HeDGE: Hybrid Dataflow Graph Execution in the Issue Logic

Suriya Subramanian and Kathryn S. McKinley

Department of Computer Sciences
The University of Texas at Austin
{suriya,mckinley}@cs.utexas.edu

Abstract. Exposing more instruction-level parallelism in out-of-order superscalar processors requires increasing the number of dynamic in-flight instructions. However, large instruction windows increase power consumption and latency in the issue logic. We propose a design called Hybrid Dataflow Graph Execution (HeDGE) for conventional Instruction Set Architectures (ISAs). HeDGE explicitly maintains dependences between instructions in the issue window by modifying the issue, register renaming, and wakeup logic. The HeDGE wakeup logic notifies only consumer instructions when data values arrive. Explicit consumer encoding naturally leads to the use of Random Access Memory (RAM) instead of Content Addressable Memory (CAM) needed for broadcast. HeDGE is distinguished from prior approaches in part because it dynamically inserts forwarding instructions. Although these additional instructions degrade performance by an average of 3 to 17% for SPEC C and Fortran benchmarks and 1.5% to 8% for DaCapo Java benchmarks, they enable energy efficient execution in large instruction windows. The HeDGE RAM-based instruction window consumes on average 98% less energy than a conventional CAM as modeled in CACTI for 70nm technology. In conventional designs, this structure contributes 7 to 20% to total energy consumption. HeDGE allows us to achieve power and energy gains by using RAMs in the issue logic while maintaining a conventional instruction set.

1 Introduction

To attain high performance, superscalar processors seek to exploit Instruction Level Parallelism (ILP) with large instruction windows and dynamic scheduling algorithms. The instruction issue logic is thus a key component in their design.

Current instruction window designs use a monolithic Content Addressable Memory (CAM) because it implements broadcast efficiently for instruction wakeup. Unfortunately, CAM structures scale poorly with respect to latency and power. Increasing the size of CAMs to expose more ILP forces a tradeoff between ILP and the clock period; larger CAMs consume disproportionately more power, which forces a tradeoff between power and performance. This paper seeks scalable instruction issue logic to attain energy efficiency together with high performance.

Our solution replaces broadcasts in the issue logic with direct communication. Current issue logic performs broadcast when producer instructions complete, notifying dependent instructions waiting in the issue window. Prior work shows that there are few

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 308–323, 2009.

such consumers within the window [6]. We find that 94 to 96% of instructions produce a result for two or fewer consumers in windows ranging from 64 to 512 instructions. This observation motivates a design that uses dataflow encoding of dependent instructions in the window, i.e., *direct instruction communication* between producers and consumers, instead of a broadcast based on physical register names. Our design dynamically identifies and encodes consuming instructions during the register rename stage. When an instruction produces its result, the wakeup logic identifies consumers and marks them ready. Eliminating the need for a broadcast leads to an instruction window implementation that uses a Random Access Memory (RAM) instead of a CAM. RAMs offer two significant advantages over CAMs: they consume significantly less energy per access and have a faster access time.

We call our design Hybrid Dataflow Graph Execution (HeDGE) because it takes an intermediate point between conventional superscalars and dataflow ISAs, such as WaveScalar [24] and TRIPS [18]. HeDGE requires no changes to a conventional ISA. It dynamically converts dependences specified with register names in the ISA as follows. When a consumer enters the window, HeDGE register renaming adds the consumer to a wakeup list for the producer. This logic generates a dataflow encoding, but only for instructions in the issue window. HeDGE implements the wakeup list by adding target fields to the reservation stations. When the number of consumers exceeds the number of target fields, HeDGE introduces forwarding instructions. Dynamically inserting forwarding instructions differentiates HeDGE from prior approaches to direct instruction communication in conventional designs, which stall the pipeline [26], or continue to use some associative logic for the instruction window [13,21], or sacrifice more ILP to track consumers [19].

The contribution of this paper is the demonstration and design of a power efficient instruction window that supports many in-flight instructions by using a more scalable hardware structure. We measure HeDGE in a cycle-accurate simulator on SPEC CPU and DaCapo Java benchmarks. Given two to four target fields in the HeDGE reservation stations and a range of issue window sizes of 64 to 512, HeDGE requires 2 to 30% additional forwarding instructions on average. Although these instructions degrade performance by an average of 3 to 17%, they enable energy efficient execution in large instruction windows. Using CACTI to model RAM and CAM structures in 70nm technology, we find that the energy per access consumed by a HeDGE RAM is 98% less than a CAM. In a conventional design, prior work shows that the CAM-based instruction window contributes 7 to 20% to total energy consumption [4,10,11], and the contribution increases as a function of the window size. Assuming a conservative 10% contribution, we show that HeDGE configurations reduce total processor energy by an average of 6%. RAMs also offer faster access times, but we do not explore this benefit here. These results demonstrate the potential of HeDGE designs to improve power efficiency.

2 Related Work

This section describes related work in issue logic design that uses explicit dependence tracking, that reduces issue load, and that uses dataflow ISAs. We also provide a brief taxonomy of dependence encoding. We refer the reader to Abella et al. for a comprehensive survey of issue logic design [1].

Dependence tracking. The most closely related work seeks to use direct instruction communication to reduce the complexity of the issue logic in dynamically scheduled superscalar processors [13,19,21]. These approaches explicitly track register dependences between instructions and completely or partially avoid associative lookup during instruction wake up. Similar to HeDGE, these approaches rely on the observation that only a few dependent instructions are typically in the issue window at a time and therefore propagate result tags only to those instructions in the window. However, none of these approaches considered or modeled energy-delay benefits.

Similar to HeDGE, Önder and Gupta use a fixed fanout degree [19]. However, when the number of targets exceeds the fanout degree, they encode the chain of forwarding instructions together with the consuming operands. When an instruction executes, the hardware forwards its result to consumers and its input operands to other instructions needing the same value. Each value is forwarded on a separate cycle, whereas HeDGE inserts `MOV` instructions, and delivers all the target fields of `MOV` and other instructions at once by using additional logic.

Sato et al. use a RAM-based instruction window with a register file called the Dataflow Management Table (DMT) to keep track of dependences [21]. This scheme eliminates associative wakeup; however, they must checkpoint the DMT on every branch prediction, as the DMT might contain incorrect dependences after a branch misprediction. HeDGE instead uses the misprediction handling mechanism that already exists in a superscalar processor.

Huang et al. modify the instruction window to maintain dependence information between a producer with a single consumer within the window, and then wake up just the consumer, avoiding a broadcast [13]. If more than one consumer enters the window, the wakeup logic reverts to a conventional broadcast scheme. This hybrid design combines direct instruction wakeup and broadcast, but comes with additional complexity. HeDGE uses `MOV` instructions when there are multiple dependent consumers within the window. This design adds instruction overhead compared to Huang et al., but enables the use of RAM hardware and simplifies the instruction window design.

Reducing issue logic latency. To reduce the issue logic latency, a number of approaches perform some form of dependence-based pre-scheduling to reduce the number of instructions considered for issue every clock cycle [16,17,20]. Palacharla et al. performed an analysis of circuit delay of various structures in a superscalar processor, and showed that the wakeup and select logic is a key element of the processor's critical path [20]. They proposed the first dependence-based instruction window design where the issue queue is implemented as a set of FIFOs with only the head of the FIFOs considered for issue. Michaud and Seznec pre-schedule instructions based on dataflow order, grouping instructions based on the clock cycle at which they will issue, thereby reducing the number of instructions considered for selection [17]. Lebeck et al. identify instructions dependent on long-latency operations such as cache misses and move them to a larger buffer [16]. They move these instructions back to the issue queue when the long latency operation completes. The number of instructions in the issue queue is smaller, and thus the issue queue is faster. These approaches are orthogonal to HeDGE and can coexist with our approach.

A taxonomy of dependence encoding. The taxonomy in Table 1 classifies Von Neumann and dataflow architectures according to the way they specify dependences between instructions in the ISA and between instructions in the issue window. Conventional out-of-order superscalar processors like the Alpha 21264 [15] use a Reduced Instruction Set Computer (RISC) instruction stream that encodes dependences between instructions using register names. The initial stages of the pipeline use register renaming to eliminate write-after-read and write-after-write dependences. Read-after-write dependences between instructions within the window are specified using physical register names.

WaveScalar, TRIPS, and other dataflow machines directly encode dependences in the ISA to exploit the inherent efficiencies of dataflow execution [7,18,24]. In a dataflow ISA, the compiler must explicitly specify dependences between instructions using target instruction identifiers. The execution model maps instructions to execution units on a distributed substrate, preserving the dependence information encoded in the ISA. In WaveScalar and TRIPS, both the ISA and microarchitecture use instruction identifiers to specify dependences. HeDGE exploits some of the same efficiencies, but in the context of a conventional ISA.

Table 1. Taxonomy of dependence encoding

		Instruction Window Encoding	
		Register names	Instruction names
ISA encoding	Register names	Alpha 21264 [15], Pentium	Huang et al. [13], Sato et al. [21], HeDGE
	Instruction names	None	Dataflow machines [7], WaveScalar [24], TRIPS [18]

3 Background

This section describes a conventional superscalar pipeline, with the register renaming and instruction wakeup, to provide context and motivation for our approach.

Figure 1 depicts the pipeline stages for dynamic instruction scheduling in an out-of-order superscalar processor. The frontend of the processor (not shown) fetches, decodes, and transfers instructions to the rename stage, which keeps track of instructions by reserving reorder buffer entries, reservation stations, and physical registers. The issue stage holds instructions in reservation stations, waiting for their input operands to become available. The select logic chooses candidates for execution, from ready instructions whose input operands are all available, based on availability of execution units and other policy considerations such as age of the instruction and criticality of the instruction [8]. Instructions selected for execution, read values from the register file and execute on appropriate functional units.

Register renaming. The register renaming stage updates a Register Alias Table (RAT) that maps architectural register names to physical register names. The rename stage eliminates all write-after-write and write-after-read register dependences by mapping the write target to a unique unused physical register location. The rename logic uses this

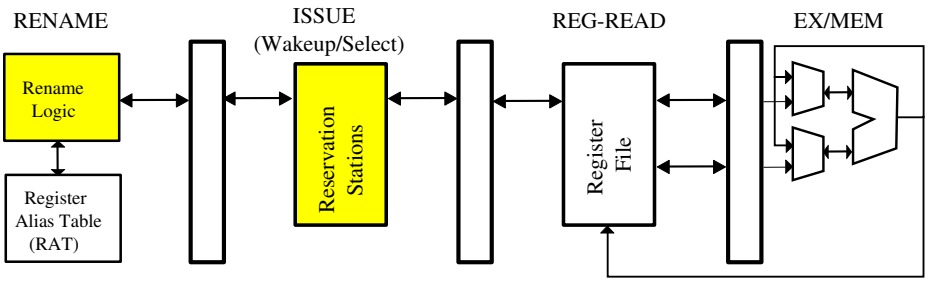


Fig. 1. Superscalar pipeline stages (HeDGE modifies the shaded parts of the pipeline)

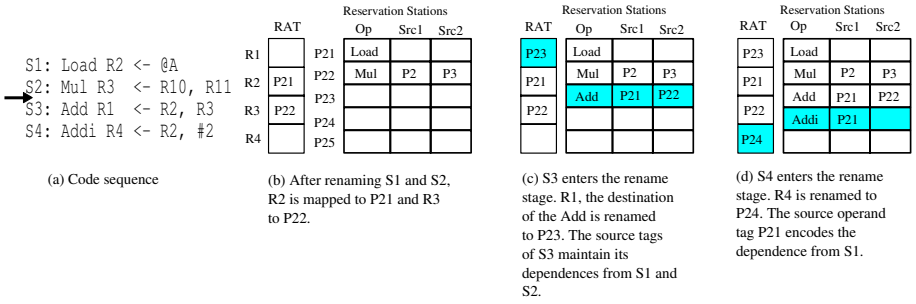


Fig. 2. Conventional superscalar pipeline register renaming example

physical register name to satisfy any subsequent reads from the original architectural register. Instructions speculatively issue and write to physical register storage, with the value becoming part of the architectural state only after the instruction commits. The wakeup logic uses physical register tags to check availability of operands that were not ready during register renaming.

Register renaming example. Figure 2 walks through a simple code sequence, showing the contents of the RAT and reservation station entries at each clock cycle assuming a 1-wide pipeline. Each diagram shows the RAT indexed from `R1` through `R4`, and reservation stations indexed by physical register names `P21` through `P25`. The shaded entries indicate those written in the current clock cycle. Figure 2(b) shows the state after renaming the `Load` and `Mul` instructions. Physical register `P21` maps to `R2`, the destination architectural register of the `Load`. Similarly, `R3` maps to `P22`. S3, the `Add` instruction enters the rename stage next. The rename logic allocates a new physical register `P23` to the output register `R1`. The source physical register tags respectively contain `P21` and `P22`, and maintain the read-after-write dependences from the `Load` and `Mul` instructions as shown in Figure 2(c). Similarly in Figure 2(d), the `Addi` enters the rename stage, and its output register `R4` is mapped to `P24`. Its source tag P21 encodes the dependence on S1.

Instruction wakeup. The wakeup logic is a significant source of complexity for out-of-order superscalar processors. The issue stage uses the source physical register tags set

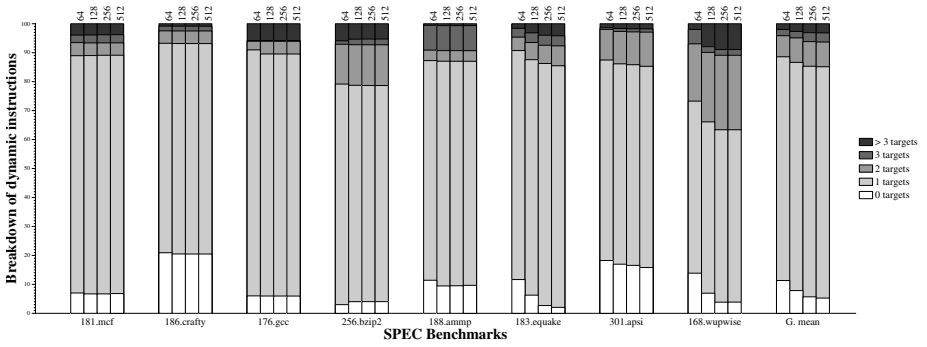


Fig. 3. Breakdown of dynamic instructions based on the number of dependent instructions within the window for instruction window sizes ranging from 64 to 512

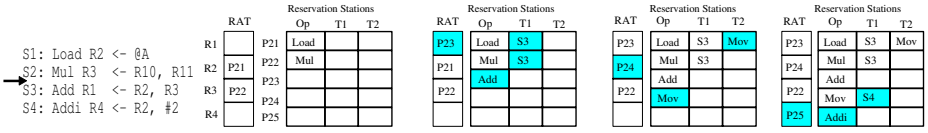
by the rename stage to wake up dependent instructions waiting in reservation stations. Just before an instruction finishes executing, it broadcasts its destination physical register tag to a common result tag bus. Waiting instructions snoop this bus and notify the *select logic* when all their operands are available. The select logic chooses candidate instructions for execution based on some heuristic. The wakeup and select logic of the instruction window is a key component in the critical path for an out-of-order superscalar processor [20]. The tag comparisons performed every cycle are a main source of complexity [20] and power dissipation in the instruction window [4,9].

4 Hybrid Dataflow Graph Execution (HeDGE)

To show the potential of direct instruction communication in the instruction window, Figure 3 shows the dynamic distribution of this communication. We measured performance on 17 of 21 C and Fortran SPEC CPU 2000 benchmarks [22] and 7 of 11 Java programs from the DaCapo benchmark suite (version `dacapo-2006-10`) [3] on which our baseline simulator currently works. We simulated the SPEC programs using the SimpleScalar 3.0 tool suite [5] for the Alpha ISA to simulate a 4-wide dynamically scheduled superscalar processor with varying window sizes, and the DaCapo programs running on JikesRVM using Dynamic SimpleScalar [14] for the PowerPC ISA. Due to space constraints, we present geometric means and representative results. A companion technical report presents all benchmark results [23]. The figure breaks down dynamic instructions based on the number of dependent instructions within the window. These results show that 94 to 96% of instructions produce a result for two or fewer consumers in windows ranging from 64 to 512 instructions, promising an efficient alternative.

4.1 Design

We leverage this observation with an instruction window design which explicitly keeps track of dependent instructions by adding target fields to the reservation stations. In a HeDGE window, producer instructions explicitly encode dependent consumer instructions, like in a dataflow machine. The HeDGE design only requires changes to the



(a) Code sequence

(b) Initially, R2 is mapped to P21, and R3 to P22. The Load and Mul instructions have been renamed.

(c) R1, the destination of the Add is renamed to P23. The target fields of S1 and S2 now point to S3.

(d) The parent of S4 has only one target field left. We introduce a Mov.

(e) R4, the destination of the Addi is renamed to P25. The Mov now targets S4.

Fig. 4. HeDGE register renaming example

rename and issue stages in an out-of-order pipeline. These stages are highlighted in Figure 1. We first summarize the key components in our design and then describe each component in detail.

When a consumer enters the instruction window, HeDGE first translates the architectural registers to physical register names, and then uses the physical name to dynamically identify any producers already in the window. It then adds the consumers to the producers' list of targets, stored in reservation stations. If the target fields are exhausted, HeDGE inserts `MOV` instructions. The `MOV` instructions and their target fields fan out values to multiple consumers when necessary. This process explicitly encodes read-after-write register dependences between instructions. This additional complexity in the rename stage results in simpler wakeup logic. The HeDGE wakeup logic looks up dependent instructions in the target fields of the reservation stations and sends the result tag only to these consumers.

Rename stage. Like in a conventional pipeline, HeDGE's rename stage maps architectural registers to physical register names for every instruction. In addition, it looks up an instruction's physical register operands in the RAT. If there is no entry for an operand, the rename stage marks the input operand as ready. Otherwise, an entry in the RAT provides the identifier for the producer. HeDGE adds this consumer to the producer's list of dependent instructions. This step dynamically encodes read-after-write dependences. Each reservation station has a small, fixed number of target fields, and there may be more consumer instructions within the window than target fields. Instead of stalling the pipeline [26], HeDGE introduces `MOV` instructions into the pipeline to track multiple consumers. We describe and use a simple algorithm that inserts a linear chain of `MOV` instructions. Although we do not evaluate it here, the renaming logic could create a tree of `MOV` instructions to fanout values in parallel.

Register renaming example. This section illustrates how HeDGE renames registers and introduces `MOV` instructions into the pipeline with the example from the previous section. Figure 4 is similar to Figure 2 but shows reservation stations with target instruction fields instead of those for source registers.

Figure 4(b) shows the state after renaming `S1` and `S2`. Renaming assigns physical register `P21` to `R2`, the destination architectural register of the `Load`. Similarly, it assigns `R3` to `P22`. Next, the `Add` instruction enters the rename stage. The rename stage allocates a new physical register `P23` to output register `R1`. For each input operand, it looks up the producer instruction identifier in the RAT. The rename stage then adds the current

instruction identifier to the target field of each producer. Figure 4(c) shows these updates by shading the new RAT and reservation station entries.

When `S4`, the `Addi` instruction, enters the rename stage, its producer, `S1`, has only one target field left. To accommodate more potential future consumers of `S1`, the rename stage inserts a `MOV` instruction and puts the `MOV` instructions identifier in the producer's target field. To make the `MOV` instruction the new producer of `R2`, it changes the RAT entry for `R2` to `P24`. Figure 4(d) shows this intermediate step. This process is semantically equivalent to adding `Mov R2 <- R2` at this point in the program. The `MOV` introduces a bubble in the pipeline. In the next cycle, the rename logic inserts `S4`'s instruction identifier in the `MOV`'s target field and inserts `S4` in the reservation station, as shown in Figure 4(e).

Instruction wakeup. We now describe the instruction wakeup logic. The key difference between a conventional out-of-order processor and HeDGE lies in how producer instructions communicate availability of an operand to the wakeup logic. In HeDGE, the wakeup logic does not snoop the result bus for matching physical register tags. Instead, it directly notifies consumer instructions as producers complete. The wakeup logic indexes the reservation station table by the target fields of the producer instruction, notifying each consumer that an input operand is available. The select and execute logic in HeDGE is the same as a conventional processor; it chooses which of the ready instructions to schedule for execution and executes them on functional units.

4.2 Speculation with HeDGE

HeDGE supports existing misspeculation recovery mechanisms in a straightforward manner. Just like in conventional processors, branch instructions trigger a RAT checkpoint. When the hardware detects a branch misprediction, it squashes instructions along the mispredicted path. If a producer is on the mispredicted path, all its consumers must also be on the mispredicted path, and the hardware squashes all of them. If only the consumer in a dependent chain needs to be squashed, its producers' target fields become invalid. To address this problem, HeDGE stores *instruction numbers (inums)* together with consumer identifiers in the target fields, and only wakes up consumer instructions when inums match.

4.3 Design Tradoffs

This section discusses in more detail the implications of fixing the number of target fields and the consequent insertion of `MOV` instructions.

The number of target fields determines the number of `MOV` instructions HeDGE will insert; fewer target fields require more `MOV` instructions, but fewer ports and a simpler wakeup logic. Because the issue logic cannot predict in advance whether it will need a `MOV` instruction or not, each reservation station entry must have at least two target fields, one for a consumer and one for a `MOV` instruction to propagate the value. A design that does not dynamically insert `MOV` instructions or has a single target field must stall the pipeline when an instruction runs out of targets. HeDGE avoids such pipeline stalls, by reserving the last target field for `MOV` instructions. In the case where there is

exactly one additional consumer, the MOV is unnecessary. However, our conservative policy simplifies the logic for introducing MOV instructions and handling them in the other stages of the pipeline.

The semantics of a dynamically introduced MOV instruction are the same as a MOV instruction of the form "MOV Ra, Ra" where Ra is the architectural register name. A MOV instruction behaves just like any other MOV instruction within the pipeline, occupying reservation station slots and reducing the effective size of the window. They also reduce the effective issue and commit width of the processor. Finally, whenever a MOV instruction forwards a data value, it introduces a bubble in the pipeline. Section 5.2 quantifies these effects.

A HeDGE design must choose a sweet spot between increasing complexity to support more targets and consequently inserting fewer MOV instructions, or inserting more MOVs to reduce complexity.

5 Evaluation

This section describes our cycle-accurate and power-modeling simulation methodologies and results. To demonstrate the tradeoffs in the HeDGE design, we measure the number of forwarding instructions HeDGE introduces and their effect on total performance for a range of HeDGE configurations. We then model the power and energy characteristics of the circuits in a conventional CAM instruction window and in a HeDGE RAM instruction window structures using CACTI 4.2 [25]. We use prior research that specifies 7 to 20% of total energy consumption of a superscalar processor is due to the dynamic scheduling structures [4,10,11]. Over this range of values, we compare total power and energy-delay of HeDGE to a conventional design. We show that even with a conservative 10% contribution of the issue logic to total processor power, HeDGE configurations reduce total energy by an average of 6% for SPEC programs and 10% for DaCapo programs.

5.1 Methodology

We extend sim-outorder, a cycle-accurate simulator from the SimpleScalar 3.0 tool suite [5] for the Alpha ISA to implement HeDGE, for executing C and Fortran programs. We use Dynamic Simplescalar [14] for the PowerPC ISA for simulating Java programs running on JikesRVM [2]. The cycle-level simulator models an aggressive 4-way out-of-order superscalar microarchitecture. The simulator is execution-driven and accounts for instructions along the wrong path of a misspeculation. The memory hierarchy has two levels of caches with split L1 instruction and data caches and a unified L2 cache. HeDGE modifies the register renaming and wakeup logic to track a parent instruction's targets. We explore configurations with two, three, and four target fields in the reservation stations. Table 2 contains the simulation parameters, latencies, and branch predictor information.

We evaluate HeDGE on all the programs that successfully execute on our baseline simulators. We execute 17 of the 21 C and Fortran SPEC CPU2000 benchmarks [22]. We compiled all the SPEC benchmarks with Compaq's GEM compiler with full optimization for an Alpha 21264 machine. We used SimPoint 3.0 [12] to identify regions

Table 2. Processor parameters

Parameter	Value
Pipeline width	4
Instruction window/ LSQ sizes	64/32, 128/64, 256/128, 512/256
Branch Predictor	GSHARE with an 8-bit global history, and an 8K BTB
Branch Target Buffer	512 entries, 4-way associativity
Functional units	four integer ALUs, one integer MULT/DIV, two load/stores, four FP adders, one FP MULT/DIV
Latencies	1-cycle integer operations, 3-cycle multiply 2-cycle FP add, 4-cycle multiply 20-cycle integer divide (non-pipelined) 12-cycle FP divide (non-pipelined)
Split L1 I/D caches	64 KB, 2-way set associative, 64 byte lines, 1 cycle hit latency
Unified L2 cache	1 MB, 64 4-way set associative, 64 byte lines, 10 cycle hit latency
DRAM	100-cycle latency, bandwidth of 8 bytes per CPU cycle
HeDGE target fields	2, 3, and 4

of execution that characterize program behavior for a particular input set and simulated these regions. We evaluate seven Java programs from the DaCapo benchmark suite (version `dacapo-2006-10`) [3]. These programs executed 1 billion instructions after forwarding the initialization portion of the execution. In the following discussion, we present results for a subset of programs by including the geometric mean, high and low extremes, and representative samples in each of SPEC INT, SPEC FP, and DaCapo benchmark suites. We refer the reader to a technical report [23] for complete results.

5.2 Performance of HeDGE

This section quantifies the additional `MOV` instructions that HeDGE inserts, their effect on performance, and the contributions due to `MOV` instructions occupying window slots and pipeline bandwidth.

HeDGE introduces `MOV` instructions into the dynamic instruction stream to maintain register dependences when a parent instruction runs out of target entries. These `MOV` instructions behave like regular instructions, and occupy instruction window space and issue and commit bandwidth in the pipeline. Figure 5 plots the percentage of `MOV` instructions that HeDGE adds to communicate dependences for window sizes of 64 to 512 with two, three, four target fields in the reservation stations. We include `MOV`s along misspeculated paths as well, and compute them as a percentage of the total number of committed instructions.

As expected, the number of `MOV`s added decreases as the number of target fields increases. On average, two targets increase executed instructions by 20 to 30%, whereas four targets only increase executed instructions by 2 to 4% (the white portion in the final set of bars). In addition, increasing the instruction window size from 64 to 512 increases the number of `MOV`s. These `MOV` instructions cause a corresponding drop in

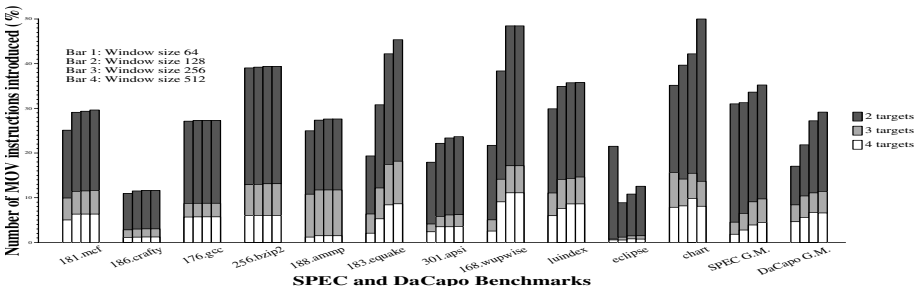


Fig. 5. Percentage HeDGE `MOV` instructions over non-speculative committed instructions in the base processor

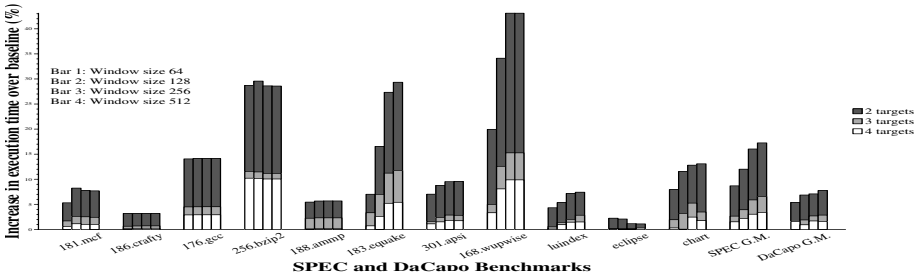


Fig. 6. Increase in execution time for HeDGE over the baseline processor

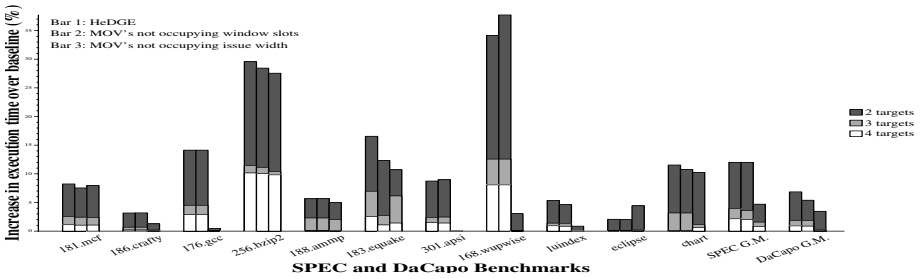


Fig. 7. Instruction window slots and pipeline bandwidth contributions of HeDGE `MOV`s. The first column shows the performance of HeDGE relative to the baseline, the second column shows HeDGE if `MOV`s do not occupy instruction window slots, and the third column shows `MOV`s not consuming pipeline bandwidth.

performance as shown in Figure 6. These plots compare the performance in simulated cycle counts of HeDGE over the baseline configuration. Cycle counts are a more appropriate comparison point than Instructions Per Cycle (IPC) in this work because HeDGE adds additional instructions. For a HeDGE implementation with only two targets, execution time increases by 17%. This number falls to 6.5% for a HeDGE implementation with three targets, and to less than 3.4% for four targets. The increase in execution time for

the DaCapo programs is even smaller, ranging from 8% with two targets, to 1.5% with four target fields. Although these results suggest using four targets rather than two in the reservation stations, four targets require four ports and parallel logic to wakeup four instructions at once. This increased complexity thus favors a lower numbers of targets, but performance favors more targets.

We now further quantify three effects of `MOV` instructions: (1) They occupy instruction window entries and reducing the effective size of the instruction window. (2) They utilize issue and commit bandwidth, which reduces the effective width of the pipeline. (3) They occupy execution units, conceptually introducing bubbles in the pipeline. To measure the first effect, we assign `MOV`s to their own window, thereby using the entire instruction window for other instructions. To measure the second effect, we model separate, special issue logic that only executes `MOV` instructions. Figure 7 plots increases in execution time over a baseline configuration with a moderately aggressive instruction window size of 128, for the SPEC and DaCapo benchmarks. The three columns in the figures show HeDGE, HeDGE when `MOV` instructions do not occupy instruction window slots, and HeDGE when `MOV` instructions do not consume pipeline width.

These results show that for both SPEC and DaCapo programs, `MOV` instructions occupying pipeline bandwidth is the main reason for HeDGE performance degradations; i.e., the third bar in which `MOV`s do not occupy execution is on average much lower than the other two. A few counterintuitive performance degradations occur when `MOV` instructions do not occupy instruction window slots, e.g., the second bar is higher than HeDGE for 168.wupwise (and 171.swim and 172.mgrid, not shown in the graphs). When `MOV` instructions reside in their own buffers, the effective window size for regular instructions increases and there are now more instructions in the window. As a result, HeDGE must add more `MOV` instructions to the pipeline, which utilize pipeline bandwidth and cause a drop in performance.

To execute a fanout instruction, the processor does not need Arithmetic Logic Units (ALUs) or commit width. Since the only purpose of `MOV` instructions is to wake up dependent instructions, the issue logic could include an additional bypass path that implements the `MOV` instructions, waking up dependent consumers.

5.3 Energy Characteristics

We used CACTI 4.2 to model the power and energy characteristics of a conventional CAM-based instruction window and a RAM-based HeDGE design. We use CAM entries with 64 decoded instruction bits and four ports—to support broadcasting up to four physical register tags every cycle. The HeDGE RAMs window adds two to four target fields to every instruction. For a 4-issue processor, the RAM requires four read ports, but eight write ports. Since each instruction has at most two operands, HeDGE needs two write ports to install the target fields in each operand producer for each issued instruction. Given two to four target fields, HeDGE uses eight to sixteen one-bit write ports to set the ready bits of consumer instructions. The number of ports is equal to the number of target fields in the reservation station times the issue width which indicates the maximum number of instructions HeDGE can wakeup in a single cycle.

Table 3 shows the energy consumed per access and leakage power, for 100, 70, and 45 nm technology nodes. CACTI does not currently provide leakage power for 45nm

technology. Although, the HeDGE RAM structure occupies more area than the CAM because it has more ports, the HeDGE RAM consumes 94 to 98% less energy per access than the CAM design. These results show that the CAM leaks 72 to 87% more power than the HeDGE RAM. For the HeDGE RAM, leakage power increases as a function of target fields because each field requires additional transistors.

Table 3. Energy per access (nJ)

IW Size	Energy per access (nJ)				Leakage power (mW)			
	Baseline	HeDGE			Baseline	HeDGE		
		2	3	4		2	3	4
100 nm technology								
64	0.336	0.016	0.014	0.020	14.619	1.965	2.578	2.472
128	0.524	0.019	0.020	0.029	21.703	4.142	4.489	4.530
256	0.932	0.026	0.027	0.030	42.582	7.796	8.501	10.824
512	1.748	0.036	0.038	0.041	84.340	15.823	17.237	21.890
70 nm technology								
64	0.149	0.007	0.007	0.009	60.959	9.339	10.184	11.630
128	0.227	0.008	0.009	0.013	82.736	19.313	21.010	21.727
256	0.403	0.011	0.012	0.017	162.874	37.107	40.540	43.352
512	0.756	0.017	0.018	0.021	323.149	71.437	78.303	89.341
45 nm technology								
64	0.058	0.002	0.003	0.004	Leakage power numbers not available			
128	0.091	0.003	0.003	0.005				
256	0.161	0.004	0.004	0.005				
512	0.303	0.006	0.006	0.007				

We now compare the overall power and energy-delay product of HeDGE against the baseline design. We use energy consumption data obtained from CACTI for the individual structures. We do not include leakage power when computing total energy-delay because how to estimate its contribution to total power is still an open research problem. We rely on prior results for the relative contribution of the instruction window to overall processor power [4,10,11].

Let $e_{Baseline}$ and e_{HeDGE} respectively be the energy consumed each clock cycle by the baseline design and HeDGE. Let $e_{Baseline,scheduler}$ and $e_{HeDGE,scheduler}$ respectively be the energy consumed each clock cycle for each structure. Let f be the contribution of dynamic scheduling towards the overall power consumption in the baseline design. Following Amdahl's law, we obtain $e_{Baseline} = e_{Baseline,scheduler}/f$ and $e_{HeDGE} = e_{HeDGE,scheduler} + (1 - f)e_{Baseline}$. The total energy (E) consumed while executing a program, by the baseline and HeDGE designs, are related by $E_{Baseline}/E_{HeDGE} = (e_{Baseline}.Cycles_{Baseline})/(e_{HeDGE}.Cycles_{HeDGE})$. This ratio is independent of clock frequency, and assumes that the baseline and HeDGE clocks run at the same frequency. We do not take into account that HeDGE structures have a faster access time and hence could be clocked faster.

For 70nm technology, a conservative 10% contribution of the issue logic to total power, and a 512-entry instruction window, Figure 8 plots the relative energy-delay for

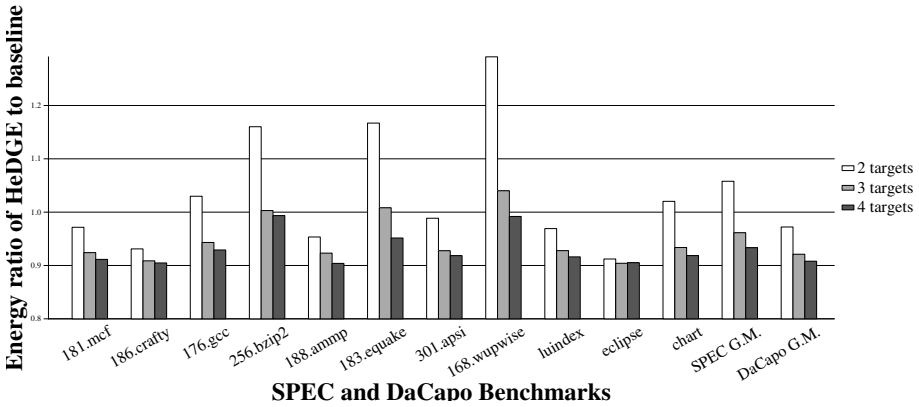


Fig. 8. Energy ratio of HeDGE to the baseline with a 512 instruction window

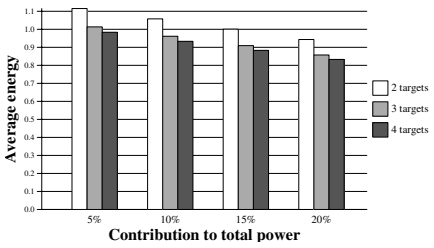


Fig. 9. Energy ratio as a function of the issue logic's total contribution to power

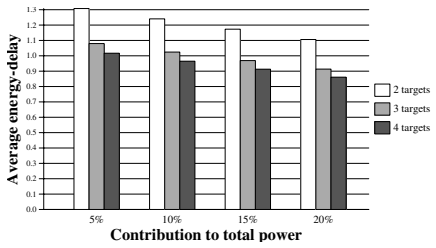


Fig. 10. Energy-delay ratio as a function of the issue logic's total contribution to power

HeDGE designs with two, three, and four target fields compared to the baseline superscalar processor design for all benchmarks. Figures 9 and 10 show the energy and energy-delay as a function of the contribution of the issue logic to total power for ranges from 5 to 20%. Each figure plots the geometric mean of all the benchmarks for 512-entry instruction windows with two, three, and four target fields for 70nm technology. These graphs show that even if a CAM-based processors consumes only 5% of total energy, a four-entry HeDGE RAM improves total energy and energy-delay. If current CAM-based designs are consuming 20% of total power, HeDGE offers significant advantages even with only two target fields.

6 Conclusion

Prior work has shown that the central CAM structure in the issue logic scales poorly with respect to power and latency, and that the issue logic is an integral component of the critical path in superscalar processors. We present HeDGE, a new, more scalable design for the instruction issue logic. HeDGE dynamically transforms instruction dependences implicitly encoded in the register names from a conventional ISA into explicit dependences

by adding target fields to the reservation stations and `MOV` instructions to the instruction stream. HeDGE modifies only the issue, register renaming, and wakeup logic. The main advantage of this design is that it naturally leads to the use of a RAM as the central structure in the issue logic instead of a CAM. We show that even without quantifying the cycle advantages RAMs offer, HeDGE offers substantial power improvements for the issue logic. Furthermore, these results translate to improvements in total processor power, energy, and energy-delay.

References

1. Abella, J., Canal, R., González, A.: Power- and Complexity-Aware Issue Queue Designs. IEEE Micro. 23(5), 50–58 (2003)
2. Alpern, B., Attanasio, D., Barton, J.J., Cocchi, A., Flynn Hummel, S., Lieber, D., Mergen, M., Ngo, T., Shepherd, J., Smith, S.: Implementing Jalapeño in Java. In: ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications, Denver, CO (November 1999)
3. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications, Portland, OR (October 2006)
4. Brooks, D., Tiwari, V., Martonosi, M.: Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In: International Symposium on Computer Architecture, Vancouver, British Columbia, Canada, pp. 83–94 (2000)
5. Burger, D., Austin, T.M.: The Simplescalar Tool Set Version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin (June 1997)
6. Canal, R., González, A.: Reducing the complexity of the issue logic. In: International Conference on Supercomputing, Sorrento, Italy, pp. 312–320 (2001)
7. Dennis, J.B., Misunas, D.P.: A Preliminary Architecture for a Basic Data-Flow Processor. In: International Symposium on Computer Architecture, pp. 126–132 (1975)
8. Fields, B., Rubin, S., Bodík, R.: Focusing Processor Policies via Critical-Path Prediction. In: International Symposium on Computer Architecture, Göteborg, Sweden, pp. 74–85 (2001)
9. Folegnani, D., González, A.: Energy-Effective Issue Logic. In: International Symposium on Computer Architecture, Göteborg, Sweden, pp. 230–239 (2001)
10. Gewnnap, L.: Intel's P6 uses Decoupled Superscalar Design. Microprocessor Report 9(2), 9–15 (1995)
11. Gowan, M.K., Biro, L.L., Jackson, D.B.: Power Considerations in the Design of the Alpha 21264 Microprocessor. In: Design Automation Conference, pp. 726–731 (1998)
12. Hamerly, G., Perelman, E., Lau, J., Calder, B.: Simpoint 3.0: Faster and More Flexible Program Phase Analysis. The Journal of Instruction-Level Parallelism 7 (September 2005)
13. Huang, M., Renau, J., Torrellas, J.: Energy-Efficient Hybrid Wakeup Logic. In: ISLPED 2002: Proceedings of the 2002 International Symposium on Low Power Electronics and Design, Monterey, California, USA, pp. 196–201 (2002)
14. Huang, X., Moss, J.E.B., McKinley, K.S., Blackburn, S.M., Burger, D.: Dynamic Simplescalar: Simulating Java Virtual Machines. Technical Report TR-03-03, Department of Computer Sciences, The University of Texas at Austin (February 2003)
15. Kessler, R.E.: The Alpha 21264 Microprocessor. IEEE Micro. 19(2), 24–36 (1999)

16. Lebeck, A.R., Koppanalil, J., Li, T., Patwardhan, J., Rotenberg, E.: A Large, Fast Instruction Window for Tolerating Cache Misses. In: ISCA 2002: Proceedings of the 29th annual International Symposium on Computer Architecture, Anchorage, Alaska, pp. 59–70 (2002)
17. Michaud, P., Seznec, A.: Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors. In: HPCA 2001: Proceedings of the 7th International Symposium on High-Performance Computer Architecture, Monterrey, Mexico (2001)
18. Nagarajan, R., Sankaralingam, K., Burger, D., Keckler, S.W.: A Design Space Evaluation of Grid Processor Architectures. In: MICRO 34: Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture, Austin, Texas, pp. 40–51 (2001)
19. Önder, S., Gupta, R.: Superscalar Execution with Direct Data Forwarding. In: International Conference on Parallel Architectures and Compilation Techniques, pp. 130–135 (1998)
20. Palacharla, S., Jouppi, N.P., Smith, J.E.: Complexity-Effective Superscalar Processors. In: ISCA 1997: Proceedings of the 24th annual International Symposium on Computer Architecture, Denver, Colorado, United States, pp. 206–218 (1997)
21. Sato, T., Nakamura, Y., Arita, I.: Revisiting Direct Tag Search Algorithm on Superscalar Processors. In: Workshop on Complexity-Effective Design (2001)
22. SPEC. Standard Performance Evaluation Committee, `http://www.spec.org`
23. Subramanian, S., McKinley, K.S.: HeDGE: Hybrid Dataflow Graph Execution in the Issue Logic. Technical Report 2008-42, Department of Computer Sciences, The University of Texas at Austin (2008)
24. Swanson, S., Michelson, K., Schwerin, A., Oskin, M.: WaveScalar. In: MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, San Diego, CA, pp. 202–291 (2003)
25. Tarjan, D., Thoziyoor, S., Jouppi, N.P.: CACTI 4.0. Technical Report WRL-2006-86, Hewlett-Packard Labs, Palo Alto (June 2006)
26. Weiss, S., Smith, J.E.: Instruction Issue Logic for Pipelined Supercomputers. SIGARCH Comput. Archit. News 12(3), 110–118 (1984)