RENO: A Rename-Based Instruction Optimizer

Vlad Petric, Tingting Sha, Amir Roth
Department of Computer and Information Science, University of Pennsylvania
{*vladp, shatingt, amir*}*@cis.upenn.edu*

Abstract

RENO is a modified MIPS R10000 register renamer 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 apply optimizations in certain situations where static compilers cannot.

Several of RENO's component optimizations have been previously proposed as independent mechanisms. Unified renaming [13] implements dynamic move elimination and speculative memory bypassing [19] (the dynamic counterpart of register allocation). Register integration [21] implements common-subexpression elimination and speculative memory bypassing. RENO unifies these mechanisms and adds a dynamic version of constant folding, RENO$_{CF}$. RENO$_{CF}$ uses an extended map table format and a limited form of dynamic operation fusion.

Cycle-level simulation shows that RENO dynamically eliminates (i.e., optimizes away) 22% of the dynamic instructions in both SPECint2000 and MediaBench. RENO$_{CF}$ is responsible for 12% and 17% of the eliminations, respectively. Because dataflow dependences are collapsed around eliminated instructions, performance improves by 8% and 13%, respectively. 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

RENO (RENaming Optimizer) is a modified MIPS-R10000 register renaming mechanism that uses map table "short-circuiting" to implement the dynamic counterparts of well-known static optimizations: move elimination, common-subexpression elimination, register allocation, and constant folding. 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 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.

RENO unifies several previously proposed mechanisms. **RENO**$_{ME}$ is dynamic move elimination [13]. **RENO**$_{CSE+RA}$ is register integration [21], an implementation of redundant instruction elimination and speculative memory bypassing, the dynamic counterparts of static common-subexpression elimination and register allocation. This paper introduces **RENO**$_{CF}$, the dynamic counterpart of constant folding. Previous RENO optimizations work within the confines of conventional renaming which maps logical registers to physical registers, e.g., $r1 \rightarrow [p1]$. RENO$_{CF}$ extends this mapping and maps logical register to physical-register/displacement pairs, e.g., $r1 \rightarrow [p1{:}4]$. The interpretation of a RENO$_{CF}$ mapping is the sum of the register value and the displacement. RENO$_{CF}$ folds register-immediate additions by annotating the addition in this extended map-table format, and then fusing it to any subsequent instruction that attempts to use the result.

Register-immediate additions are used in programmatic increments, address calculation, and loop control. They account for 12% and 17% of all dynamic instructions in SPECint2000 and MediaBench [16], respectively. Collapsing register-immediate additions in RENO$_{CF}$ also off-loads that responsibility from RENO$_{CSE+RA}$, enabling a much simpler implementation of that mechanism that uses smaller tables and fewer table accesses.

RENO "short-circuiting" eliminates (i.e., optimizes away) instructions and relinks dependences around them. The direct benefit is the removal of eliminated instructions from the dataflow graph. If an instruction on the execution critical path [11] is eliminated, performance improves. The indirect benefit is that eliminated instructions do not consume physical registers or issue queue slots, and do not contend for scheduling, bypass, and register file read/write bandwidths. Cycle-level simulation of RENO shows that on a 4-way superscalar, dynamically scheduled processor, RENO yields performance improvements of 8% and 13% on SPECint and MediaBench, respectively. Alternatively, if the primary concern is execution core "engineering complexity" rather than performance, RENO's bandwidth and capacity amplification effects can be used to maintain a given level of performance but with fewer physical registers and issue queue slots, and reduced issue, bypass, and register file bandwidths.

In this paper, we make the following contributions:

- We present a unifying framework for several previously-proposed techniques called RENO, which uses map-table short-circuiting to implement dynamic counterparts of traditional static optimizations.
- We propose RENO$_{CF}$, a renaming-based implementation of dynamic constant folding. RENO$_{CF}$ uses an extended map-table format and a limited form of dynamic operation fusion.
- We describe in detail RENO's implementation within both a conventional two-stage renaming pipeline and a conventional execution core. We argue that RENO does not require additional renaming stages and that its additional execution core complexity is low.

Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)
1063-6897/05 $20.00 © 2005 **IEEE**

Initial 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

Initial 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

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

- We present 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 exploits the indirection in MIPS R10000-style register renaming to perform optimizations on the dynamic instruction stream. RENO modifies the renamer to collapse (i.e., optimize away) instructions from the dynamic instruction stream and relink dependences around them using a single simple trick: *physical register sharing*. RENO looks for instructions whose output values it can "prove" already exist (or will 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 collapsed instruction is allocated a re-order buffer (ROB) entry so that it may retire, but otherwise is not entered into the execution core.

Several static optimizations have dynamic RENO formulations: move elimination, common-subexpression elimination, and register allocation. All use the same basic map-table manipulations and register-sharing framework. They differ in the types of instructions they eliminate and the machinery they use to detect elimination opportunities. We show that constant folding fits into this framework and synergizes with its previously described components.

2.1 RENO$_{ME}$: Move Elimination

Dynamic move elimination [13] (RENO$_{ME}$) is the simplest RENO optimization. RENO$_{ME}$ requires only the basic register-sharing machinery—which is needed by all RENO optimizations—and a circuit that identifies register moves.

The top half of Figure 1 shows how a conventional processor executes a move. It allocates a new physical register to the *move* and executes an "add-to-zero" operation (in most architectures, register moves are "add-to-zero" pseudo-instructions). The bottom half of the figure shows the same instruction sequence on a processor augmented with RENO$_{ME}$. Rather than allocating a new physical register to the *move*, the processor sets its output register, $r2$, to point to its input register, $p3$. The *move* itself is dropped from the instruction stream and not executed. The example also illustrates the latency reduction effect of this action. The conventional processor executes the *add* in cycle 1, the *move* in cycle 2, and the *load* in cycle 3. The RENO$_{ME}$ enabled processor executes the *add* in cycle 1 and the *load* in cycle 2. Because the *add* and the *move* "share" physical register $p3$, the *load*'s dependence on the *move* "short-circuits" and becomes a dependence on the *add*. When the *add* completes, it wakes up the *load* directly.

2.2 RENO$_{CSE}$: Common-Subexpression Elimination

Register integration [21, 23, 24] is RENO$_{CSE}$, the RENO formulation of common-subexpression elimination. Register integration treats the physical register file as a value cache. Just like a compiler maintains a table of "available expressions" that 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 its value. At renaming, 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. A match implies the current instruction is redundant; it is eliminated and its output is set to $preg_{out}$.

The top of Figure 2 shows an example of RENO$_{CSE}$. 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 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, $p1$. 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 8 and input physical register $p1$) matches the signature of the IT entry created by the first load. The now familiar collapsing operation sets $r4 \rightarrow p3$, making the first and second loads "share" physical register $p3$. The *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* non-redundant with the first two. The signature of this instruction (*load* with immediate 8 and input physical register $p6$) rightfully does not match the signature in the IT entry created by the first *load*.

Because the IT does not track addresses, RENO$_{CSE}$ may eliminate a load in the presence of an older conflicting

Initial 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>

Initial 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**	—	—

Fig. 2. TOP: Common subexpression elimination (RENO$_{CSE}$). **BOTTOM:** Speculative memory bypassing (RENO$_{RA}$)

store. To detect such false eliminations, eliminated loads re-execute prior to commit using the data cache read/write store retirement port [24]. Store Vulnerability Window (SVW) [22] is a general-purpose store-load conflict tracking mechanism that dramatically reduces the number of eliminated loads that must re-execute and mitigates the performance overhead of re-execution.

2.3 RENO$_{RA}$: Register Allocation

Register integration's dataflow-style tuples can be used in a slightly different way to implement speculative memory bypassing [13, 19, 20, 21]—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 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. RENO$_{RA}$ exploits the fact that there are more physical registers than logical registers, so that spilling out of the physical register file is less frequent. Collectively, we refer to RENO$_{CSE}$ and RENO$_{RA}$ as RENO$_{CSE+RA}$.

The bottom part of Figure 2 illustrates. RENO$_{RA}$ uses reverse IT entries. The stack *store* (first instruction) does not 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.

2.4 RENO$_{CF}$: Constant Folding

The RENO optimization new to this paper is RENO$_{CF}$, a dynamic implementation of constant folding. Constant folding differs from move elimination, common-subexpression elimination, and register allocation in that it eliminates 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 eliminates instructions that compute, but the values themselves are not new. Because of this difference, RENO$_{CF}$ requires an extension to register renaming that goes beyond register sharing.

Conventional renaming maps each logical register to a physical-register which acts as proxy for a single value, $l\rightarrow[p]$. RENO$_{CF}$ extends this mapping to $l\rightarrow[p{:}d]$, where p is a physical register and d is a constant displacement. Mappings with non-zero displacements represent *de-normalized values*, whose normal form is the sum of value in p and the displacement d. RENO$_{CF}$ eliminates register-immediate additions by representing the addition operation as a de-normalized value in the map-table. The addition operation itself is deferred until the de-normalized value is needed as an input to another operation. At that time, the value is *normalized* by dynamically fusing the addition to the dependent operation. RENO$_{CF}$ exploits the fact that additions can be fused to many other operations at little cost. The most common fusion scenario is addition-to-addition (e.g., address calculation). The resulting three-input additions can be implemented at minimal cost over two-input additions using the carry-save technique.

Figure 3 shows an example of RENO$_{CF}$. The instruction sequence resembles the one from Figure 1, but we replace the *move* with two dependent register-immediate additions $addi\ r3,4,r3$; $addi\ r3,8,r2$. Again, the example seems less contrived if one imagines interceding instructions. A conventional processor (top half of the figure) treats the *addi*'s like any other instructions, allocates new physical registers $p4$ and $p6$ to them, and executes the dependent operations $(p3{+}4)\rightarrow p4$ and $(p4{+}8)\rightarrow p6$. RENO$_{CF}$ (bottom half) eliminates the first *addi* by setting the extended mapping $r3\rightarrow[p3{:}4]$. It then eliminates the second *addi* by setting the extended mapping $r2\rightarrow[p3{:}12]$. The RENO renamer itself accumulates the displacements $(4{+}8)$, thereby "folding the constants." Finally, RENO$_{CF}$ renames the *load* to $load\ p5,8([p3{:}12])$. Notice, while the latency of both *addi*'s has been removed from the dataflow graph, the *load* address calculation now has to perform a three-input $((p3{+}12){+}8)$ rather than a two-input addition.

Register-immediate additions only, please. De-normalization/annotation and subsequent normalization/fusion could be applied to any instruction. We restrict RENO$_{CF}$ to folding register-immediate additions to limit the complexity of both the extended map-table format and the execution engine which would execute the fused operations.

Initial 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,r3	addi p3, 4, p4	r3→p4	(p3+4)→p4
r1→p1, r2→p2, r3→p4	addi r3,8,r2	addi p4, 8, p6	r2→p6	(p4+8)→p6
r1→p1, r2→p6, r3→p4	load r4, 8(r2)	load p5, 8(p6)	r4→p5	MEM[p6+8]→p5

Initial 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, r3	—	**r3→[p3:4]**	—
r1→[p1:0], r2→[p2:0], r3→**[p3:4]**	addi r3, 8, r2	—	**r2→[p3:12]**	—
r1→[p1:0], r2→**[p3:12]**, r3→[p3:4]	load r4, 8(r2)	load p5, 8(**[p3:12]**)	r4→p5	MEM[(**p3+12**)+8]→p5

Fig. 3. TOP: Conventional processing. **BOTTOM:** Dynamic constant folding (RENO$_{CF}$)

Folding register-register operations potentially creates fused operations with three register inputs. Such operations would require additional register file ports, bypass paths, and scheduler tag matching hardware. Limiting RENO$_{CF}$ to register-immediate instructions ensures that fused operations have at most two register inputs and that scheduler, register file, and bypass complexity is not exacerbated.

A load is a register-immediate instruction but it would not be practical to add a load port to the input paths of every other functional unit. Fusing a multiplier or divider to existing functional units is similarly impractical. Only simple single-cycle operations like additions, small shifts, and logical functions can be reasonably fused. In fact, certain three-input fused functional units may be only trivially more complex than their two-input counterparts. The most common fusion scenario is addition-to-addition (e.g., addition to load/store address calculation). A three-input adder can be made only slightly slower than a two-input adder using the carry-save technique.

Additions, shifts, and logical operations are also good candidates for folding because they have fixed format denormalized representations that can express collapsed operation chains of arbitrary length. Consider addition. A map-table 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 $[p{:}(d_1{+}d_2)]$, and so on. The fact that addition is associative allows RENO to perform the immediate-immediate portion of the operation itself—in other words to "fold the constants"—and to defer only a single register-immediate normalization step to the execution core.

Shifts and logical functions are also associative, as are combinations of adds, shifts, and logical functions. However, three-input functional units that sequentially fuse a shift, logical operation *and* an addition to another operation would be slow. We restrict constant folding to register-immediate additions because additions are much more common—and have more synergy with other instructions optimized by RENO, specifically loads—than either shifts or logical operations.

2.5 Combining RENO$_{CSE+RA}$ and RENO$_{CF}$

RENO$_{CF}$ changes the map-table representation from $r{\rightarrow}[p]$ to $r{\rightarrow}[p{:}d]$. RENO$_{ME}$ is unaffected by this change: a move is effectively a register-immediate addition with a zero immediate. For RENO$_{CSE+RA}$, we extend the integration table tuple format and attach a displacement to each register name. The new format is $<opcode/imm, [p_{in1}{:}d_{in1}], [p_{in2}{:}d_{in2}]{\rightarrow}[p_{out}{:}d_{out}]>$. With this change, RENO$_{CF}$ and RENO$_{CSE+RA}$ mesh seamlessly. RENO$_{CSE+RA}$ 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$_{CF}$ eliminated instruction (i.e., a de-normalized value), then the instruction with which it is redundant must also depend on the same eliminated instruction.

The more significant interaction between RENO$_{CF}$ and RENO$_{CSE+RA}$ concerns the fact that both collapse ALU operations. RENO$_{CSE+RA}$ collapses all kinds of ALU operations (including register-register operations) but requires collapsed operations to be redundant and uses table accesses to identify redundancies. RENO$_{CF}$ collapses only register-immediate additions, but does not require them to be redundant and does not use table accesses.

As it turns out, the main benefit of RENO$_{CSE+RA}$ is the elimination of loads, which are more costly (in both latency and bandwidth) than ALU operations. RENO$_{CSE+RA}$ collapses ALU operations less for their direct benefit than to expose more load elimination opportunities. The prime example is RENO$_{RA}$'s reverse entries for stack-pointer decrements (bottom of Figure 2) that allow the collapse of stack-pointer increments, which in turn allow speculative memory bypassing—a form of load elimination—to be applied across function calls. Incidentally, stack-pointer increments and decrements are register-immediate additions, as are many other operations that enable load elimination. This is not a coincidence as register-immediate additions are used in address calculations for many memory access idioms.

In light of this, we restrict RENO$_{CSE+RA}$ to loads. This division of labor reduces the size (by 50%) and bandwidth requirements (by 56%) of the IT while maintaing near-peak elimination rates. In fact, it can sometimes improve elimination rates by removing ALU operation tuples from the IT, which reduces contention with load tuples.

3 RENO Implementation

We describe several aspects of the RENO implementation.

3.1 Physical Register Reference Counting

All RENO optimizations exploit physical register sharing which relies on a physical register reference counting

scheme. A register reference counting scheme tracks the number of times each physical register is used as 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; such "use" counts have other applications like dead code elimination [6] and early register reclamation [17]. Reference counting naturally extends conventional free list semantics: allocations and RENO "sharing operations" become increments; deallocations become decrements. Registers with reference counts of zero are free. A detailed design of a reference counting mechanism is beyond the scope of this paper.

3.2 The RENO Renaming Pipeline

We now show how RENO is incorporated into an existing MIPS R10000-style renaming pipeline. Interestingly, most of the inline machinery is needed to support register sharing in general, not any particular RENO optimization. This machinery is required even for the most basic optimization, RENO_{ME}. Once in place, other sharing optimizations can be implemented at low cost. RENO_{CF}, the RENO component new to this paper, requires only a displacement accumulation circuit that parallels the physical register name manipulation path.

Our proposed RENO implementation includes a an important simplification. Except for eliminations by RENO_{CSE+RA} (register integration [21]), we disallow RENO from eliminating two dependent instructions in the same cycle. This simplification actually prevents RENO from performing the full optimization shown in Figure 3. However, it has little impact in practice. The instruction sequence in Figure 3 is highly contrived and unlikely to appear in a real program. Dependent *addi* pairs that are close enough dynamically to be renamed in a single cycle are also close enough statically to be folded by a compiler.

With the help of Figure 4, 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}. To simplify the figures, we use a 2-way issue 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. When speaking about the complexity of different aspects of renaming and RENO, we will use the variable N to denote the number of instructions renamed per cycle.

Basic, RENO-less renaming pipeline. Figure 4a shows a basic renaming pipeline, which consists of two stages. In RENAME1, map-table lookups (MTr) rename the inputs of the current instructions and logical register names are cross-checked to determine intra-group dependences. In our example, cross-check consists of comparing I_0's output to I_1's input. In RENAME2, the output mappings of current instructions are written to the map-table (MTw). Finally, dependence information is used to "fixup" input mappings to reflect intra-group dependences by potentially redirecting them to the output registers allocated to older instructions from within the group. The results of input fixup (which here requires a two-input mux for I_1 to select between I_1's input and I_0's output) are only written to the issue queue entries and are not needed by subsequent instructions. Input fixup is therefore considered part of DISPATCH, than than the renaming pipeline proper. The complexity of RENO-less renaming (measured as number of cross-check comparisons and number of inputs to the input fixup mux) is proportional to N for an individual instruction, and to N^2 overall.

Adding RENO_{ME}. Figure 4b shows the changes that implement the simplest application of register sharing: move elimination. The basic renaming circuit is grayed.

The first change is the addition of two *move* signals from the decoder. For I_0, a high $move_0$ 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 "output selection". Second, it cancels the creation of an issue queue entry for I_0; this is how RENO removes the instruction from the execution stream.

I_1 requires similar although slightly more complicated changes. Let us examine the potential inputs to I_1's output selection mux. I_1's output can be mapped to one of four physical registers:

- If I_1 is not a move, it is the new register from the free list.
- If I_1 is a move but does not depend on I_0, it is I_1's input register from the map table.

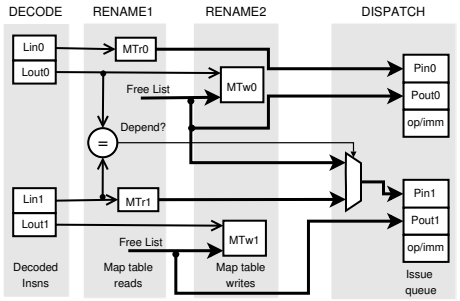
These first two cases are identical to I_0's. However, there are also two other possiblities:

- If I_1 is a move and depends on I_0 and I_0 is not a move, then it is I_0's output register.
- If I_0 and I_1 are dependent moves, it is I_0's input register.

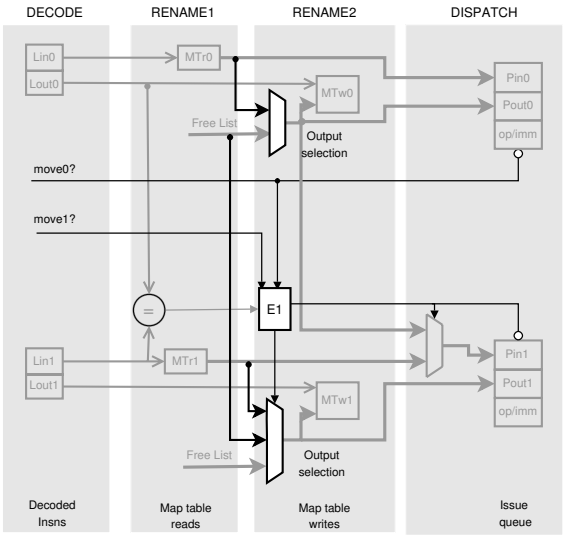
If we were to allow all four scenarios, the complexity of individual output selection muxes would grow as $2N+2$: I_2's mux would have six inputs, I_3's eight, and so on. However, because the fourth scenario is extremely rare we ignore it; when it does occur we collapse I_0 and treat I_1 as a non-move, taking its output from the free list. This decision is performed by the logic block $E1$ which internally computes $(move_1 \& (!move_0 \mid !dependence))$. With this simplification, output selection mux complexity grows only as $N+2$. I_1's output selection mux has only three inputs, I_2's will have four, I_3's five, and so on.

Adding RENO_{CF}. Figure 4c shows the changes required to implement RENO_{CF} over RENO_{ME}. RENO_{ME} is grayed to highlight the new structures and paths. RENO_{CF} adds virtually nothing to the core renaming circuit, i.e., physical register name manipulation path. From the point of view of output selection, constant folding is identical to move elimination. Move elimination sets an instruction's output register to its input register; constant folding does the same. RENO_{CF} does require a circuit that parallels the physical register name path but manipulates immediates and displacements instead. We use the term *immediate* to mean an immediate value in an instruction and the term *displacement* to mean immediates accumulated by RENO_{CF} in the extended map table.

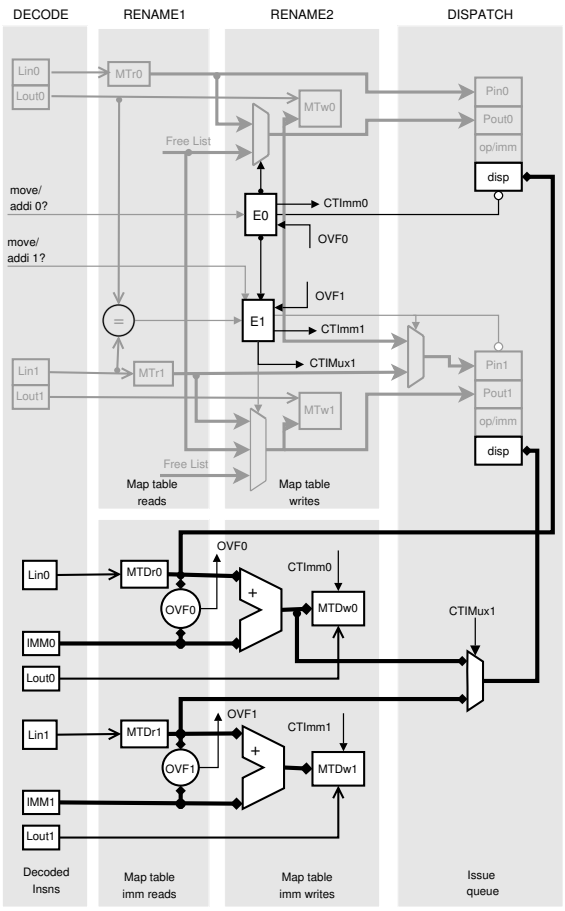
The *displacement path* parallels the register name path in structure and action. In RENAME1, displacements are read from the map table (MTDr). Simple logic examines the upper two bits of the instruction immediate and the map-table displacement to conservatively check for displacement overflow. If potential overflow is detected, the folding operation is canceled. The overflow signals (ovf_0



(a) Non-RENO superscalar pipeline



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



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

Fig. 4. Simplified pipeline, for two instructions, each with one input and an output

and ovf_1) are sent to the elimination logic blocks in the register name path.

In RENAME2, displacements (from the map table) and immediates (from the instructions) are added using narrow (16-bit) adders and written to the corresponding extended map-table entries (MTDw). Notice, unlike the register name path, there are no output selection muxes for displacements. An output mapping can only have a non-zero displacement if the instruction is eliminated, i.e., if it is a register-immediate addition or a move that depends on an older register-immediate addition. By definition, instructions that generate new values have map-table displacements of zero, so the inverses of the elimination signals (i.e., $!E0$ and $!E1$) can serve as the map-table displacement clear signals. But even if an instruction *is* eliminated, its output mapping's displacement can *only* be the sum of its input register's displacement and its immediate, such that a two-input adder suffices. This simple implementation is a direct result of our decision to disallow the folding of dependent *addi*'s in a single cycle. Without this simplification, a fully-general RENO$_{CF}$ renamer would have to be capable of combining immediates for any subset of the N instructions currently in RENAME2. This would require adders with as many as $N+1$ inputs (for the youngest instruction in the group) as well as N^3 logic to control which displacement inputs to each adder should be active.

Finally in DISPATCH, we route an additional displacement into the issue queue entries of non-eliminated instructions. For I_0 there is no choice, the additional displacement is the one attached to the input register. For I_1, we perform the displacement equivalent of input-fixup:

- If I_1 does not depend on I_0, we choose the displacement from I_1's register input.
- If I_1 depends on I_0, we choose the displacement from I_0's register output.

Again, the number of inputs to the individual displacement-fixup muxes grows as $N+2$.

Adding RENO$_{CSE+RA}$. RENO$_{CSE+RA}$ adds an input to each output-selection mux on the physical register name path, corresponding to the physical register output from the IT tuple. The logic to select this input is the "integration test" [21] which compares the physical register input (and

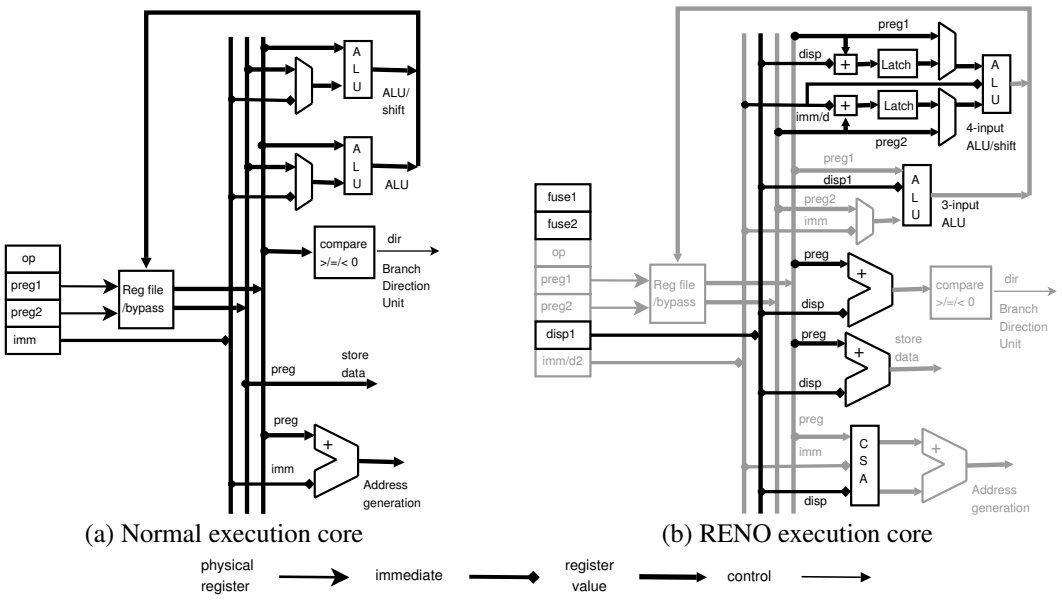


Fig. 5. Simplified view of the execution engine

displacement if RENO$_{CF}$ is implemented) from the map table with the one(s) from the tuple. The results of these comparisons feed the elimination logics, $E0$ and $E1$. Renaming pipeline diagrams for RENO$_{CSE+RA}$ (i.e., register integration) can be found elsewhere [24].

Summary. RENO starts with a MIPS R10000 renamer and adds output selection logic to each instruction slot. This logic includes a three-input mux (four-input if RENO$_{CSE+RA}$ is also implemented) and control which is derived from the existing dependence-check signals and additional signals from the decoder regarding the collapsibility of each instruction. RENO$_{CF}$ adds a circuit that parallels this renaming circuit in structure but accumulates and selects displacements, rather than physical register names.

Because RENO circuits are either pipelined with conventional renaming circuits (output selection) or parallel to them (displacement accumulation), a RENO implementation should not necessitate the addition of renaming pipeline stages. RENO does not form new critical loops [3] with conventional renaming and only slightly exacerbates (by adding output-selection muxes) the critical loop that already exists. RENO also does not require new paths between the renamer and pipeline stages with which it does not otherwise interact. A RENO implementation that includes RENO$_{CF}$ also enables a simple implementation of the most expensive RENO optimization, the table-driven RENO$_{CSE+RA}$.

3.3 RENO$_{CF}$ Execution Core Modifications

RENO$_{ME}$ and RENO$_{CSE+RA}$ change register renaming itself, but do not require changes to the execution engine. RENO$_{CF}$ changes the map-table format, creating denormalized values which must subsequently be normalized by fusion, and thus requires execution-core changes. These changes are limited in scope and impact.

Figure 5a shows a simplified integer side of a conventional scalar execution core. Issue queue entries include input and output physical register names, one immediate, and one operation descriptor. The register file has two read ports and one write port. The functional units include two ALUs, one of which can also perform shifts, a branch direction comparison unit, the store data path, and the load/store address generation unit. All functional units have two inputs.

Figure 5b shows the modifications required by RENO$_{CF}$; unmodified structures are grayed. Arguably the most important components—the physical register file and the register bypass network—are unchanged, as even fused instructions have at most two register inputs. RENO$_{CF}$ requires the following changes:

- An additional displacement field (*disp*) in each issue queue entry and a path from this field to the functional units.
- Two additional bits ($fuse_0$ and $fuse_1$) per issue queue entry indicating whether either register input must be normalized before the primary operation is executed.
- Enhancements to the functional units themselves.

The functional unit enhancements are as follows. The simple ALU (the one that does not perform shifts) and the load/store address generation unit, both originally two-input units, are extended to three-input units that accept one de-normalized input and one conventional (i.e., register *or* immediate) input and perform *addi-X* operations. Via appropriate use of carry-save adders, these conversions are assumed to have no impact on clock cycle time or the latency of these units.

The multiply/divide unit (not shown) and the general ALU/shifter are extended to four-input units that can accept two de-normalized inputs. These units use conventional adders (not carry-save adders) to normalize their inputs and incur a one-cycle execution penalty if normalization on either input is required.

Finally, we place two-input adders on the branch direction computation path and on the store data path. The latter normalizes register/displacement pairs before writing

them to the store queue (en route to memory) and extends store data latency by one cycle.

3.4 Recovery and Precise State under RENO$_{CF}$

Processors with conventional register renaming recover from mis-speculation by some combination of map table checkpointing and re-order buffer rollback. The existing machinery extends naturally to support RENO$_{CF}$: where there is a physical register name that could potentially be used in recovery (the map table checkpoints and the output physical register name in each re-order buffer entry) that name is augmented with a displacement. RENO$_{CF}$ does not change the rollback/checkpoint-restoration algorithm itself.

RENO$_{CF}$ also supports precise interrupt state, despite the fact that it defers operations to future instructions. Although interrupts can occur while some mappings are denormalized, program state always looks as if each operation occurred in its intended spot in program order. There are two keys to this behavior. First, instructions that logically follow an interrupt are correctly rolled back and unoptimized. Second, interrupt handler instructions also execute on the RENO pipeline so any attempt by them to either operate on a de-normalized register or store that register transparently performs a normalization step.

4 Experimental Evaluation

We present a simulation-driven evaluation of RENO. We begin by investigating RENO's instruction elimination effects and its performance impact. We then evaluate our chosen division of labor between RENO$_{CF}$ and RENO$_{CSE+RA}$. Finally, we show that RENO's capacity and bandwidth amplification effects can be used to offset reductions in physical register file size and execution bandwidth.

4.1 Methodology

Our benchmarks are the SPEC2000 integer and MediaBench [16] programs. We compile them for the Alpha EV6 architecture using the Digital OSF compiler with -O4 optimizations. We run all programs to completion: SPECint on their training input sets with 2% periodic sampling and 10M instructions per sample; MediaBench on their supplied inputs with no sampling.

Our timing simulator is based on the SimpleScalar Alpha AXP ISA and system call modules [5]. It models a dynamically scheduled superscalar processor with pointer-based (i.e., MIPS R10000-style) register renaming. The on-chip memory hierarchy 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, 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. 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 dual-ported 512-entry 2-way set-associative IT with a total size of 8KB.

4.2 Instruction Elimination Rate and Speedup

Figure 6 shows RENO's instruction elimination rates (striped bar) and the corresponding performance improvements on 4-way and 6-way issue processor configurations (solid bars 4 and 6). Each bar is split into a vertical stack which isolates the effects of the different RENO optimization. The bottom portion corresponds to RENO$_{ME}$, the middle to RENO$_{CF}$, and the top to RENO$_{CSE+RA}$.

Elimination rates. With a few exceptions (*mcf* and *mesa*) RENO$_{ME}$ eliminates fewer than 8% of the dynamic instructions; the average is 4%. Register moves are compilation artifacts and a good compiler will generate few of them. RENO$_{CF}$ eliminates an additional 12% (SPECint) and 16% (MediaBench) of the dynamic instructions. Because they are used in many common address-generation and loop-control 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% (SPECint) and 3% (MediaBench) of the dynamic instructions. However, these are loads and so the benefit of eliminating them is greater relatively than the benefit of eliminating moves and additions.

Performance improvements. On a 4-way issue processor RENO improves the performance of SPECint programs by an average of 8% and of MediaBench programs by an average of 13% with peak speedups of 14% (*perl.scrabbl*) and 27% (*gsm.decode*). Average speedups relative to a 6-way issue baseline are naturally lower, 6% on SPECint and 11% on MediaBench. A wider baseline processor diminishes the relative impact of RENO's capacity and bandwidth amplification effects.

Our performance simulator includes a critical path model [10, 11]. It collects timing and dependency information for all retired instructions, then builds dependence graphs and computes maximum edge depth and edge breakdown for 1M dynamic instruction chunks. We use this model to isolate RENO's performance impact by comparing breakdowns of RENO and RENO-less configurations. Figure 7 shows critical path breakdowns for a baseline RENO-less machine (*B*), a machine with RENO$_{ME}$ and RENO$_{CF}$ only (*C*), and a machine with full RENO (*R*). The latency of each critical edge is added to one of

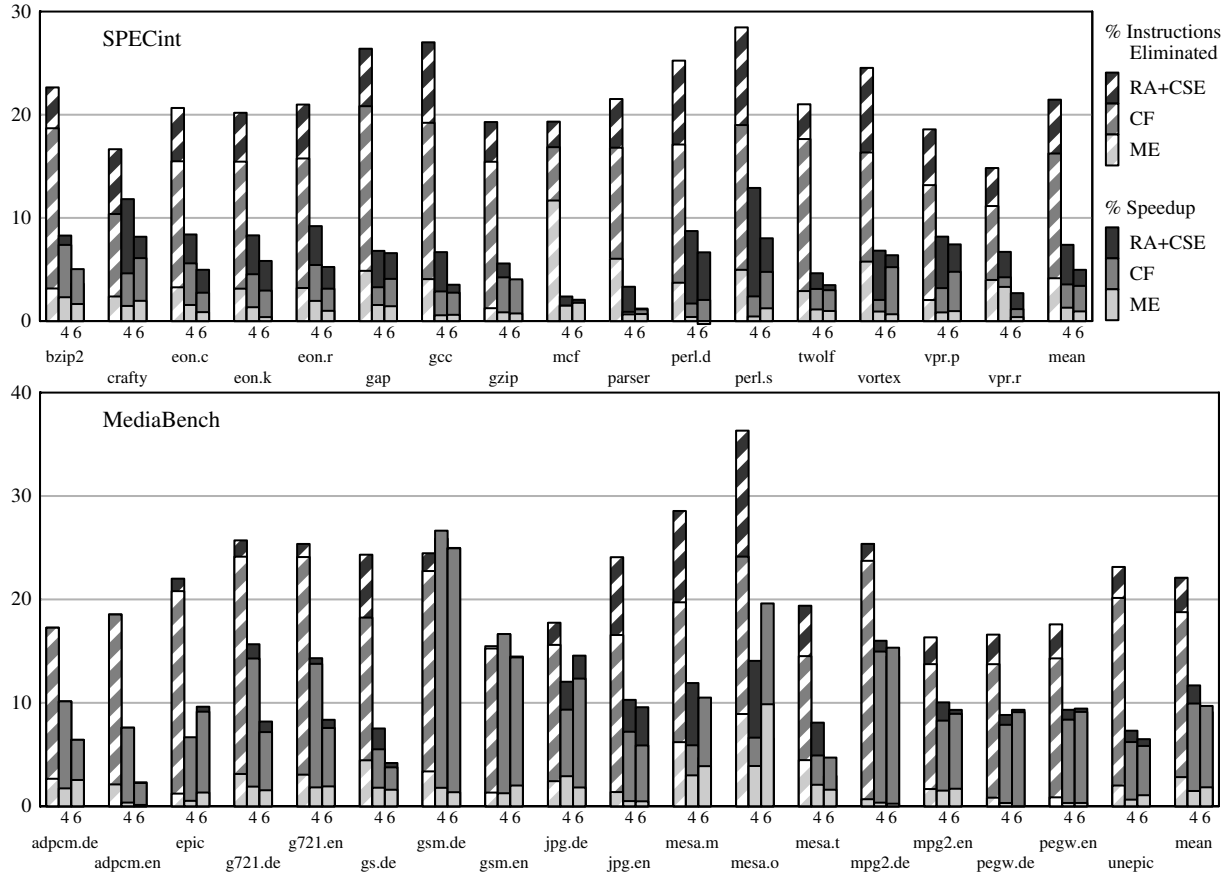


Fig. 6. Instruction elimination rates and performance improvements for 4- and 6- wide machines.

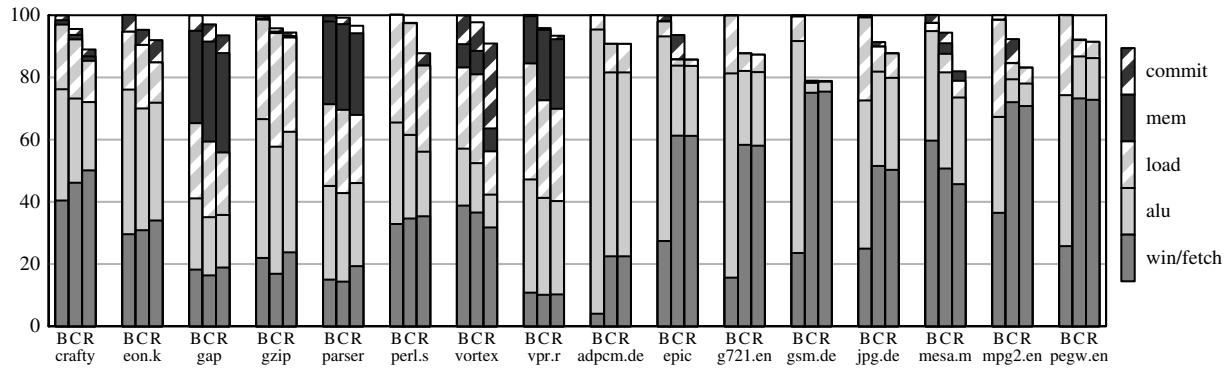


Fig. 7. Critical-path breakdown for baseline RENO-less execution (B), RENO$_{ME}$+RENO$_{CF}$ (C), and full RENO (R)

five buckets: *commit* (commit bandwidth), *mem* (memory dataflow latency), *load* (D$ and L2 dataflow latency), *alu* (integer dataflow latency), and *fetch/win* (fetch bandwidth, instruction cache misses, finite instruction window and issue queue).

RENO's most noticeable impact is on the execution latency (i.e., dataflow) components of the critical path: *alu* and *load*. MediaBench is ALU-critical, so RENO$_{ME}$ and RENO$_{CF}$ which attack this component have the greatest performance impact. In contrast, SPECint is more load-critical. Even though RENO$_{CSE+RA}$ accounts for less than 50% of the eliminated instructions, it has a much higher performance impact than RENO$_{ME}$ and RENO$_{CF}$ here. RENO$_{ME}$ and RENO$_{CF}$ act as a cheap mechanism that exposes more load elimination opportunities. Finally, no RENO technique attacks memory latency (*mem*), so RENO does not fare as well on benchmarks with significant memory components like *gap* and *parser*.

RENO has an interesting effect on fetch/window criticality. In most cases, RENO makes the processor more fetch bound. It eliminates so many instructions that fetch bandwidth can no longer keep up with highly amplified execution bandwidth and execution criticality "decays" into fetch criticality. This is particularly visible in MediaBench programs. However, in rare cases (e.g., *vortex* and *mesa.m*) RENO reduces fetch criticality. Here, RENO allows in-

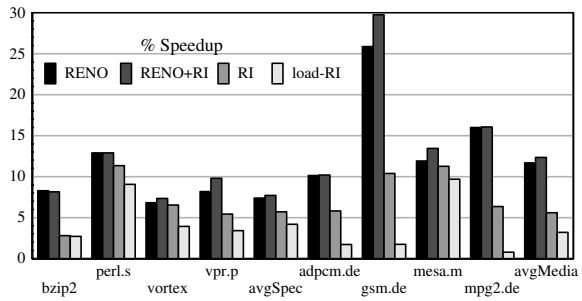


Fig. 8. Combining $RENO_{CF}$ and $RENO_{CSE+RA}$

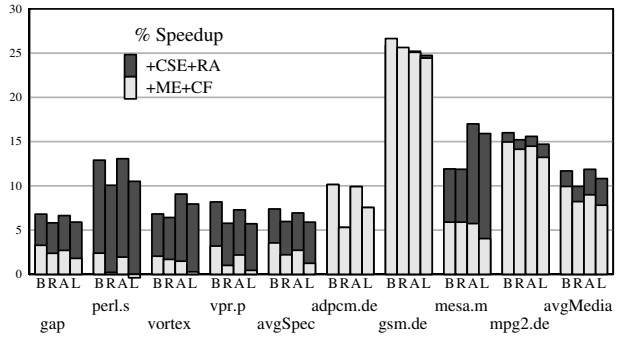


Fig. 9. RENO implementation effects

structions to dispatch more quickly by reducing contention for issue queue entries. These two effects are difficult to precisely isolate from one another in our critical path model because they involve a common set of critical path edges.

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

4.3 Combining $RENO_{CF}$ and $RENO_{CSE+RA}$

In our default RENO configuration, $RENO_{CF}$ collapses register-immediate additions and $RENO_{CSE+RA}$ eliminates only loads. This configuration (*RENO*) is the first bar in each group in the graph in Figure 8. We also examine three other possible divisions of labor.

The second configuration (*RENO+RI*) implements RENO and full register integration, i.e., it allows $RENO_{CSE+RA}$ to eliminate all instructions, not just loads. This configuration produces average performance improvements of less than 0.5% over RENO on both SPECint and MediaBench, with slight degradations due to increased $RENO_{CSE+RA}$ table conflicts on several programs (e.g., *bzip2*). It also requires 70% more IT accesses than RENO.

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

4.4 RENO Implementation Effects

The performance data presented so far assumes that arithmetic and logical operations can take a third (but not a fourth) input without requiring an additional cycle, and that RENO does not increase the number of register renaming pipeline stages. Although we have argued that these assumptions are reasonable, in this section we relax them. Figure 9 shows speedups for our baseline RENO configuration (*B*) and three others. All speedups are relative to a RENO-less processor with a two-stage renaming pipeline. In the *R* configuration, RENO adds one renaming pipeline stage. In the *A* configuration, all three- and four-input ALU operations *except* load/store address generation require an extra execution cycle. In the *L* configuration, all three- and four-input operations *including* load/store address generation take an additional cycle.

Adding a register renaming stage (R) elongates the branch misprediction penalty and induces an average performance penalty of 1.3%. RENO speedups degrade from from 8% to 6.8% for SPECint and from 13% to 11.6% for MediaBench.

Adding one execution cycle to all three- and four-input non-loads/stores (A configuration) degrades RENO gains by 1% for both SPECint and MediaBench. Adding an execution cycle to three-input loads and stores reduces gains by an additional 1.2%. These penalties effectively neutralize $RENO_{CF}$'s latency reduction benefit, but not its bandwidth and capacity amplification effect. Even under these assumptions, $RENO_{CF}$ (and RENO in general) maintains some performance benefit. One exception *perl.s*, for which $RENO_{CF}$ in the L configuration induces a slight slowdown. *perl.s*'s *addi*'s are more execute-critical than they are in other programs and have a higher fan-out (i.e., more consumers). "Bandwidth-only" $RENO_{CF}$ (L) saves one execution cycle in the producer, but adds one execution cycle to *each* consumer; in *perl.s* this is a poor trade.

4.5 RENO and Reduced Execution Cores

RENO's bandwidth and capacity amplification effects can be exploited to trade performance for reductions in the execution core.

Execution width. As shown in Figure 10, RENO can compensate for reductions in scheduling, execution, register read/write and bypass bandwidths. Our baseline 4-way issue configuration (4) can issue 3 integer operations per cycle, 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; one is limited to issuing a total of three instructions per cycle (configuration 3), the other has a total issue width of two (configuration 2).

In SPECint, $RENO_{ME}$ and $RENO_{CF}$ together can compensate for the loss of one issue slot and the associated functional unit, paths, and ports (3). Adding $RENO_{CSE+RA}$ results in an overall 5% gain over the baseline configuration (4). RENO cannot compensate for a 50% reduction in issue bandwidth (2)—neither is it expected to since it does not eliminate 50% of the dynamic instructions—but can restore performance back to within 6% of the 4-way issue baseline.

The more execution-bound MediaBench programs see a sharper decline in baseline performance when mov-

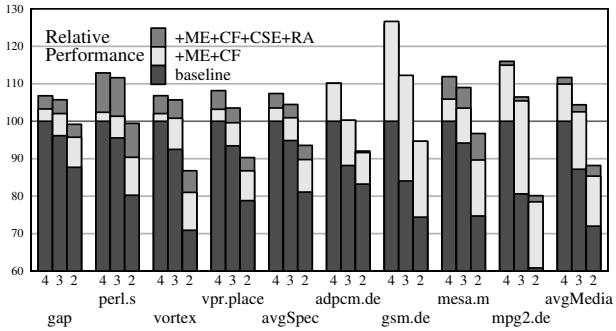


Fig. 10. RENO and reduced execution cores

ing from 4- to 3-way issue. However, precisely because they *are* execution-bound, RENO is especially effective at compensating. MediaBench programs execute 2% faster on a 3-way issue processor with RENO$_{ME}$ and RENO$_{CF}$ (no RENO$_{CSE+RA}$) than on a RENO-less 4-way issue processor. For the 2-way issue configuration, RENO recoups 18% of the overall performance loss of 29%.

Other simplifications. In addition to execution width, RENO can compensate for 40% reductions in physical register file and issue queue sizes, or for the presence of a two-cycle scheduler [3]. In the interest of space, we do not show data for these experiments. In both cases, RENO not only fully offsets the performance loss (11% for physical registers and 9% for the two-cycle sheduler) it actually produces a small net performance gain, an average of 2.5%. It is interesting to note that RENO tolerates scheduling loop latency in a fundamentally different way than other fusion techniques do [4, 12, 15]. RENO does not create multi-cycle compound operations (recall, we assume that most fusions do not increase execution latency), it simply eliminates many single cycle operations.

5 Related Work

Several lines of work are related to RENO.

Physical register sharing. There are two classes 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, 26]. Their main aim is to facilitate the early freeing of registers and the subsequent use of smaller, faster register files. The most aggressive of these techniques is physical register inlining (PRI) [18] which allows a physical register value to share storage with its own name if that value is narrow enough.

The second class of techniques detects sharing opportunities at the rename stage and exploits these to reshape the dynamic register dataflow graph and to avoid the execution of certain instructions. Unified renaming [13] uses reference counts to implement move elimination and memory-dependence prediction to implement store-load bypassing. An earlier implementation of store-load bypassing [19] does not use explicit reference counts and only operates if both store and load are simultaneously in-flight. Register integration [21] implements common-subexpression elimination in this framework and also includes a 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 point to an existing physical register. This is accomplished via temporary de-normalization, a combination of map-table extensions and a limited form of dynamic operation fusion.

Register tracking. Register tracking [2] uses the rename stage to fold stack-pointer-immediate additions and create a fast, speculative address-generation and issue path for stack loads. Unlike RENO, register tracking does not optimize the main instruction stream inline. Rather, it maintains and optimizes a bit of redundant state (a copy of the stack pointer) for a specific purpose.

Operation fusion. RENO$_{CF}$ performs a limited form of operation fusion, fusing register-immediate additions to dependent operations. RENO$_{CF}$ fusion exploits carry-save three-input adders to provide a single cycle latency reduction in the common fusion case. This form of latency reduction is similar in spirit to fused multiply-add and collapsing ALUs [27].

Recent proposals for dynamic [15] and static [12] pairwise instruction fusion target scheduling loop latency and scheduler capacity and bandwidth, but not dataflow graph execution latency. RENO targets all three. Dataflow mini-graphs [4] use static multi-instruction fusion to amplify execution capacity and bandwidth, including register file and bypass capacity and bandwidth. They also support a different form of limited fusion (collapsing ALU pipelines) to reduce execution latency.

Dynamic instruction stream optimization. Optimization of the dynamic instruction stream can be done at the decode stage [14]. However, the inability to name physical registers at that stage limits the kinds of optimizations that can be applied. RENO operates on physical register names and can perform a wider range of optimizations.

A more powerful optimization model is the offline optimization of cached traces used in rePLay [8]. The offline setting enables optimizations like dead code elimination that require backwards-dataflow analysis (e.g., liveness analysis). rePLay also makes traces atomic, removing the constraint that transformations must be correct on all paths. These features compensate for the fact that rePLay operates on logical registers.

RENO is similar to Continuous Optimization (CO, our acronym) [9], a recently proposed technique that also modifies register renaming to perform move elimination, redundant load elimination, store-load bypassing, and constant folding. RENO and CO differ in implementation philosophy: CO is *primarily* value-based while RENO is *purely* name-based. Because value equivalence subsumes name equivalence [25] (i.e., two values can be equal even if two names are not) CO has a broader optimization scope. However, RENO arguably has a simpler implementation.

CO tries to execute as many instructions as it can in the renamer itself. To do this, it maintains a table that maps logical registers to their values. This table is filled as values are computed by the execution core. At rename, instructions whose inputs are available in this table are executed immediately and their results written into the register file directly, their dataflow graph latency effectively reduced to zero. The parallel map table also contains a symbolic

description of the dataflow that computes each register. This description is used to perform partial calculations and dataflow height reductions even when input values are unavailable. CO adds complexity to register renaming (effectively a simple in-order execution pipeline) and adds paths out of the execution core and into the physical register file but leaves the core otherwise unmodified.

In contrast, RENO reduces dataflow graph height using *only* physical register name identities. When RENO does compute, it does so only on immediates using fusion to defer computation on full values to the execution core. RENO adds a little complexity to renaming and a little (three-input adders) to the execution core.

6 Conclusion

RENO is a modified MIPS-R10000 style register renamer that uses map-table "short-circuiting" to implement dynamic counterparts of several static optimizations: move elimination, common subexpression elimination, register allocation, and constant folding. RENO unifies several previously proposed techniques [13, 19, 20, 21, 23] into a single framework and adds a dynamic implementation of constant folding that uses an extended map table format and a simple form of operation fusion.

Cycle-level simulation on the SPEC2000 integer and MediaBench benchmarks shows that RENO can dynamically eliminate 22% of the dynamic instructions. Dataflow dependences are collapsed around eliminated instructions, improving performance by averages of 8% and 13%, respectively. Because eliminated instructions do not consume issue queue entries, physical registers or issue bandwidth, RENO can alternatively be used to maintain performance with a scaled-down execution core.

We continue to study the potential of adding other optimizations to the RENO framework and of combining RENO with other techniques. One interesting combination is RENO and physical register inlining (PRI) [18]. PRI maps logical registers to either a physical register *or* a value. RENO maps logical registers to a physical register *and* a value. It may be possible to exploit this similarity to incorporate PRI's register reclamation functionality into RENO. We are also evaluating RENO's energy aspects.

Acknowledgements

The authors thank the anonymous reviewers for their comments. Anne Bracy, Milo Martin, and Marci McCoy Roth helped improve the final manuscript. This work was supported by NSF CAREER award CCF-0238203.

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] A. Bracy, P. Prahlad, and A. Roth. Dataflow Mini-Graphs: Amplifying Superscalar Capacity and Bandwidth. In *MICRO-37*, Dec. 2004.

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

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

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

[13] 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.

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

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

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

[17] 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.

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

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

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

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

[22] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *ISCA-32*, Jun. 2005.

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

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

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

[26] 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.

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