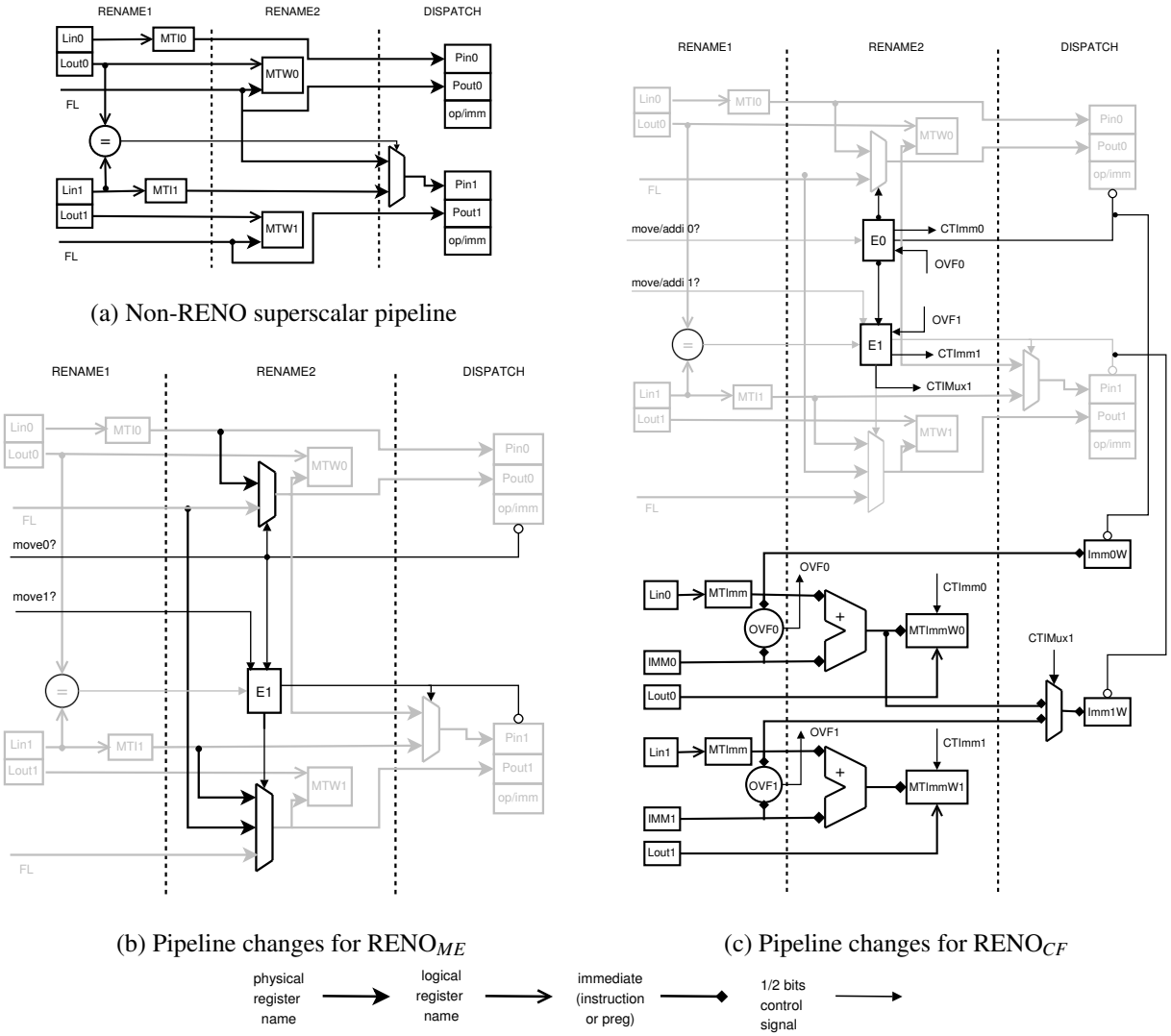
The challenge in implementing reference counting is not in maintaining a counter per physical register it is in updating the counts in a coherent way, interfacing them with the free list, and checkpointing and recovering reference counts. Despite the fact that many previous techniques—including all the ones we build on—rely on reference counting, we have not found a description of that mechanism more detailed than "a table of reference counts". We have developed an efficient implementation which we briefly sketch here.

Our design eliminates the explicit free list; registers are free if their reference counts are zero. Efficient management relies on the observation that free list—and by extension reference count—updates are not in the critical loop of either renaming (register allocation) or retirement (register freeing). There is no special relationship between a given instruction and the new physical register which is allocated to it, so allocation can be pipelined. Similarly, freeing registers—and by extension decrementing reference counts—need not be instantaneous because registers are freed only so that they can be subsequently reallocated. The only time the reference table must provide instant feedback is when a RENO sharing operation causes counter overflow; such a scenario must be flagged quickly so that sharing can be canceled. The table is designed for lazy feedback, we simply avoid this scenario by making reference counters wide enough so that overflow is impossible. The maximum sharing degree is achieved when a single physical register is shared by every architectural register and every active instruction. To avoid overflow, counters must be wide enough to accomodate the sum of these two numbers. This design effectively removes the latency of reference counting/free list management from the register renaming stage.

3.2 The RENO Renaming Pipeline

We reference counting out of the way, we show how RENO is incorporated into an existing MIPS R10000-style register renaming pipeline. It is interesting to note that most of the inline machinery is needed to support register sharing, not any particular optimization. This machinery is required even for the most basic optimization, move elimination. Once already in place, however, other sharing optimizations can be implemented at zero or very low cost. RENO$_{CF}$, the RENO component new to this paper, requires only a parallel displacement accumulation circuit, and no changes to the physical register manipulation circuit relative to RENO$_{ME}$.

With the help of Figure 6, we begin with a discussion of conventional RENO-less renaming and incrementally build up the RENO logic. Our focus here is on RENO$_{ME}$ and RENO$_{CF}$ which builds on top of it. Although we will describe how RENO$_{CSE+RA}$ is incorporated no diagrams for this case are given. Renaming pipeline diagrams for RENO$_{CSE+RA}$—which recall is basically register integration—can be found elsewhere [26]. To simplify the figures, we use a simple 2-way issue



(a) Non-RENO superscalar pipeline

(b) Pipeline changes for RENO$_{ME}$

(c) Pipeline changes for RENO$_{CF}$

Figure 6: Simplified pipeline, for two instructions, each with one input and an output

pipeline in which each instruction has one input register. In the ensuing discussion we call the older instruction in the group (the top one) I_0 and the younger (bottom one) I_1.

Basic, RENO-less renaming pipeline. Figure 6a shows a basic renaming pipeline, which consists of three stages. In the first stage (RENAME1): (i) map-table lookups (MTI) rename the inputs of the current instructions, and (ii) logical register names are cross-checked to determine intra-group dependences; here cross-check consists of comparing I_0's output to I_1's input. In the second stage (RENAME2), the output mappings of the current instructions are written to the map-table (MTW). In the final stage (DISPATCH), dependence information is used to "fixup" input mappings to reflect intra-group dependences by potentially repointing them to the output registers allocated to older instructions from within the group. The results of "fixup"—which here requires a 2-input mux for I_1 to select between I_1's renamed input and I_0's allocated output—are only evident in the issue queue entries, and are not needed by any subsequent instructions. This third stage is therefore not technically considered to be part of the renaming pipeline. As more instructions are renamed in parallel, the number of cross-check comparisons grows linearly per instruction, i.e., I_2's would have to make two comparisons, as does the number of "fixup" mux inputs.

Adding RENO$_{ME}$. Figure 6b shows the changes necessary to implement the simplest application of register sharing:

move elimination; the basic renaming circuit is grayed to highlight the changes.

The first change is the addition of two "move" signals from the decoder. For I_0 a high *move* signal has two effects. First, it sets I_0's output physical register to be I_0's input physical register; this is "map-table short-circuting" or in renaming speak, "output selection". Second, it cancels the creation of an issue queue entry for I_0; this is how RENO eliminates the instruction from the execution core.

The younger instruction, I_1 requires similar changes, but these are a little more complicated. Let's examine the potential inputs to I_1's "output selection" mux. In general, I_1's output physical register can take one of four values. If I_1 is not a move, it is the new physical register from the free list. If I_1 is a move but does not depend on I_0, it is I_1's input physical register from the map table. These two cases are identical to I_0's. However, there are also two other possiblities. If I_1 is a move and depends on I_0 and I_0 is not a move, then I_1's output register should be set to I_0's output register. Finally, if I_1 is a move and depends on I_0 and I_0 is also a move, then I_1's output register must be set to I_0's input register.

The complexity of the output selection mux grows linearly if we rename more instructions in parallel: I_2 mux (not shown) would have 6 inputs, I_3's 8, and so on. We limit this complexity by ignoring the fourth possibility and disallowing the elimination of chains of dependent moves in a single cycle. If I_0 and I_1 happen to be dependent moves, we collapse I_0, but treat I_1 as a non-move and rename it as usual. This decision is performed by the logic block $E1$ (eliminate 1) which internally computes $(move_1 \& (!move_0 \mid !dependence))$. This simplification means that I_1's output selection mux has 3 inputs, I_2's will have 4, and I_3's only 5 (rather than 8). In practice, we expect this simplification to have no performance impact because we expect that most dependent move pairs that are close enough dynamically to be renamed in a single cycle are also close enough statically to be collapsed by the compiler.

Adding RENO$_{CF}$. Figure 6c shows the changes required over RENO$_{ME}$ to implement RENO$_{CF}$. Again, RENO$_{ME}$ is grayed to highlight the new structures and paths. Notice, RENO$_{CF}$ adds virtually nothing to the core renaming path, i.e., physical register name manipulation. The reason is that from the point of view of output selection, constant folding is identical to move elimination. A move elimination sets the output register of an instruction to equal its input register; constant folding does the same thing. What RENO$_{CF}$ does require is a parallel circuit that calculates and accumulates map-table displacements.

Map table displacements are read in the first renaming stage (RENAME1), in parallel with physical register name counterparts. Simple logic examines the upper two bits of the immediate from instruction and the corresponding map-table displacement. A comparison of these bits conservatively checks for potential overflow of the map-table displacement field; if potential overflow is detected, the folding operation must be canceled. The overflow signals ($ovf0$ and $ovf1$) are sent to the elimination logic blocks on the physical register side of the renaming circuit.

In the second renaming stage, map-table displacements and instruction immediates are sent through narrow (16-bit) adders for accumulation, and written to the corresponding map-table entries. Notice, unlike the physical register side of renaming, there are no "output select" muxes for displacements. An output mapping can only have a non-zero immediate if the instruction is eliminated, i.e., if it itself is a register immediate addition or a register move that depends on an older register-immediate addtion. All instructions that generate new values by definition have map-table displacements of zero. The inverse of the elimination decision can serve as the map-table displacement clear signal. Then, even if an instruction *is* eliminated, the output mapping's displacement can only be the sum of the old displacement and the instruction's own immediate, because we are not collapsing dependent instructions in a single cycle.

Finally, in the dispatch stage, we do need to route an additional immediate/displacement per instruction into the issue queue. Recall, this action is only relevant if the instruction is not eliminated. For I_0 there is no choice, the additional immediate is the displacement attached to the input register in the map table. For I_1, we perform the equivalent of "input fixup". If I_1 does not depend on I_0, we choose I_1's own map-table input immediate. If I_1 does depend on I_0, we choose I_0's map-table output immediate. With this configuration we can fold an register-immediate addition into a dependent instruction in one cycle. However, we still cannot fold two dependent register-immediate additions in a single cycle.

Adding RENO$_{CSE+RA}$. Adding RENO$_{CSE+RA}$ requires another input to each "output select" mux on the physical register

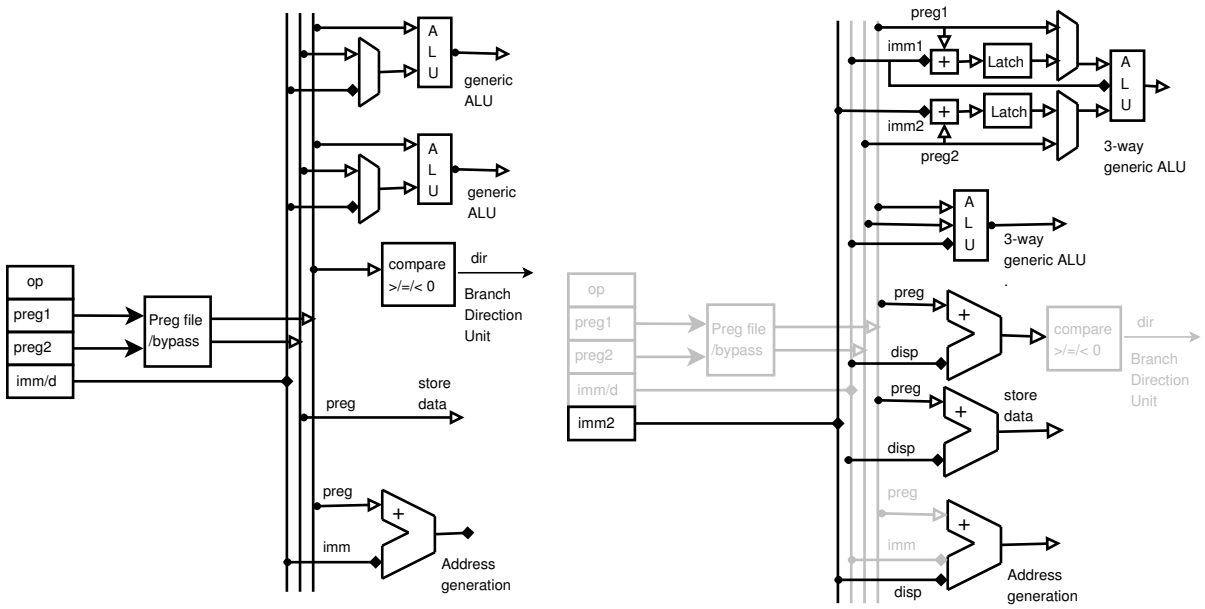


Figure 7: Simplified view of the execution engine

side, corresponding to the physical register output from the IT tuple. The logic to select this input requires performing the "integration test" [24], comparing the physical register input (and immediate if RENO$_{CF}$ is implemented) from the map table with the one(s) from the tuple. The results of these comparisons are fed to the "elimination" logics, $E0$ and $E1$.

Summary. RENO takes a MIPS R10000 renamer and adds "output selection" logic to each instruction slot. This logic includes a 3-input mux (4-input if RENO$_{CSE+RA}$ is also implemented) and control which is derived from the existing dependence check signals and information from the decoder regarding the collapsibility of each of the individual instruction. RENO$_{CF}$ adds a circuit that parallels this renaming circuit in structure but accumulates and selects immediates, rather than physical register names. Actually, this circuit is a little simpler than its register counterpart due to the fact that only eliminated instructions can produce non-zero map-table displacements.

The complexity of a fully general per-instruction RENO renamer grows linearly with each added instruction, so that the complexity of whole circuit grows quadratically. We can limit this complexity by disallowing two dependent instructions from being eliminated in a single cycle. This restriction would for instance, prevent RENO from fully optimizing the instruction sequence in Figure 4 in a single cycle. Fortunately, such contrived sequences are rare as most compilers are smart enough to fold the two *addi*'s statically.

3.3 Execution Core Modifications Required to Support RENO$_{CF}$

RENO$_{ME}$ and RENO$_{CSE+RA}$ change register renaming itself, but do not require changes to the execution engine. RENO$_{CF}$ performs fusion and thus does require execution-core changes. These changes are limited in scope and impact.

Figure 7a shows the integer register paths and functional units of a conventional superscalar execution core. The functional units include two ALUs, the first of which can also perform shifts, a branch direction comparison unit, the store data path and the address generation unit; all adders here are 2-input adders. Issue queue entries consist of input and output physical register numbers, one immediate, and one operation descriptor. The register file has two read ports.

Figure 7b shows the modifications necessary for RENO$_{CF}$; structures that are left unmodified relative to the conventional configuration are grayed. Notice that arguably the most important components—the physical register file and the register

bypass network—are unchanged. This is due to our restriction that even fused instructions have at most two register inputs. The changes RENO$_{CF}$ does require are: (i) an additional immediate field in each issue queue entry, (ii) two additional bits indicating whether either physical register input must be added to the corresponding immediates before fusion with the primary operation (op), (iii) an extra 16-bit immediate path from the issue queue to the functional units, and (iv) enhancements to several of the functional units themselves.

In general, the functional unit changes involve extending all 2-input units to 3-input units that can handle fused add-X operations, and placing two 2-input adders where none existed before. All 2-input integer adders are replaced with 3-input adders, this includes the load/store address generation units. If the caches use sum-address indexing [18] rather than an address generation adder, a 2-gate level 3-input 2-output adder (i.e., the unchained piece of a carry-save adder) is added to the cache address path. 2-input adders are also placed on the branch direction computation and store data, i.e., register to store queue, paths. The rationale for the latter is that register/displacement pairs cannot be written as such into memory, and must be collapsed to singleton values first. 2-input adders are placed at the inputs of the barrel shifter (not shown in the figure), multiply/divide unit (also not shown), and one of the ALUs. We assume that register-immediate addition cannot be "zero-cycle" fused to either a general shift or a multiply/divide. Such attempted fusions—which are recognized by the scheduler—take an additional cycle. The augmented ALU takes care of the rare case in which both inputs of a register-register ALU operation are attached to non-zero displacements. This case also incurs a one cycle penalty.

Our experiments assume that 3-input carry save adders are "free" relative to 2-input adders. We believe this assumption to be valid because a 3-input adder adds only two gate levels—and no carry—over a 2-input adder. However, even if the additional delay cannot be "hidden" and its impact on the clock cycle is deemed too high, most of RENO$_{CF}$'s performance improvement would remain intact. Our results show that RENO$_{CF}$ (not all of RENO) would lose only 20-25% of its relative performance advantage (1-2% absolute) if every fused operation took two cycles. RENO$_{CF}$'s resource and bandwidth consumption benefits would also remain intact as would its synergy with RENO$_{CSE+RA}$. In the worst case, RENO$_{CF}$ can be ommitted and the RENO configuration would include only RENO$_{ME}$ and RENO$_{CSE+RA}$.

3.4 Mis-Speculation Recovery

Processors with conventional register renaming recover from mis-speculation by some combination of map table checkpointing and re-order buffer rollback. The existing machinery is extended naturally to support RENO$_{CF}$: where there exists a physical register number that could potentially be used in recovery, the map table checkpoints and the output (not overwritten) physical register tag in the re-order buffer, that entry must be augmented with an immediate. Re-order buffer immediates are instruction-only immediates and have rollback semantics. Map table checkpoint immediates are accumulated immediates and have checkpoint-restoration semantics. The mechanics of recovery are unchanged.

3.5 Precise State under RENO$_{CF}$

RENO$_{CF}$ is not the equivalent of instruction fusion (e.g., Macro-op scheduling [16]), but it does have a similar effect in that the final piece of an operation is deferred to a future instruction. Despite this fact, RENO$_{CF}$ supports program order precise state. In other words, although interrupts can occur while some folded operations are still outstanding—or in practical terms, while some registers are mapped to non-zero immediates—program state always looks as if each operation occurred in its intended spot in program order. There are two keys to this behavior. The first is that exception handler instructions also execute on the RENO pipeline, so they also naturally interpret input registers with non-zero immediate the intended way. The second key lies in the 2-input adder that sits on the store data path: any attempt to save register state to memory transparently collapses the immediate.

4 Experimental Evaluation

We present a simulation-driven evaluation of RENO. We begin by investigating the instruction coverage and the performance of RENO. We evaluate our design choices regarding the collaboration of RENO$_{CF}$ and RENO$_{CSE+RA}$. Finally, we show that RENO can be used to simplify the execution core by effectively compensating for reductions in physical register file size and issue width, and for the presence of a 2-cycle wakeup-select critical scheduling loop [3].

4.1 Methodology

Our benchmarks are the SPEC2000 integer and MediaBench [17] programs. We compile them for the Alpha EV6 architecture using the Digital OSF compiler with -O3 optimizations and run them to completion. We run the SPEC benchmarks on their training input sets with 2% periodic sampling with 10M instructions per sample, and the MediaBench programs on their supplied inputs with no sampling.

Our timing simulator is based on the SimpleScalar Alpha AXP ISA and system call modules [4]. It models a dynamically scheduled superscalar processor with pointer-based register renaming. The on-chip memory includes a 16KB 1-cycle access instruction cache and a 32KB 2-cycle access data cache; both are 2-way set-associative with 32B blocks. The L2 is 512KB is 4-way set associative with 64B lines and a 10 cycle access latency. Main memory has an access latency of 100 cycles and is accessed via a 16B bus that is clocked at one quarter of the core frequency. A maximum of 16 misses may be outstanding at any time. The fetch engine uses a 16Kb hybrid branch predictor, a 2K-entry, 4-way set-associative BTB, and a 32-entry RAS and can fetch past one taken branch per cycle. The pipeline has 13-stages: 1 branch prediction, 2 instruction cache, 1 decode, 2 rename, 1 dispatch, 1 schedule, 2 register read, 1 execute, 1 complete, and 1 retire. The out-of-order execution core has a 128-entry re-order buffer (ROB), a 48-entry load buffer, a 24-entry store buffer, a 50-entry issue queue, and 160 physical registers. Loads are scheduled aggressively using a 64-entry store sets predictor [7]. Replays due to cache misses and squashes due to memory ordering violations are modeled accurately. Because RENO amplifies effective issue and execution bandwidths, we experiment with two processors configurations. A 4-wide fetch/issue/commit configuration can issue up to 3 integer operations, 1 floating point operation, 1 load, and 1 store per cycle. A 6-wide fetch/issue/commit configuration can issue 4, 2, 2, and 1, respectively.

The Alpha ISA uses 8- and 16-bit immediates, so RENO$_{CF}$ displacements are 16-bits wide. RENO$_{CSE+RA}$ uses a 512-entry 2-way set associative reuse table.

4.2 Instructions Eliminated and Folded

Figure 8 shows the fraction of dynamic instructions eliminated or folded by each of the RENO optimizations. The bottom portion of each bar stack shows moves eliminated by RENO$_{ME}$, the middle shows register-immediate additions folded by RENO$_{CF}$ and the top shows loads eliminated by RENO$_{CSE+RA}$.

With a few exceptions—*mcf* and *mesa*—RENO$_{ME}$ eliminates fewer than 8% of the dynamic instructions in a program; the average of 4%. As we argued previously, register moves are compilation artifacts and a good compiler will generate few of them. With 16-bit displacements, RENO$_{CF}$ folds an additional 12% (SPEC) and 16% (MediaBench) of the dynamic instructions. Because they are used in many common address-generation, loop-control, and stack management idioms, register-immediate additions account for at least 10% of instructions in all programs except *crafty, vpr.place* and *mcf*. They account for 23% of all instructions in *mpeg2.decode*. Adding RENO$_{CSE+RA}$ eliminates an additional 5% (SPEC) and 3.3% (MediaBench) of the dynamic instructions. However, these instructions are loads and so the benefit of eliminating is greater relatively than the benefit of eliminating moves and additions.

Notice, there is a small drop in eliminations—noticeable in *perl.diffmail* for instance—when moving from the 4-wide configuration to the 6-wide one. Remember, we limit RENO so that it cannot eliminate two dependent instructions per cycle.

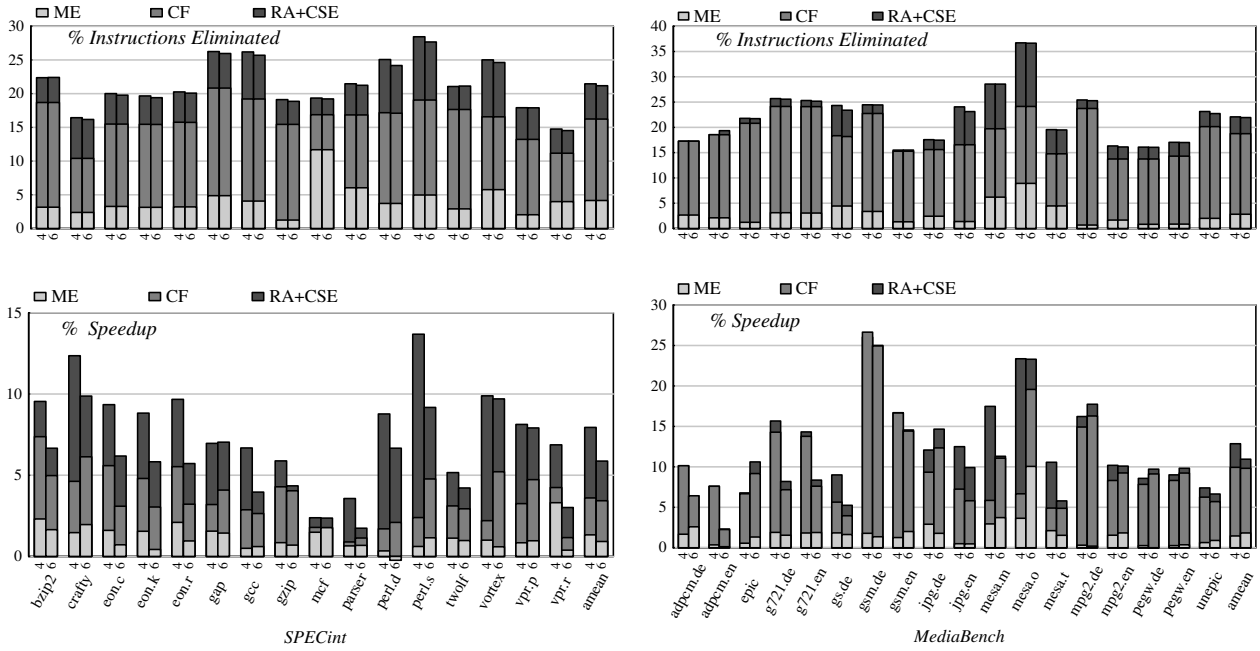


Figure 8: Instruction elimination rates and performance improvements for 4- and 6- wide machines.

Such combinations are rare to begin with; if two instructions are close enough to be renamed in the same cycle, they tend to be close enough for the compiler to optimize them. However, they are somewhat more common in the 6-wide configuration.

4.3 Performance Improvements

On a 4-wide machine RENO improves the performance of SPECint programs by an average of 8% and MediaBench programs by an average of 13% with peak speedups of 14% (*perl.scrabbl*) and 27% (*gsm.decode*) respectively. Average speedups on a 6-wide machine are naturally lower, 6% on SPECint and 11% on MediaBench.

Although RENO_{ME} and RENO_{CF} account for the majority of eliminated instructions, on their own they only provide 4% and 10% speedups on the 4-wide machine. Most of the heavy lifting in SPECint is done by load elimination which is the domain of RENO_{CSE+RA}. Here, RENO_{ME} and RENO_{CF} act as a cheap mechanism that exposes more load elimination opportunities. MediaBench programs are more ALU intensive so RENO_{CF} provides the bulk of the horsepower here.

Critical path analysis. Our performance simulator includes a critical path model based on the one described by Fields [11]; we use a set of dependence edges similar to the one presented in [10]. The main simulator collects timing and dependency data for all retired instructions. An attached analyzer processes this information—i.e., builds dependence graphs and computes maximum depth and edge breakdown—for 1M dynamic instructions at a time. We use this model to gain insight into where RENO makes its performance impact by comparing the breakdowns of RENO and RENO-less configurations.

Figure 9 shows critical path breakdowns for a baseline, RENO-less machine, a machine with RENO_{ME} and RENO_{CF} only, and a machine with full RENO. The latency of each critical edge is added to one of five buckets: *fetch* (fetch bandwidth, instruction cache misses, branch mispredictions, and finite window resources), *ALU* (integer dataflow latency), *load* (D$ and L2 dataflow latency), *memory* (memory dataflow latency), and *commit* (commit bandwidth).

The reason for the peformance discrepancies between the two benchmark sets despite similar elimination rates is apparent. MediaBench is significantly more ALU-critical than SPECint, and because of that it benefits more from RENO_{CF} than from RENO_{CSE+RA}. SPECint, is more load- and memory- critical than integer-critical. RENO_{CSE+RA} attacks the load-execution component, and manages to produce more performance improvement on SPECint than RENO_{CF}. No RENO tech-

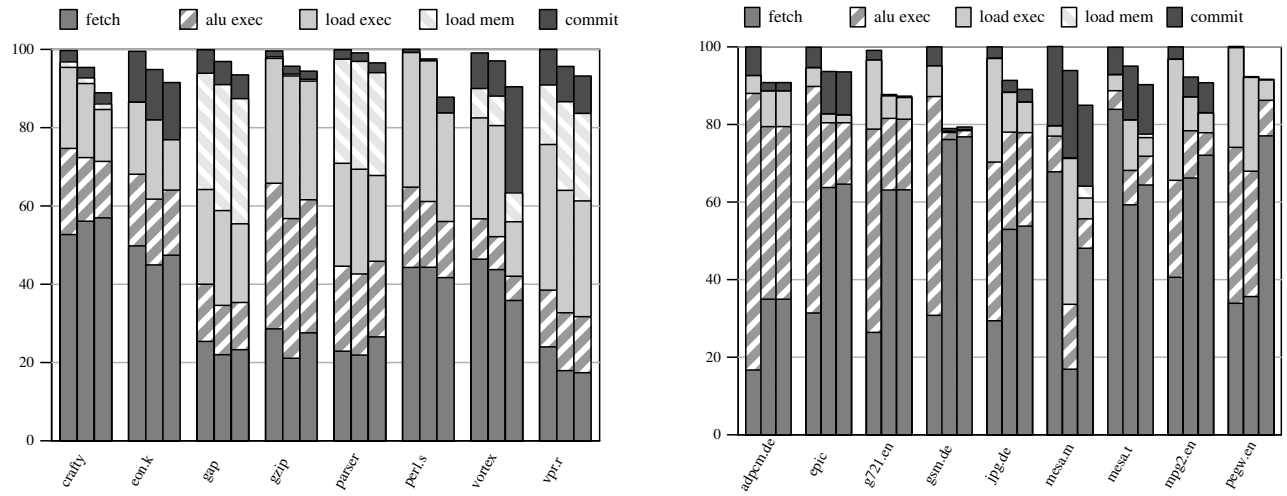


Figure 9: Critical-path breakdown. First bar corresponds to an unoptimized run; second bar - to RENO_{ME} + RENO_{CF}; third bar - to full RENO

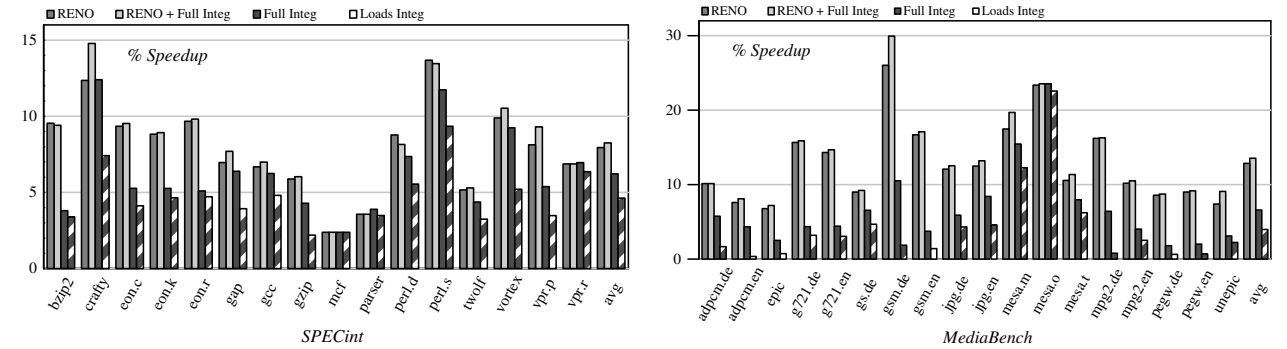


Figure 10: Cooperation between RENO_{CF} and RENO_{CSE+RA}

nique attacks memory latency, so RENO does not fare as well on benchmarks with significant memory components like *gap* and *parser*.

RENO has an interesting effect on fetch-criticality. In *vpr.route* RENO reduces fetch criticality (which includes the finite resources of the machine) by eliminating instructions from the execution engine buffers allowing instructions to dispatch and thus fetch more quickly. In other cases, the fetch component is larger. In almost all MediaBench programs, RENO increases the fetch component. Here RENO removes so many instructions that fetch bandwidth can no longer keep up with highly amplified execution bandwidth and ALU criticality "decays" into fetch criticality.

Vortex is commit store bandwidth bound, and RENO_{CSE+RA} which requires load re-execution to protect from false eliminations exacerbates the problem.

4.4 Dividing Labor Between RENO_{CF} and RENO_{CSE+RA}

In our default RENO configuration, RENO_{CF} collapses register-immediate additions, and RENO_{CSE+RA} handles only loads. This configuration RENO, is the first bar in each group in the graph in Figure 10. We also examine three other configurations.

Implementing RENO_{CF} and RENO_{CSE+RA} and allowing the latter to eliminate all kinds of instructions (second bar) produces average performance improvements of less than 0.5% on both SPECint and MediaBench, with slowdowns (due to increased RENO_{CSE+RA} table conflicts) on several programs. This configuration requires 70% more table accesses than

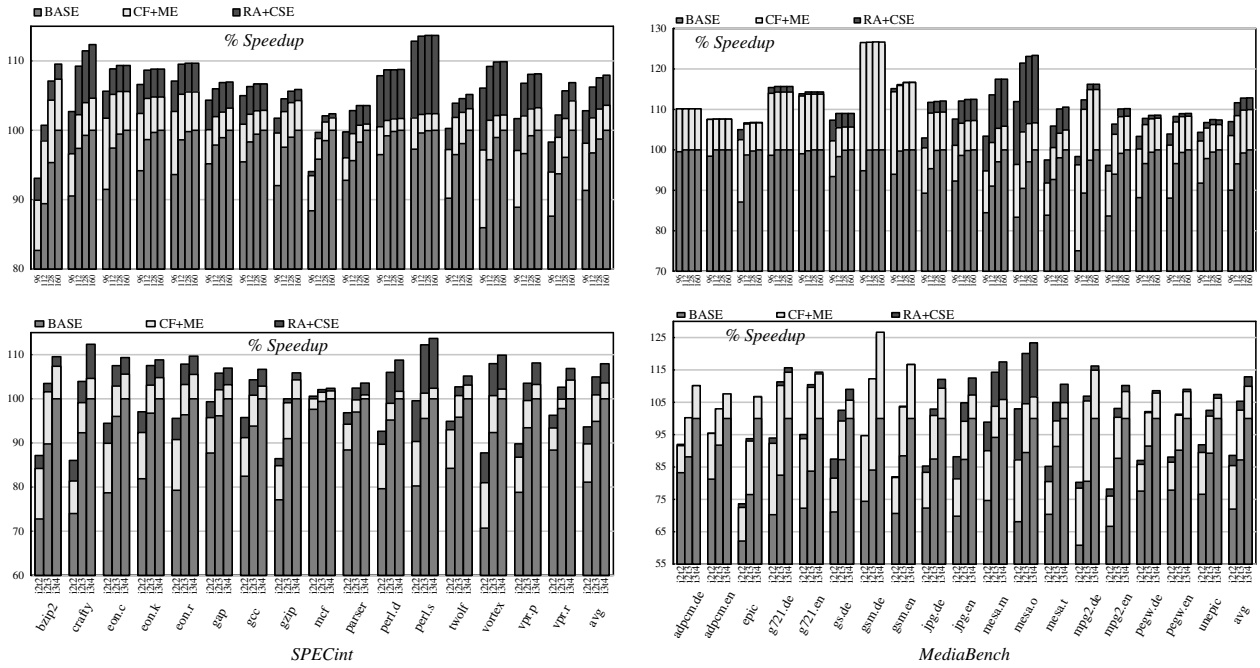


Figure 11: RENO compensating for a reduction in physical register size (top) and execution width (bottom)

RENO.

The starker contrast is between RENO and RENO$_{CSE+RA}$ (i.e., register integration), both of the full-blown variety (third bar) and a variant that eliminates only loads (final bar). Here, RENO wins handily, by an average of 3% over full-blown integration for SPECint and 6% for MediaBench, and by significantly more over load-only integration. RENO's advantage here derives from RENO$_{CF}$'s ability to collapse and eliminate non-redundant instructions and the resulting synergy with load elimination.

4.5 Using RENO to Reduce Execution Core Complexity

RENO improves performance in two primary ways: (i) by collapsing dataflow graph nodes and the edges around them, and (ii) by virtually amplifying the capacity and bandwidth of the execution core. In this section, explore the second effect by using RENO to compensate for reductions in physical register file size and issue bandwidth. We also look at RENO's effects in the presence of a 2-cycle scheduler [3].

Physical Register File Capacity. Scaling the physical register file is one of the main challenges of building a large instruction window, and several physical register late-allocation [12], early-reclamation [19, 20], and caching [6] have been proposed to cope with this problem. RENO reduces the required number of physical registers by not allocating physical registers to collapsed instructions.

In Figure 11, we use RENO to compensate for smaller physical register files. When it comes to performance, RENO$_{CSE+RA}$ contributes more than its share of eliminated instructions because it targets loads. However, when it comes to resources, the much simpler RENO$_{ME}$ and RENO$_{CF}$ are more important because they collapse more dynamic instructions. Acting alone, i.e., with no reuse table at all, they can compensate for a 30% reduction in register file size, from 160 to 112 entries, for SPEC benchmarks. Adding RENO$_{CSE+RA}$ tolerates an additional 10% reduction, to 96 registers. Although 96 physical registers are only 32 more than the minimum required to hold the architected state of a single thread, recall that RENO collapsing works outside the instruction window and persists when an instruction has retired.

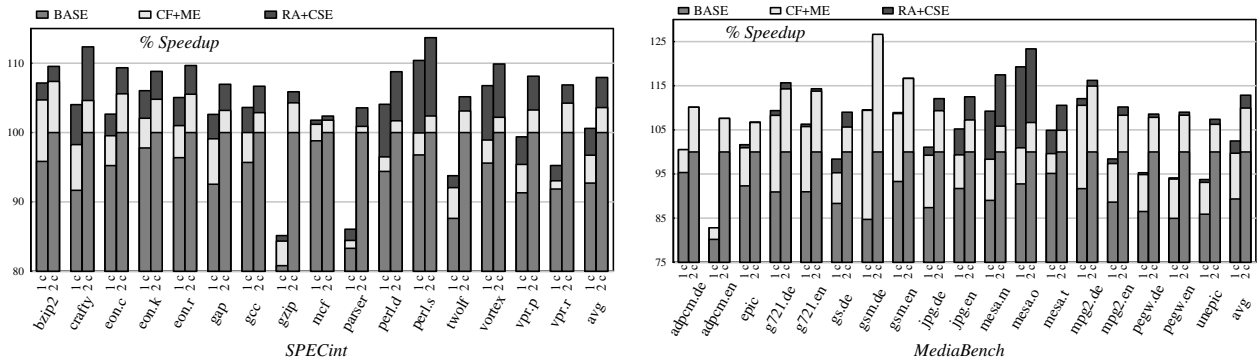


Figure 12: RENO with a 2-cyle wakeup-select loop

Execution Width. Because eliminated instructions do not create register values, RENO can also compensate for reductions in issue bandwidth, including scheduling, execution, register read/write bandwidths and bypass paths. Figure 11 shows RENO's ability to compensate for reduced issue bandwidth.

Our baseline 4-wide configuration could issue 3 integer operations per cycle, plus 1 floating-point, 1 load and 1 store. We look at two additional configurations, both of which can execute only two integer operations per cycle, with one (3W) limited to issuing a total of 3 instructions per cycle and the other (2W) limited to issuing a total of 2.

In the SPEC integer case, RENO$_{ME}$ and RENO$_{CF}$ can compensate for the loss of one issue slot (3W) and an ALU; adding RENO$_{CSE+RA}$ actually results in an overall 5% gain over the baseline 4W configuration. RENO cannot compensate for a 50% reduction in issue bandwidth (to 2W)—neither is it expected to since it does not eliminate 50% of the dynamic instructions—but can restore performance back to within 6% of baseline.

MediaBench programs—which are more execution bound than SPEC integer programs—see a sharper decline in baseline performance when moving from a 4-wide issue configuration to a 3-wise one. However, because they *are* execution-bound, specifically integer-bound, RENO can fully compensate. MediaBench programs execute 2% faster on a 3-way issue processor with RENO$_{ME}$ and RENO$_{CF}$ (no integration) than on a RENO-less 4-way issue processor. For the 2-wide execution, RENO manages to recoup only 18% of the overall performance loss of 29%. Overall performance of 2-way issue is 11% under the 4-way baseline, but some benchmarks (*epic, mpeg2*) exhibit sharper losses.

Viewed a different way, these results indicate that RENO can be used either to improve performance over an aggressive baseline or to maintain baseline performance, but with fewer resources. Again, we are not talking about simply removing an ALU, but also register file ports, bypass paths, and tag-matching hardware.

Scheduling Loop Latency. A scheduler in which wakeup and select are pipelined such that an instruction cannot be woken and selected in a single cycle effectively makes all single-cycle operations appear like two-cycle operations (multi-cycle operations are not affected). This scheduling loop latency [3] can be tolerating by fusing single-cycle operations to their dependents either dynamically [16] or statically [13], removing the scheduling latency of the collapsed dependence edge and making the unit look like a multi-cycle operation.

RENO$_{CF}$ also fuses operations, so RENO can also tolerate some scheduling loop latency as shown in Figure 12. The average performance reductions due to a 2-cycle scheduling loop are 7% for SPEC and 11% for MediaBench. RENO can compensate for this latency in SPEC, and even produce a 2.5% performance improvement in MediaBench. The interesting thing is that RENO tolerates scheduling loop latency in a fundamentally different way than other fusion techniques do. RENO does not create multi-cycle compound-operations—at least not many of them—it simply eliminates many single cycle operations from the dataflow graph.

5 Related Work

Several lines of work are related to RENO.

Physical register sharing. There are two clases of physical register sharing techniques. Techniques in the first class detect sharing opportunities post-execution using value comparisons, and back-patch the map table and optionally issue queue tags [1, 28]. Because they only change mappings after the corresponding instruction has executed, these techniques cannot collapse dataflow dependences or optimize the instruction stream in any way. Rather, their aim is to facilitate the early freeing of registers and the subsequent use of smaller faster register files. In that respect, they are closer in spirit to early resource reclamation techniques like Cherry [19] and dead instruction elimination [5]. The most aggressive of these techniques is physical register inlining (PRI) [20] which allows a physical register to effectively share storage with *its own specifier* if the value is narrow enough. In addition to being limited to register reclamation, these techniques can also be complex since allowing multiple pipeline stages to change the map table can result in both structural hazards and deadlock.

As an aside, there is an interesting juxtaposition between RENO and PRI. PRI maps logical registers to either a physical register *or* a value. RENO maps logical registers to a physical register *and* a value. We are currently investigating ways of exploiting this coincidence to incorporate PRI's register reclamation functionality into RENO.

The second class of physical register sharing techniques detects sharing opportunities at the rename stage and can exploit these to both reshape the dynamic register dataflow graph and to avoid the execution of certain instructions. Unified renaming [14] used reference counts to implement move elimination and memory-dependence prediction to implement store-load bypassing. An earlier implementation of store-load bypassing [21] did not use explicit reference counts and could only operate if both store and load were simultaneously in-flight. Register integration [24] is the first implementation of common-subexpression elimination in this framework and also includes a purely register-based implementation of speculative memory bypassing. RENO unifies these works and extends them with an optimization that applies dataflow graph collapsing to instructions that produce fresh values and which cannot simply share an existing physical register. This is accomplished by a combination of map table extensions and a limited, but cheap form of operation fusion.

Tracking or redundant execution. Register tracking [2] uses the rename stage to fold stack-pointer-immediate additions to create a fast speculative address-generation path for stack loads and allow those loads to issue earlier. Unlike RENO or any form of physical register sharing, register tracking doesn't optimize the main instruction stream inline. Rather, it maintains and optimizes bits of parallel state (here the stack pointer) for specific purposes.

Operation fusion. RENO_{CF} performs a limited form of operation fusion, fusing register-immediate additions to dependent instructions. RENO_{CF} fusion exploits 3-input adders to provide a single cycle latency reduction in the common case. This form of latency reduction by fusion is similar in spirit to fused multiply-add and collapsing ALUs [29].

Recent proposals for online [16] and static [13] pairwise instruction fusion target scheduling loop latency and scheduler capacity and bandwidth rather than dataflow graph execution latency. In addition to its dataflow graph collapsing effects, RENO also amplifies scheduler capacity and bandwidth and helps hide scheduling loop latency which effectively makes all single cycle operations seem like two cycle operations. RENO achieves this last effect not via fusion, but simply by collapsing many single cycle operations out of the program.

Dynamic instruction stream optimization. Optimization the dynamic instruction stream can be done at the decode stage [15]. However, the inability to name physical registers at that stage limits the kinds of optimizations that can be applied. RENO implements optimizations at the rename stage where it can manipulate physical register names and perform a wider range of sharing optimizations. A more powerful microarchitectural optimization model is the offline optimization of cached traces used in rePLay [8, 23]. The offline setting enables the use of optimizations like dead code elimination that require backwards-dataflow analysis (e.g., liveness analysis). rePLay goes a step further by making traces atomic, removing the constraint transformations must be correct on all paths. These features help compensate for rePLay's inablity to manipulate physical registers.

Continuous Optimization [9] (CO, our acronym) is a recent technique that is similar to RENO. Like RENO, CO modifies the renamer to perform move elimination, redundant load elimination, store-load bypassing, and constant folding. And like RENO, CO short-circuits the dataflow graph around eliminated instructions and removes eliminated instructions from execution contention. The difference between RENO and CO is subtle but significant. CO is primarly a value-based mechanism. It performs optimization on full values where possible and otherwise uses a symbolic representation of recent computations to short-circuit data dependences using a conventional map table. In contrast, RENO operates entirely on names, i.e., physical register identifiers. It has no interaction with the physical register file itself and operates on values only if those are immediate values. In RENO, the symbolic representation of recent computations *is* the map table. To use an analogy, RENO is to CO as register integration is to instruction reuse [27].

Because it feeds values from the execution core back to the optimizer, CO has a broader optimization scope than RENO. As the original instruction reuse paper [27] showed, name and equivalence and identities (which RENO exploits) are more restricted than value equivalence and identities which CO exploits. More plainly, two values can be equal even if two names are not. However, because RENO does not rely on values of any kind (addresses included), either to identify optimization opportunities or to actually optimize, it can apply identities even when values are not yet available, and it has a much simpler implementation which fits more naturally into the existing renaming pipeline structure and doesn't require additional paths between renaming and other stages.

6 Conclusion

Compilers optimize code—i.e., allocate registers, collapse dataflow graphs, and eliminate redundancies—as best as they can. However, compilers face have some basic limitations, primarily (i) a limited number of registers, (ii) a static, often function-bounded optimization scope, and (iii) the requirement that code transformations be valid along all possible static paths. Even just-in-time compilers face some of these limitations.

RENO is a modified MIPS-R10000 style register renaming mechanism augmented with physical register reference counting that implements the dynamic counterparts of several well-known compiler optimizations: move elimination, common subexpression elimination, register allocation, and constant folding. Because it implements these optimizations dynamically, RENO can overcome some of the limitations faced by static compilers and apply optimizations those must pass on. RENO has many more registers at its disposal, specifically the entire physical register file. RENO optimizations naturally cross function or any other compilation region boundary. And RENO performs optimizations along the dyanmic path without worrying about the effect of thsoe optimizations on other, non-taken paths. If the dynamic path turns out to be wrong, RENO optimizations are naturally rolled back, along with the code they optimize.

RENO is a framework that unifies several previously proposed optimizations. RENO_{ME} is dynamic move elimination [14]. RENO_{CSE} is register integration [25, 24], the MIPS-R10000 equivalent of instruction reuse [27]. RENO_{RA} is speculative memory bypassing [21, 14, 22, 24]. To these we add RENO_{CF}, a dynamic implementation of constant folding. Register-immediate additions—which surprisingly account for 12% and 17% of the dynamic instructions in SPECint2000 and MediaBench—are eliminated by RENO and dynamically fused to dependent instructions. The fusion of a register-immediate addition to a dependent operation can be done "for free" in the majority of instances, because by far the most common scenario is fusion to another addition which we implement using a 3-input carry-save adder. RENO works solely with physical register names and immediates and doesn't read or write the physical register file itself. We have attempted to show that it can be reasonably implemented within a two-stage renaming pipeline. RENO_{CF} requires only modest extensions to the execution core and does not complicate scheduling or the bypass network.

Cycle level simulation of RENO on the SPEC2000 integer and MediaBench benchmarks shows that RENO can dynamically eliminate or fold 22% of the instructions in the dynamic instruction stream. Dataflow dependences are collapsed around eliminated instructions, improving performance by averages of 8% and 13%. Furthermore, eliminated or folded instructions

do not consume issue queue entries, physical registers, or issue bandwidth and writeback bandwidth allowing this performance to be maintained even with a significantly scaled-down execution core.

References

[1] S. Balakrishnan and G. Sohi. Exploiting Value Locality in Physical Register Files. In *MICRO-36*, Dec. 2003.

[2] M. Bekerman, A. Yoaz, F. Gabbay, S. Jourdan, M. Kalaev, and R. Ronen. Early load address resolution via register tracking. In *ISCA-27*, Jun. 2000.

[3] E. Borch, E. Tune, S. Manne, and J. Emer. Loose Loops Sink Chips. In *HPCA-8*, Jan. 2002.

[4] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, Jun. 1997.

[5] J. Butts and G. Sohi. Dynamic Dead Instruction Detection and Elimination. In *ASPLOS-X*, Oct. 2002.

[6] J. Butts and G. Sohi. Use-Based Register Caching with Decoupled Indexing. In *ISCA-31*, Jun. 2004.

[7] G. Chrysos and J. Emer. Memory Dependence Prediction using Store Sets. In *ISCA-25*, Jun. 1998.

[8] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. Patel, and S. Lumetta. Performance Characterization of a Hardware Framework for Dynamic Optimization. In *MICRO-34*, Dec. 2001.

[9] B. Fahs, T. Rafacz, S. Patel, and S. Lumetta. Continuous Optimization. Technical Report UILU-ENG-04-2207, University of Illinois, Aug. 2004.

[10] B. Fields, R. Bodik, M. Hill, and C. Newburn. Using Interaction Costs for Microarchitectural Bottleneck Analysis. In *MICRO-36*, Dec. 2003.

[11] B. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies via Critical Path Prediction. In *ISCA-27*, Jul. 2001.

[12] A. Gonzalez, J. Gonzalez, and M. Valero. Virtual Physical Registers. In *HPCA-4*, Feb. 1998.

[13] S. Hu and J. Smith. Using Dynamic Binary Translation to Fuse Dependent Instructions. In *CGO-2*, Mar. 2004.

[14] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A Novel Renaming Scheme to Exploit Value Temporal Locality Through Physical Register Reuse and Unification. In *MICRO-31*, Dec. 1998.

[15] I. Kim and M. Lipasti. Implementing Optimizations at Decode Time. In *ISCA-29*, May 2002.

[16] I. Kim and M. Lipasti. Macro-op Scheduling: Relaxing Scheduling Loop Constraints. In *MICRO-36*, Dec. 2003.

[17] C. Lee, M. Potkojnak, and W. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *MICRO-30*, Dec. 1997.

[18] W. Lynch, G. Lauterbach, and J. Chamdani. Low Load Latency Through Sum Addressed Memory (SAM). In *ISCA-25*, Jun. 1998.

[19] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-Order Microprocessors. In *MICRO-35*, Nov. 2002.

[20] B. Mestan, E. Gunadi, and M. Lipasti. Physical Register Inlining. In *ISCA-31*, Jun. 2004.

[21] A. Moshovos and G. Sohi. Streamlining Inter-Operation Communication via Data Dependence Prediction. In *MICRO-30*, Dec. 1997.

[22] S. Onder and R. Gupta. Load and Store Reuse using Register File Contents. In *ICS-15*, Jun. 2001.

[23] S. Patel and S. Lumetta. rePLay: a Hardware Framework for Dynamic Optimization. *IEEE Transactions of Computers*, 50(6), Jun. 2001.

[24] V. Petric, A. Bracy, and A. Roth. Three Extensions to Register Integration. In *MICRO-35*, Nov. 2002.

[25] A. Roth and G. Sohi. Register Integration: A Simple and Efficent Implementation of Squash Re-Use. In *MICRO-33*, Dec. 2000.

[26] A. Roth and G. Sohi. Squash Reuse via a Simplified Implementation of Register Integration. *JILP*, 4, 2002.

[27] A. Sodani and G. Sohi. Dynamic Instruction Reuse. In *ISCA-24*, Jun 1997.

[28] L. Tran, N. Nelson, F. Ngai, S. Dropsho, and M. Huang. Dynamically Reducing Pressure on the Physical Register File through Simple Register Sharing. In *ISPASS-2004*, Mar. 2004.

[29] S. Vassiliadis, J. Phillips, and B. Blaner. Interlock Collapsing ALUs. *IEEE Transactions on Computers*, Jul. 1993.