WiDGET: Wisconsin Decoupled Grid Execution Tiles

Yasuko Watanabe* John D. Davis† David A. Wood*

*Department of Computer Sciences
University of Wisconsin—Madison
1210 W. Dayton Street
Madison, WI 53706
{watanabe, david}@cs.wisc.edu

†Microsoft Research — Silicon Valley Lab
1065 La Avenida
Mountain View, CA 94043
john.d@microsoft.com

ABSTRACT

The recent paradigm shift to multi-core systems results in high system throughput within a specified power budget. However, future systems still require good single thread performance—no longer the predominant design priority—to mitigate sequential bottlenecks and/or to guarantee service-level agreements. Unfortunately, near saturation in voltage scaling necessitates a long-term alternative to dynamic voltage and frequency scaling.

We propose an energy-proportional computing infrastructure, called WiDGET, that decouples thread context management from a sea of simple execution units (EUs). WiDGET's decoupled design provides flexibility to alter resource allocation for a particular power-performance target while turning off unallocated resources. In other words, WiDGET enables dynamic customization of different combinations of small and/or powerful cores on a single chip, consuming power in proportion to the delivered performance.

Over all SPEC CPU2006 benchmarks, WiDGET provides average per-thread performance that is 26% better than a Xeon-like processor while using 8% less power. WiDGET can also scale down to a level comparable to an Atom-like processor, turning off resources to reduce average power by 58%. WiDGET achieves high power efficiency ($BIPS^3/W$), exceeding Xeon-like and Atom-like processors by up to 2x and 21x, respectively.

Categories and Subject Descriptors

C.1.4 [**Parallel Architectures**]: Distributed architectures

General Terms

Performance, Design, Experimentation

Keywords

Power proportional computing, power efficiency, hardware, performance, instruction steering

ISCA'10, June 19-23, 2010, Saint-Malo, France.

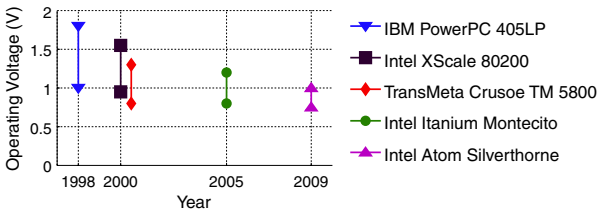


Figure 1. Operating voltage ranges

1. INTRODUCTION

Current trends in microprocessor design have appropriately recast Moore's law from the GHz race to the cores (threads) per die race. Designers trade off design complexity for power and area efficiency. Furthermore, ever increasing wire delays favor many small hardware structures over a few large monolithic structures. As a result, we have observed a paradigm shift to chips with multiple simple in-order cores, such as Intel's Larrabee [32] and Sun's Niagara [20]. Small cores deliver high thread-level parallelism and power efficiency, but Amdahl's Law dictates that we not ignore single thread performance [2,13]. Ignoring Amdahl's Law would lead to systems that are highly susceptible to sequential bottlenecks and/or fail to meet service-level agreements [28].

A key challenge in the multi-core era is developing an energy-proportional computing infrastructure and, at the same time, balancing system throughput and sequential thread performance. An Intel Nehalem processor, for instance, relies on dynamic voltage and frequency scaling (DVFS) to tackle the challenge [15]. By power gating a subset of cores, it can raise the voltage and frequency of the remaining cores to boost their performance without exceeding the power budget. Unfortunately, DVFS is no longer sufficient. Figure 1 shows that while maximum supply voltage has declined over the past decade, minimum supply voltage has remained almost the same (operating in the subthreshold regime is never energy-efficient [40]). This shrinking operating voltage range significantly reduces the benefit of DVFS.

We seek to provide an alternative approach to achieving energy-proportional computing. We propose WiDGET (Wisconsin Decoupled Grid Execution Tiles), a hardware design that decouples thread context management (i.e., instruction engines or IEs) from a sea of simple computation resources (i.e., execution units or EUs), loosely defining core boundaries. Rather than scaling voltage like DVFS, WiDGET scales cores up and down through global resource allocation, varying the number of enabled IEs and the number of EUs assigned to each IE. This decoupled design activates only the

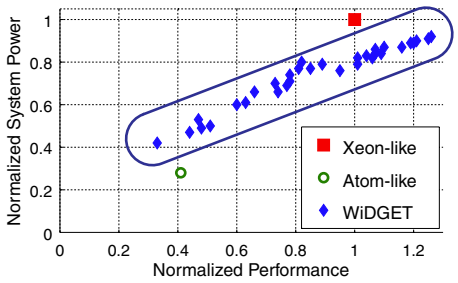


Figure 2. Power proportionality

computation resources needed for a particular power-performance target—turning off the rest to save power—permitting dynamically and individually customized cores on a single chip.

In this paper, we focus on WiDGET's single thread power and performance with emphasis on power proportionality. We define power proportionality as power dissipation in proportion to single thread performance, adapted from Barroso and Hölzle's definition of energy proportionality [4]. WiDGET achieves power proportionality through EU provisioning, under system software control, as demonstrated in Figure 2. With the maximum number of EUs, WiDGET provides 8% power savings for 26% better performance than a high performance Xeon-like processor (Section 5.1 presents configuration details) running SPEC CPU2006 benchmarks. With the minimum number of EUs, WiDGET uses 58% less power, approximating the power and performance of a low power Atom-like processor. In addition, between these two extremes, WiDGET affords a wide power-performance range.

WiDGET's decoupled modular design enables flexible EU configuration, but makes distributing instructions to the disjoint EUs a key challenge. By taking into account communication overheads, we show that simplified steering logic can yield good performance.

The key contributions of this paper are:

- A power proportional design enabled by EU modularity, allowing WiDGET to scale across the power-performance spectrum.
- A unified framework that dynamically exploits programs' parallelism within a power budget. WiDGET decouples thread context management from a sea of computation resources. We trade off complexity for power, replacing the monolithic, power-hungry out-of-order (OoO) issue logic in a conventional OoO machine with simple in-order EUs.
- WiDGET's high power efficiency (BIPS^3/W) exceeds the Xeon-like and Atom-like processors by up to 2x and 21x, respectively.

The rest of the paper first presents a high-level overview of our design and compares it to related proposals (Section 2). We present a cost model that accounts for communication overheads, making a case for an implementable hardware instruction steering design (Section 3). We then present details of the WiDGET microarchitecture (Section 4), and evaluate the power proportionality

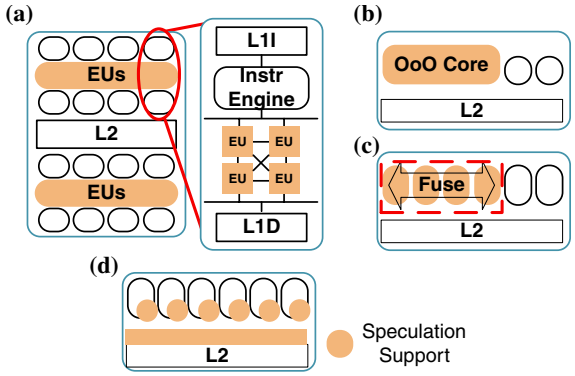


Figure 3. The general paradigm of balancing TLP and ILP
Each system uses the shaded components to accelerate single thread performance. (a)WiDGET. (b) Heterogeneous CMPs. (c) Dynamic multi-core processors. (d) TLS.

(Section 5). Finally, we present our conclusions and future work (Section 6).

2. OVERVIEW AND RELATED WORK

The goal of this work is a power proportional framework, in which cores gracefully transition across the power-performance spectrum. To address this goal, we harness multiple in-order issue resources, instead of relying on power-hungry OoO logic. This design delivers power efficiency, but three constraints must be met to preserve OoO-like performance. First, ready instructions must be exposed to the heads of the in-order issue buffers. Second, stalled instructions must not block the execution of later ready instructions. Third, the design should provide enough buffering capacity to prevent instruction queue clog—a pathologic stall condition in which dispatch is halted while all scheduling resources are occupied by earlier waiting instructions. Satisfying all of these constraints requires intelligent management of in-order issue resources.

Figure 3(a) illustrates WiDGET's sea of resources design. An instruction engine (IE) resembles a conventional OoO core's front-end and back-end pipeline functions with the addition of instruction steering logic for the distributed EUs. Each EU is capable of buffering and executing instructions in order. A hierarchical operand network connects a cluster of four adjacent EUs via full bypass, while a 1-cycle link bridges two adjacent clusters. An IE has an associated EU cluster, which is enough to deliver the performance of a comparable OoO machine (Section 5.2). Yet the decoupled design provides the flexibility to further scale up the core by borrowing up to four EUs from the neighboring IE. The hardware has control paths to distribute instructions and commands to any assigned EU in one cycle.

By varying the number of in-order EUs, which include some amount of instruction buffering, we can select a point on the power-performance spectrum best suited to the current situation. When the workload calls for aggressive exploitation of instruction-level parallelism (ILP), additional EUs can be allocated to service demands. On the other hand, the number of EUs can be reduced to conserve power, e.g., when running many threads.

2.1 Related Work

Table 1 summarizes the key aspects of WiDGET in comparison to the relevant prior proposals. Notably, WiDGET stands out by focusing on scaling both up and down (column 2) to achieve power proportionality (cols. 3-7) while retaining ISA compatibility (col. 8). We are not aware of any earlier work that demonstrates a core that can scale power and performance from close to an Atom to better than a Xeon. We briefly discuss closely related work, referencing Table 1.

2.1.1 Power-Proportional Computing

Chandrakasan et al. are among the first to introduce the concept of power-proportional computing [7]. They pointed out that once computational capability of a design meets service-level agreements, the remaining transistor budget should be devoted to power saving techniques, including DVFS [22] and power gating [14]. Barroso and Hölzle made a case for energy proportionality, especially for servers that rarely reach complete idle or near-peak utilization [4]. Whereas PowerNap proposes system-level techniques for idle-power reduction [25], Thread Motion employs several statically set voltage/frequency domains for non-idle power management [27]. Lastly, previously proposed adaptive cores focus on power savings [1,10,8], and thereby are limited to scaling cores down (col. 2).

2.1.2 Single-Thread Performance Techniques

Heterogeneous CMPs. A heterogeneous CMP (Figure 3(b)) combines a small number of aggressive superscalar cores for ILP with many lightweight cores for thread-level parallelism (TLP) (col. 3) [21]. Due to the statically set core designs for the target class of applications (col. 2), it has limited effectiveness for applications outside of the target class [31].

Dynamic Multi-Core Processors. Dynamic multi-core processors execute independent threads on a collection of small homogeneous cores (col. 3) to provide TLP, but dynamically fuse these cores together to provide greater ILP for a single thread (Figure 3(c)). Core Fusion enlarges the resources in each pipeline stage (col. 4) by aggregating OoO cores (col. 5) [16]. In contrast, Composable Lightweight Processors (CLP) leverages the EDGE ISA (col. 8) to eliminate centralized structures, such as rename logic, even in a fused mode [18], but requires recompilation, unlike WiDGET.

Thread-Level Speculation (TLS). TLS relies on software to divide a dynamic instruction stream into contiguous segments at control-flow boundaries (col. 7) [11,36]. The hardware speculatively executes the resulting chain of control dependent threads, using buffered state to recover from misspeculation (Figure 3(d)). The control driven execution makes TLS susceptible to thread squash propagation [29], which WiDGET avoids by being dataflow driven.

Clustered Architectures. The goal of early clustered architectures was to continue the superscalar trend of wider and deeper pipelines with minimum complexity for performance. Hence, each cluster may utilize complex OoO execution (col. 5). In addition, they trade off inter-cluster communication latencies for load balancing so long as the latencies can be hidden (cols. 6, 7). WiDGET, on the other hand, attacks both performance and power aspects of communication, as the next section explains, by using small units immune to long wire delays and localizing operand transfers.

Table 1: Comparison to Prior Related Work

Design	Scale Up & Down?	Symmetric?	Decoupled Exec?	In-Order?	Wire Delays?	Data Driven?	ISA Compativility?
WiDGET	Y	Y	Y	Y	Y	Y	Y
Adaptive Cores [1,10,8]	N	-	Y/N	Y/N	-	-	Y
Heterogeneous CMPs [21]	N	N	N	Y/N	-	-	Y
Core Fusion [16]	Y	Y	N	N	Y	-	Y
CLP [18]	Y	Y	Y	Y	Y	Y	N
TLS [13]	N	Y	N	Y/N	-	N	Y
Multiscalar [42]	N	Y	N	N	-	N	N
Complexity-Effective [26]	N	Y	Y	Y	N	Y	Y
Salverda & Zilles [31]	Y	Y	N	Y	N	Y	Y
ILDP [19] & Braid [38]	N	Y	Y	Y	-	Y	N
Quad-Cluster [3]	N	Y	Y	N	Y	Y/N	Y
Access/Execute [35]	N	N	N	Y	-	Y	N
Cost-Effective [6]	N	N	Y	N	Y	Y	Y

3. INSTRUCTION STEERING COST MODEL

Limits to CMOS scaling and increasing wire delays prompt a decentralized design, such as WiDGET. In this design regime, the instruction steering policy becomes an integral part of the performance equation. It determines issue time, which is governed by data dependencies, structural hazards, and the EUs' issue policy. As a result of this complex interaction, providing a cost model for instruction steering guides some of the design decisions for our architecture. We extend Salverda and Zilles's cost model by adding communication-latency-awareness [31] and discuss the implications of this new model.

3.1 Extending the Salverda and Zilles Model

Salverda and Zilles evaluated steering cost of an instruction *i* as a function of the dataflow (i.e., *horizon*) and in-order issue constraints (i.e., *frontier*) [31]. The horizon marks the time when an instruction becomes ready to issue, which is imposed by the dispatch time, *disp(i)*, and computation of the source operands, *data(i)*. Hence, the horizon of *i* is $h(i) = max\{disp(i), data(i)\}$. Throughout this section, we use Figure 4 to help explain their cost model and our extension. Figure 4(a) shows an example sequence of instructions, each of which takes one cycle to execute on one of the two available EUs. Both `i2` and `i3` depend on `i1` and must execute before `i4`. Assuming all three instructions dispatch in cycle 0, then the horizon of `i3` is 2 because it must wait for the result of `i1` to become available, as the arrow and

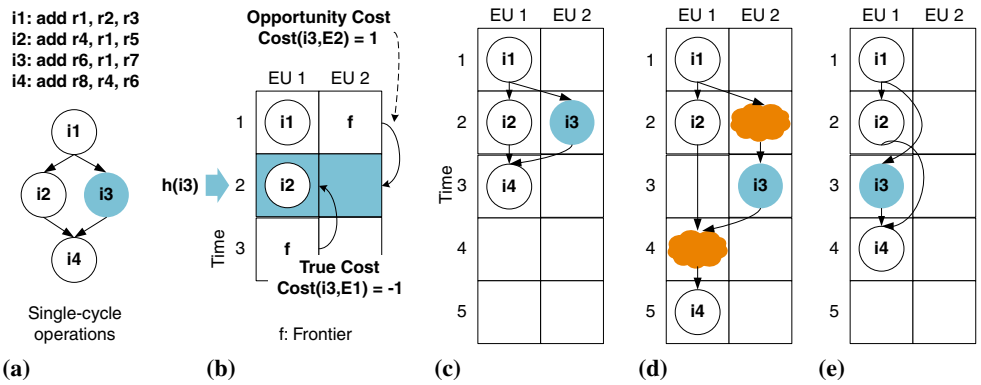
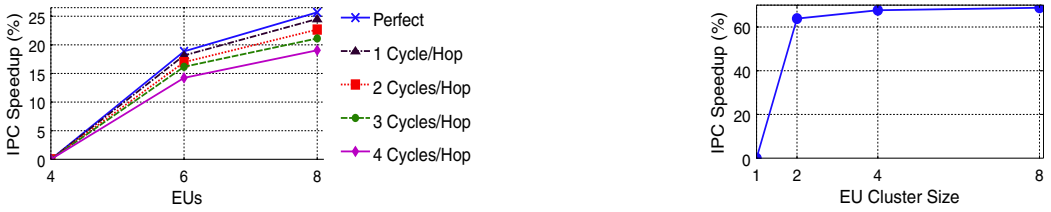


Figure 4. Limitations of the Salverda and Zilles cost model (a) An example instruction sequence and the dataflow graph. (b) Steering cost of i3. (c) Steering under idealized communication assumption. (d) Impacts of adding a 1-cycle latency between EUs using the ideal cost model. (e) Communication-latency-aware cost model under a 1-cycle latency between EUs.



(a) Unclustered EUs: EU count impact on performance

(b) Clustered EUs: Cluster size impact on performance

Figure 5. Performance sensitivity under realistic communication delays

shaded region in Figure 4(b) show. On the other hand, the frontier of an in-order EU *e*, *f(e)*, denotes the earliest time an instruction becomes the head of the FIFO queue. In Figure 4(b), the frontier of EU 1 is 3, whereas that of EU 2 is 1 due to the unutilized resource.

The cost of steering an instruction to an EU becomes: *Cost(i, e) = h(i) - f(e)*. A negative cost indicates a *true cost* of the instruction because earlier instructions in the steered EU delay the issue time. This is the case of steering `i3` to EU 1. Although `i3` becomes ready at cycle 2, it cannot issue until `i2` finishes execution at cycle 3. A positive cost, on the other hand, reflects an *opportunity cost*. The instruction becomes the earliest instruction in the EU while still waiting for the operands, potentially deferring execution of later instructions. Steering `i3` to EU 2 incurs an opportunity cost by leaving EU 2 idle at cycle 1. Thus, an ideal steering occurs when *Cost(i, e)* is zero, issuing the instruction as soon as it becomes ready without lowering EU utilization. This example has no zero-cost steering. The Salverda and Zilles cost model prefers minimal true cost to opportunity cost in order to increase parallelism, steering `i3` to EU 2. `i4` can be steered to either EU 1 or EU 2, but in either case completes in cycle three as illustrated in Figure 4(c).

Although Salverda and Zilles assumed idealized inter-EU communication, the communication cost cannot be ignored as wire delay increasingly dominates with decreasing CMOS feature scaling. If just a single-cycle inter-EU delay is added, Figure 4(d) shows that the same sequence now takes five cycles. The clouds depict the incurred operand transfer delays that did not exist under the idealized communication assumption in Figure 4(c). A cost model that is sensitive to communication overheads will instead keep all four instructions in the same EU, completing the sequence in four cycles as shown in Figure 4(e).

Figure 5 demonstrates the importance of accounting for communication delays. Figure 5(a) shows the harmonic means of IPC speedup for the SPEC CPU2006 benchmark suite by varying the communication delay from zero to four cycles. Each speedup is based on a four-EU configuration with the same delay. The idealized communication enables 26% speedup from four to eight EUs, whereas it drops to 19% under four-cycle delays. Thus, as one would expect, performance gains from increased EU count degrade as communication becomes more expensive.

However, assuming delays between every EU is rather pessimistic. A more realistic design will cluster a few EUs with no intra-cluster delay, while imposing inter-cluster delays. Figure 5(b) plots the performance implications of cluster size. It fixes the EU count to eight and assumes a 1-cycle delay per inter-cluster hop. The speedups are normalized to an unclustered design, in which inter-EU communication increases one cycle. By assigning two EUs per cluster, performance increases 64%. A cluster size of four further improves the speedup by another 4%, but the speedup gain becomes negligible beyond that point. Despite the similar performance of the 2- and 4-EU clusters, WiDGET employs the latter for more scalable power proportionality (Section 5.3).

Our extension of the Salverda and Zilles cost model incorporates the impact of communication delays. Specifically, operand availability is now governed by two variables:

operand computation time by the producer *and* the operand transfer time to reach the consumer EU. We denote the latter as *comm(i, e)*. The horizon is therefore a function of an EU as well: *h(i, e) = max{disp(i), data(i) + comm(i, e)}*. In the example of Figure 4, the horizon of `i3` is calculated as the following, provided it is dispatched at time 0 and `i1` is steered to EU 1:

h(i3, EU 1) = max{0, 2 + 0} = 2
h(i3, EU 2) = max{0, 2 + 1} = 3

We call the extended model the Communication-Latency-Aware Cost Model and measure the steering cost: *Cost(i, e) = h(i, e) - f(e)*. An ideal steering decision, therefore, sends an instruction to a different EU from the producer's EU only when the operand transfer latency can be hidden. In contrast, the Salverda and Zilles cost model, assuming no communication penalties, spreads computation across the EUs to minimize true cost, benefiting from higher EU count. Salverda and Zilles therefore conclude that dataflow properties constrain the performance improvement from fusing in-order cores. It requires either a very convoluted steering mechanism that keeps track of each EU's frontier in relation to an instruction's horizon or fusing so many cores that fusion overheads become impractical. Under realistic communication delays, however, our model tends to mitigate the pressure for more EUs and obviate the need for considering distant EUs. This reduces the number of available instruction steering slots, leading to an implementable steering policy.

3.2 Toward Practical Steering

WiDGET approximates the communication-latency-aware cost model by controlling often known variables, *disp(i)* and *comm(i, e)*, and simplifying hard-to-predict variables, *data(i)* and *f(e)*. We try to steer a consumer directly behind the producer, similar to the dependence-based steering proposed by Palacharla et al. [26]. The important difference is accounting for communication, thereby keeping dependent instructions nearby to reduce the latency and power from operand transfers. Hence, WiDGET only considers a subset of the available EUs for a given instruction, making the steering complexity tractable.

4. MICROARCHITECTURE

The current technological trends favor a hardware design based on a sea of resources. This design naturally maps well to TLP, but makes achieving high ILP very challenging. WiDGET addresses this issue by aggregating in-order-issue EUs to approximate OoO-issue capability. We therefore employ steering to distribute instructions, localizing dependent instructions into the same cluster whenever possible. The routing network forwards operands to intra-cluster EUs in time for back-to-back execution, but incurs an additional cycle for each inter-cluster transfer. Conversely, independent instructions are steered to any empty EUs and execute in parallel. When there is a long-latency instruction, the EU that is executing the instruction acts as a buffer for the chain of dependent instructions; other EUs remain in an unblocked state and can continue execution. Thus, our sea of resources design enables independent instructions to run

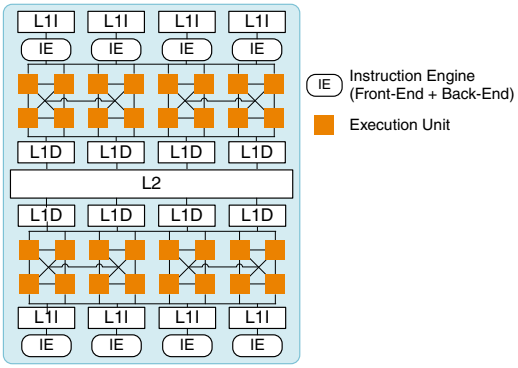


Figure 6. WiDGET microarchitecture

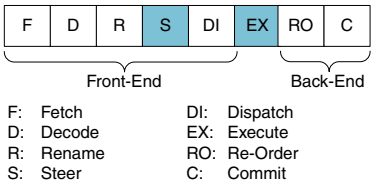


Figure 7. Pipeline Stages

ahead of the earlier stalled instructions in available EUs, extracting ILP and memory-level parallelism from a program. Figure 6 illustrates an example WiDGET chip with eight instruction engines (IEs), each of which consists of front-end and back-end pipeline functions comparable to a conventional OoO core. An IE therefore manages thread specific information, including the register file and the reorder buffer (ROB), for a thread fetched and dispatched from the IE. The following sections provide more details about the IE and EU functionality.

4.1 Pipeline Stages

Figure 7 shows WiDGET's pipeline stages, highlighting those that are unique to WiDGET. The non-shaded stages resemble a conventional OoO design except for the additional NoSQ (short for No Store Queue) support to eliminate a centralized memory disambiguation mechanism during execution [33].

WiDGET makes steering decisions at the Steer Stage so that instructions are dispatched to the appropriate EUs the following cycle. Section 4.2 provides detailed description of our steering heuristic.

The Execute stage can take multiple cycles depending on the operation and the utilization of the selected EU. Each EU independently manages instruction execution and no more than one operation issues at a time per EU. The total issue width is a function of the aggregate EU count, as each EU provides an additional execution engine. Executed instructions are removed from their EUs and forward the results directly to the consumer EUs, if any, and to the register file in the dispatching IE. Section 4.3 describes the detailed implementation.

4.2 Front-End

Figure 8 illustrates the detailed front-end of our architecture, which resembles a conventional OoO core's with the

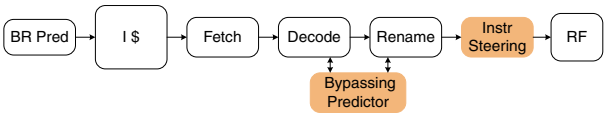


Figure 8. Front-End

```
/* Let I be an instruction under
consideration. Also, let s and EUps be I's
source operand and the producer EU of operand
s, respectively. s is omitted when only a
single operand is outstanding. */

0: switch (numOutstandingOps(I)){
1: case 0:
2:   return getEmptyEU();
3: case 1:
4:   if (!hasInstrBehind(s))
5:     return EUp;
6:   else
7:     return getEmptyEUInCluster(EUp);
8: case 2:
9:   if (!hasInstrBehind(s1))
10:    return EUp1;
11:  else if (!hasInstrBehind(s2))
12:    return EUp2;
13:  else
14:    return getEmptyEUInCluster(EUp1,EUp2);
15:}
```

Figure 9. Pseudo-code for instruction steering

addition of the NoSQ mechanism (*Bypassing Predictor*) and instruction steering. We derive the steering heuristic from the observation made in Section 3 that dependent instructions must be kept nearby, obviating the need for considering every EU each time. Specifically, we send consumers directly behind the producer or to an empty EU in the same EU cluster. If no such EU is found, we simply stall steering until either a desirable EU becomes available or the producer finishes execution. It is through stalling that we ensure steering complexity is manageable and communication overheads do not diminish the benefits from parallelism.

The heuristic requires three pieces of information: a producer's steered EU, whether a producer has another consumer steered to the same instruction buffer, and a list of empty EUs. We employ a Last Producer Table (LPT) and an empty bit vector to keep track of the first two and the last information, respectively. The LPT is indexed by a register and contains two fields. The first field indicates the instruction buffer ID to which the producing instruction of the given register is steered. The second field consists of a single bit; when set, this bit indicates at least one instruction has been steered as a result of the producer-consumer relationship. An LPT entry is updated when an instruction is steered and is invalidated when the register value is written back to the register file. An invalid entry, therefore, indicates that the value has been computed and is available in the register file. The empty bit vector is sized to the total number of instruction buffers, marking the corresponding buffer's occupancy status. We similarly use a full bit vector to ensure a producer instruction buffer still has room for the consumer instruction. Feedback from the EUs updates both of the bit vectors every cycle.

Figure 9 provides pseudo-code for the steering heuristic. The location of a producer (*EUp*) is tracked by accessing LPT with the consumer's source operand. If the indexed entry's buffer ID is null, the producer has already computed the operand. *getEmptyEU()* accesses the empty bit vector and returns an ID whose corresponding entry is set to 1. If more than one entry is set, it randomly chooses an ID from an EU cluster with more empty buffers for load balancing. If all of the entries are set to 0, the function returns -1, indicating a steering stall. *hasInstrBehind(s)* returns true if an LPT entry indexed by *s* has the consumer field set to 1; otherwise, it returns false. Given one or more EU IDs, *getEmptyEUInCluster()* searches the empty bit vector only within the corresponding EU cluster(s). It returns either an available ID similarly to *getEmptyEU()* or -1 if no empty buffers are found in the cluster(s), stalling the steering.

This is best explained with an example, illustrated in Figure 10. Suppose eight EUs spanning two clusters are dedicated to this instruction engine. Further assume all operands are initially available in the register file and all EUs are empty. Figure 10(a) shows a dataflow graph of instructions with each node denoting an instruction sequence number and the destination register in parenthesis.

In the first cycle, instructions `i1` through `i4` are steered. Since `i1` has no data dependencies, it is steered to the empty EU 0 (line 2 in Figure 9). It marks the steered EU ID in the LPT entry for the destination register `r1`, leaving the consumer field unchanged. It also resets the empty bit vector for EU 0. Conversely, `i2` depends on `i1`. An access to the LPT entry for `r1` reveals the producer of `i1` is steered to EU 0 and no other instructions have followed `i1` yet. Hence, `i2` is steered to the producer EU 0 (line 5). It updates the corresponding LPT entry as well as `i1`'s to prevent other consumers of `i1` from steering to EU 0. `i3` begins a new independent chain. It selects the empty EU 4 in Cluster 1 to balance the load (line 2). Both the LPT entry for `r3` and the empty bit vector are updated accordingly. `i4` is analogous to the case of `i2`, following the producer EU 4 (line 5). As both of the head instructions in EUs 0 and 4 are ready, they execute in their EUs. Figure 10(b) shows the steering result at the end of cycle 1.

In the second cycle, `i5` through `i8` are steered. `i5` is sent to the producer EU 0 (line 5), setting the consumer field in the `r2`'s LPT entry. `i6` also depends on `i2`, yet an LPT lookup informs that the slot immediately succeeding `r2` has been claimed. `i6` therefore finds the empty EU 2 in the same Cluster 0 by accessing the Empty Bit Vector (line 7). Note that `i2` will forward the result to EU 2 at the end of the execution, enabling both `i5` and `i6` to execute in parallel. `i7` depends on both `i5` and `i4` which are in EUs 0 and 4, respectively. Although both of the producers have empty slots behind them, `i7` selects the producer EU 0 of the first source operand `r5` (line 10). Finally, `i8` is sent to the producer EU 2 (line 5). Figure 10(c) displays the final state at the end of cycle 2.

The naive implementation of steering is serial, since it is constrained by the serial dependencies among a group of instructions to be steered. However, we utilize a parallel prefix computation. The parallel dependence-check performed in renaming is employed to detect dependencies in the same steering group [30]. Concurrently, each instruction accesses the LPT and the bit vectors to choose a candidate EU. Then

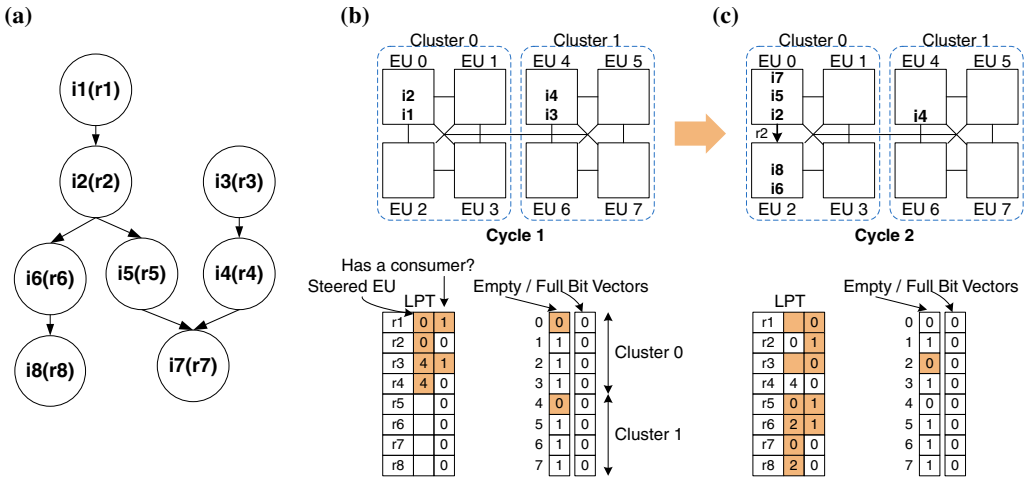


Figure 10. Instruction Steering Example

the candidate EUs are compared and are modified if necessary to reflect the intra-group dependencies, followed by updating the LPT and the bit vectors accordingly.

4.3 Execution Unit

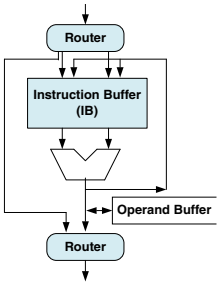


Figure 11. Execution Unit

Figure 11 shows an EU, consisting of a small instruction buffer (IB) FIFO, execution engine (an integer ALU, floating-point unit, and address generation unit), operand buffer, and router connections at the input and output. The IB is configured to be four times larger than the fetch width to minimize front-end stalls. Each entry has four fields: instruction, operand 1, operand 2, and a bit vector of consumer EU IDs. The consumer EU ID field indicates EUs to forward the result to. Although our dynamic instruction steering provides the ability to adjust to dynamic events, the caveat is that consumer EUs are not known until the dependent instructions are steered. Therefore, an instruction has to send its EU ID to the producer EU via control paths after steering is performed. However, a race can occur if a producer has finished execution and has been removed from the IB by the time the consumer reaches the producer EU. To prevent this situation, an operand buffer holds the result of an instruction for a cycle, which is the latency of register file write backs.

The units in the execution engine are pipelined, though an EU can only issue one instruction per cycle. Note that each additional EU increases both issue bandwidth and buffer space for scheduled instructions. This primarily contributes to WiDGET's high single thread performance with simple in-order EUs.

4.4 Back-End

The back-end resembles a conventional OoO core's. The ROB ensures in-order commit, and stores write their values to the data cache at commit as in a traditional pipeline.

5. EVALUATION

This section evaluates the microarchitecture described in Section 4 and the potential for power-proportional computing. Since this paper focuses on single thread performance, we leave the evaluation of parallelism management as future work. Instead, we focus on two crucial properties of power proportionality: wide performance and power ranges.

5.1 Simulation Methodology

We use a full-system execution-driven simulator, composed of three parts: 1) A detailed timing-first microarchitecture-level processor simulator, which models dynamically-scheduled SPARC v9 processors, 2) Wisconsin GEMS [24,39] for memory system timing, and 3) Virtutech Simics [23] for full-system functionality and to verify the timing-first components. Wattch [5] and CACTI [34] are integrated for modeling power.

For the single thread performance evaluation, we use the entire SPEC CPU2006 benchmark suite, which is a collection of sequential workloads. All programs were compiled for the 64-bit SPARC ISA using the Sun Studio 11 compiler with base tuning.

WiDGET's power proportionality can be best assessed by commercial processors on the opposite ends of a design spectrum. Hence, we use two baselines designed after a low power Intel Atom Silverthorne [9] and a high performance Intel Xeon Tulsa [37]. We call the former Mite and the latter Neon. The memory hierarchy of the baselines and WiDGET is configured to emulate Neon's in order to isolate the performance of the three different core designs. We evaluate WiDGET configurations with 1 through 8 EUs allocated to an instruction engine. These initial experiments assume *a priori* static allocation of EUs, as might be done by low-level system software. The mechanisms and policies to trade-off ILP and TLP via EU provisioning are beyond the scope of this paper.

Table 2 lists the key configuration parameters. The area estimate only accounts for a single-threaded core with the listed memory hierarchy. We derived the area of the Neon and Mite from published die area and attributed core com-

Table 2: Machine configurations

	Mite	Neon	WiDGET
LI-I / L1-D	32 KB, 4-way, 1 cycle; Next-line prefetching for L1-I		
BR Predictor	4K-entry Bimodal branch predictor; 16-entry RAS; 64-entry, 4-way BTB		
Instruction Engine	2-wide FE and BE	4-wide FE and BE; 128-entry ROB	
Execution Core	16-entry unified in-order instruction queue; 2 INT, 2 FP, 2 Addr Gen	32-entry unified OoO instruction queue; 3 INT, 3 FP, and 2 Addr Gen; 0-cycle operand bypass to anywhere in core	16-entry in-order IB per EU; 1 INT, 1 FP, and 1 Addr Gen per EU; 0-cycle operand bypass within a cluster of four EUs; 1-cycle inter-cluster link
Disambiguation	NoSQ; 256-entry, 4-way store-load bypassing predictor; 1K-entry T-SSBF		
L2 / L3 / DRAM	1 MB, 8-way, 12 cycles / 4 MB, 16-way, 24 cycles / ~300 cycles, 16-entry MSHR		
Area Estimate (45nm)	~30 mm^2	~41 mm^2	~33 mm^2 (8 EUs)

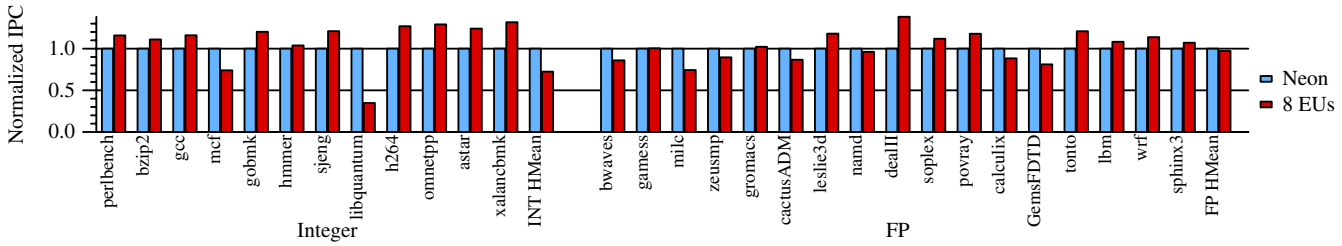


Figure 12. 8-EU performance relative to the Neon

ponent or unit area. The Neon's area was then halved because of the process technology change from 65 and 45nm. Note, our more aggressive memory hierarchy increases Mite's memory die area. WiDGET's area estimate includes an instruction engine, 8 EUs, and the memory hierarchy. We estimate that the Atom's core area is roughly equivalent to WiDGET with 2 EUs due to the similar core structure sizes. The area of an each additional EU is based on a TRIPS processor's Execution Tile [17], which resembles the EU composition. Despite WiDGET's greater ALU resources, our area model concludes that WiDGET is smaller than the Neon, mainly due to WiDGET's simpler structures.

In this evaluation, we fully provision ports on the register file and L1-D cache. When modeling power consumption, we outfit the register file with read ports numbering twice the dispatch width (8), and write ports numbering the commit width (4). A similar simplifying assumption is applied to the L1-D power model.

5.2 Performance Range of WiDGET

A wide performance range is vital for power proportionality, yet WiDGET's in-order issue constraint makes it challenging to match OoO execution performance. Therefore, we first evaluate single thread performance of WiDGET when configured with the maximum number of EUs: 8 EUs with an instruction buffer per EU. Figure 12 presents IPCs relative to the high-performance Neon baseline, with integer benchmarks on the left and floating-point benchmarks on the right. Even with more than double the ALU resources, one third of the benchmarks fail to match the Neon's performance. In particular, WiDGET is only able to produce 35% of the Neon performance for the outlier libquantum, drastically impacting the integer harmonic mean.

Figure 13 demonstrates the average EU utilization, revealing the sources of the performance degradation. *Empty* is when an EU has no instructions in the instruction buffer (IB). *Waiting for Producer* and *Waiting for Op Transfer* are when EU utilization is wasted because the head instruction in the IB has at least one outstanding operand. The former is waiting for the operand to be computed, whereas the latter indicates that the operand has been computed but has not yet reached the EU due to the inter-cluster communication delay. Finally, *Accessing Memory* and *Executing ALU* are when an EU is executing memory and non-memory instructions, respectively. Since instructions reside in IBs until execution is complete, memory-intensive workloads cause EUs to spend much of the time waiting for load data (*Accessing Memory*).

The under-performing benchmarks demonstrate common characteristics: frequent stalls due to memory access and outstanding producers. This increases the pressure on the EUs to buffer more dependent instructions for a longer period of time. As a result, the steering logic becomes more prone to stalls due to the lack of desirable EUs. libquantum is the most prominent example. It spends 62% and 37% of the time on memory accesses and waiting for producers, respectively, leaving non-memory execution to a mere 1%. In contrast, benchmarks that have comparable performance to the Neon have the opposite trends. They have a larger portion of time spent on executing non-memory instructions and are less likely to waste EU utilization by waiting for operands. Hence, fewer EUs are necessary to buffer stalled chains of instructions, leveraging more EUs to execute inde-

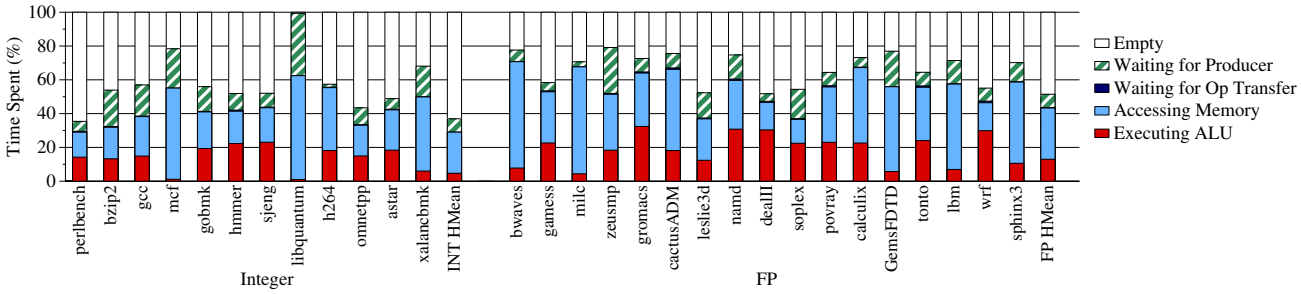
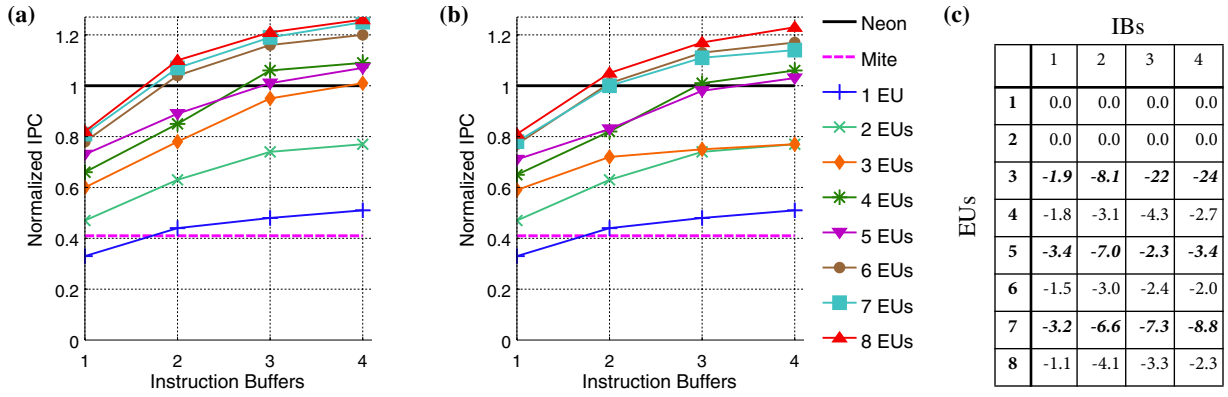


Figure 13. Average cycles spent on each EU state with 8 EUs



(c)

EUs \ IBs	1	2	3	4
1	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
3	***-1.9***	***-8.1***	***-22***	***-24***
4	-1.8	-3.1	-4.3	-2.7
5	***-3.4***	***-7.0***	***-2.3***	***-3.4***
6	-1.5	-3.0	-2.4	-2.0
7	***-3.2***	***-6.6***	***-7.3***	***-8.8***
8	-1.1	-4.1	-3.3	-2.3

Figure 14. Harmonic mean IPCs relative to the Neon (a) 4-EU cluster size. (b) 2-EU cluster size. (c) IPC degradation (%) of the 2-EU cluster size compared to the 4-EU cluster size.

pendent chains. Note that WiDGET's hierarchical operand network in lieu of the Neon's full operand bypass has little effect on the EUs. EUs spend less than 1% of the time on waiting for operands to be transferred, which is accomplished by enforcing cluster affinity at the steering logic.

As Salverda and Zilles observed [31], the limiting factor of WiDGET's performance is the number of independent instruction chains the system can expose, not the issue bandwidth. To overcome this, we expand the buffering capability by allocating multiple IBs to each EU. Despite the same issue bandwidth, an EU can now buffer more than one stalled chain while permitting an independent chain in another IB to utilize the otherwise idle execution engine. This change, however, requires each EU to have simple instruction issue selection logic; we use an oldest-instruction-first policy. Nevertheless, the logic is much less complex than that of a monolithic OoO as long as the number of IBs per EU is kept small. WiDGET's selection logic, with 8 EUs and 4 IBs each, only consumes 3% of the Neon's centralized instruction selection logic power. We also enlarge the size of the empty and full bit vectors in the steering logic to the total number of IBs. The steering complexity is managed by keeping the cluster locality invariant.

Figure 14(a) summarizes the performance benefits of increased buffering. The harmonic mean IPCs of the entire SPEC CPU2006 benchmark suite is presented, normalized to the Neon. WiDGET performs comparably to the Neon with at least 12 IBs, which is explained by the benchmarks' dataflow characteristics. When an ROB has 128 entries, which is our configuration, the integer and floating-point benchmarks have 8 and 12 extractable independent chains, respectively (these results not shown). With 4 IBs per EU, as few as 3 EUs are sufficient, while 8 EUs outperform the Neon by 26%—a sharp contrast to 18% degradation by the 1 IB counterpart. Therefore, mapping chains of dependent instructions to the sufficient number of buffers achieves extraction of ILP and memory-level parallelism despite the in-order issue constraints. WiDGET can also scale down with a single EU and an IB with performance slightly less than the Mite, offering a wide performance range of 3.8.

It is clear that the performance sees diminishing returns after 7 EUs with 4 IBs each, obviating the need for more than 2 clusters. Rather, an interesting comparison is to 8 EUs with 3 IBs, which yield similar performance. 7 EUs with 4 IBs deploy more buffers than 8 EUs with 3 IBs, while the latter uses more ALUs. We evaluate the tradeoff from the power dissipation perspective in Section 5.4.

5.3 Impacts of a Cluster Size

Cluster sizes impact WiDGET performance, making it non-monotonic with increasing EU count. Figure 14(a) illustrates this anomaly for the 5-EU case with three or four IBs, which is caused by the hierarchical operand network that employs a single-cycle link to bridge two adjacent clusters of 4 fully bypassed EUs. Hence, the 5-EU design forms a highly unbalanced system of a cluster of 4 EUs and a single EU. As dependent instructions are steered to the same cluster for locality, instruction chains steered to the single-EU cluster are likely to stall the front-end due to a lack of buffers. Increased buffer count mitigates the stalls, though it also creates opportunities for other independent chains to be steered to the single-EU cluster, causing structural hazards.

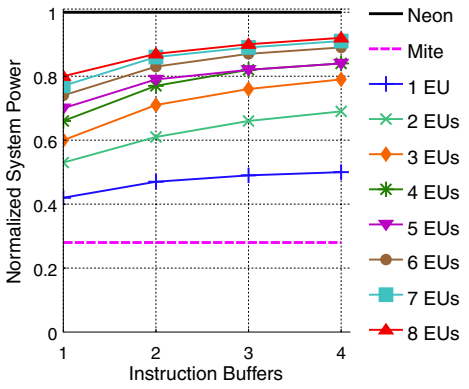


Figure 15. Harmonic mean system power relative to the Neon

mcf, which has a complex dataflow, is primarily responsible for degrading the harmonic mean IPC.

Reducing the cluster size to 2 has more dramatic performance impacts. Even though it can simplify steering logic and intra-cluster full-bypass network, Figure 14(b) demonstrates that the performance becomes highly sensitive to the cluster formation. With 3 or more IBs per EU, the odd EU configurations degrade the performance of the even EU configurations once 2 EUs are assigned. Therefore, under this cluster size, WiDGET must allocate a pair of EUs to gain benefits, resulting in coarser-grained power proportionality. We thus use a cluster size of 4 for the rest of the paper.

5.4 Power Range of WiDGET

Figure 15 presents the harmonic mean system power of the SPEC CPU2006 benchmark suite, normalized to the Neon. We did a best-effort validation of the Neon and Mite power consumption against Atom [9] and Xeon [37] processors, respectively, by first configuring them using the published data. We power down non-provisioned EUs [14].

The shape of WiDGET's curve resembles that of the performance curve in Figure 14, demonstrating the power proportionality. WiDGET, composed of simple building blocks, achieves 8-58% power savings compared to the Neon. Furthermore, WiDGET's EU modularity enables scaling down the power by up to 2.2 to approximate the Mite's low power. The crossing of the 4- and 5-EU lines, unlike the rest of the monotonic power increase with EU count, is due to the relationship of the hierarchical operand network and the steering heuristic as before.

Figure 15 elucidates the previous section's performance and EU provisioning tradeoff. Since power consumption of 7 EUs with 4 IBs and 8 EUs with 3 IBs are almost identical, one can resort to the former, the slightly higher performing configuration. This has additional benefits of stealing less EUs from a neighbor, allowing more threads to run in parallel as the power budget permits.

WiDGET's power increase from dedicating more EUs is not solely due to the additional resources, but is also the result of higher utilization in the existing resources. As Figure 16 shows, the breakdown for harmonic mean system power of the SPEC CPU2006 benchmark suite is divided into two broad categories, each with four subcategories: caches and core logic. *Fetch/Decode/Rename*, which includes a branch predictor and an instruction translation buffer, accounts for most of the front-end logic. WiDGET's instruction steering logic, despite residing in the front-end, and the execution core are included in the *Execution* component to make a fair comparison with Neon and Mite for power stemming from the execution model. Hence, this category encompasses in-order instruction queues for the Mite and WiDGET, an OoO instruction queue for the Neon, operand network, and a data translation lookaside buffer. *Backend* and *ALU* include the commit logic and the ALU resources, respectively.

Enlarging resource allocation has a first-order impact on the ALU and execution power, whereas a second-order impact is increased activity in the system caused by the larger effective window size, resulting in proportional power growth in Fetch/Decode/Rename, L1D, and L2. Yet, WiDGET's considerable power savings compared to Neon comes from the difference in the execution models. WiDGET effectively replaces the Neon's associative search in the OoO issue queue and the full bypass with simple in-order EUs and the hierarchical operand network, resulting in 24-29% reduction in the execution power. This is sufficient to mask out WiDGET's additional power resulting from the higher ALU count, even in the 8-EU case.

The breakdown is also useful to understand the power gap between Mite and WiDGET's 1 EU with 1 IB configuration. WiDGET's OoO support in the instruction engine, namely register renaming and ROB, is primarily responsible for the extra power. Nonetheless, WiDGET achieves a wide power range of 2.2.

Figure 17 puts together the power-performance relationship of the three designs. It is the same graph as Figure 2, except with differentiation of WiDGET's EU count. WiDGET consumes 21% less power than the Neon for the same performance (i.e., 3 EUs with 4 IBs), and yields 8% power savings for 26% better performance (i.e., 8 EUs with 4 IBs). WiDGET dissipates power in proportion to the performance, covering both the high-performance Neon and the low-power Mite on a single chip.

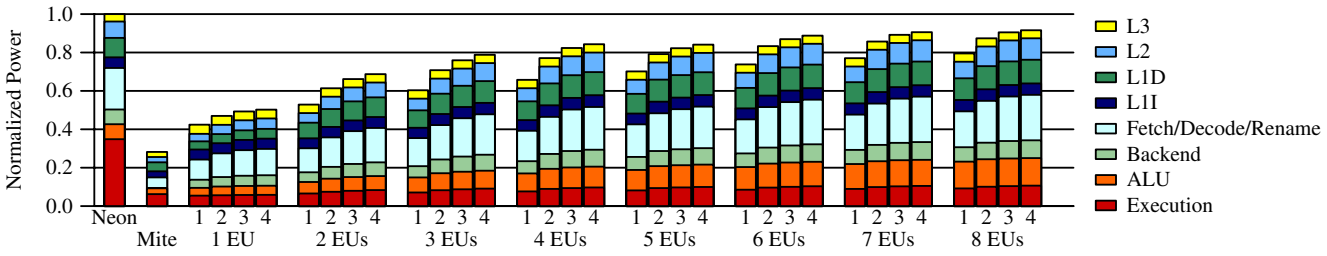


Figure 16. Power breakdown relative to the Neon

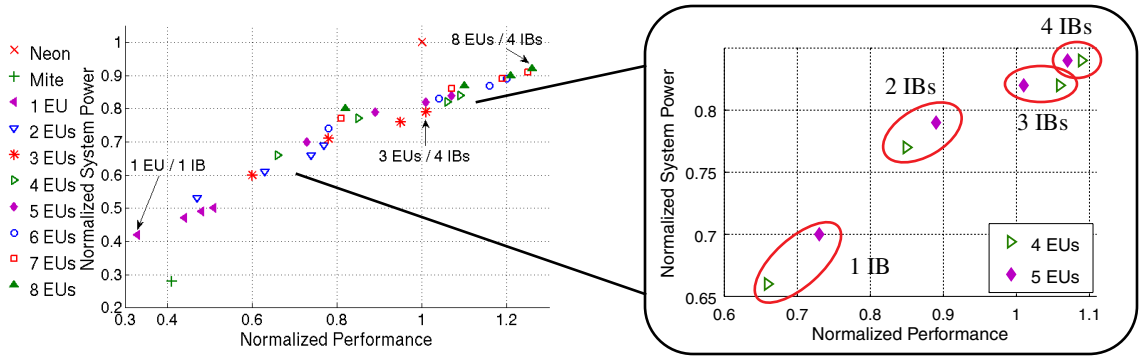


Figure 17. Power Proportionality of WiDGET compared to Neon and Mite

For a more detailed look at Figure 17, we provide a blown-up cutout focusing on the 4- and 5-EU data points that do not follow the rest of the power-performance trend due to the unbalanced clusters. The data points of each EU count correspond to 1 through 4 IBs from left to right. The distance between the 4- and 5-EU points shrinks as the number of IBs per EU increases from 1 to 2. With 3 or 4 IBs, the positions of the 4- and 5-EU points are swapped; the 5-EU points are to the left and lower than their 4-EU counterparts. Hence, it is not beneficial to increase buffers after allocating 2 IBs to the 5-EU configuration.

Finally, Figure 18 compares power efficiency of the three processor designs. We use BIPS^3/W as the metric, which is appropriate for evaluating efficiency differentials caused by microarchitectural designs [12]. Notably, the diagonal line in the figure demarks the seven of WiDGET's configurations below the line that deliver higher power efficiency than the Neon despite the lower performance: 6-8 EUs with 1 IB, 3-5 EUs with 2 IBs, and 3 EUs with 3 IBs. Furthermore, WiDGET is 48% more power efficient than the Neon when achieving the same performance and exceeds the Neon and Mite by up to 2x and 21x, respectively.

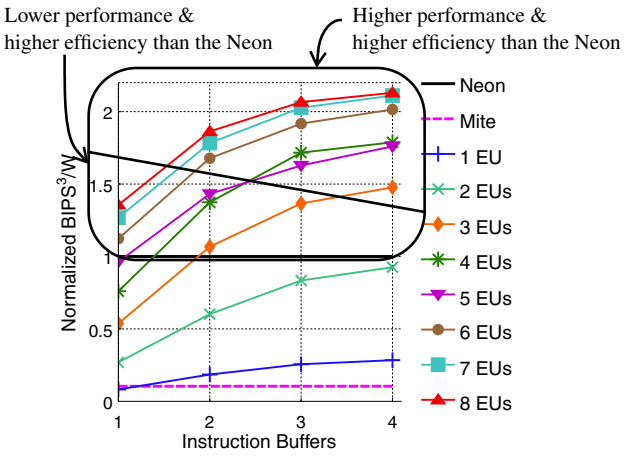


Figure 18. Geometric mean power efficiency (BIPS^3/W)

6. CONCLUSION

We proposed a power proportional design called WiDGET. The decoupling of the computation resources enables flexibility to provision EUs to meet different power-performance goals. WiDGET can be optimized for high throughput and low power by provisioning a small number of EUs to each instruction engine. When one or more powerful cores are needed to meet service-level agreements, for instance, system software can dedicate more EUs to accelerate single thread performance. By using only as many resources as necessary to deliver target performance, WiDGET achieves power-proportional computing.

Despite the use of in-order EUs to save power, WiDGET yields even higher performance than the aggressive Neon by deploying sufficient buffering for scheduled instructions and steering based on dependences. This distributed instruction buffering was the key to single thread performance and we believe global distributed buffering, if managed well, can yield performance for other forms of parallelism. The distributed instruction buffers improve latency tolerance, allowing independent chains to execute ahead of earlier, stalled chains, which is our mechanism to extract ILP and memory level parallelism. Furthermore, the removal of OoO execution logic contributes to the primary power savings of WiDGET, resulting in 24-29% reduction in the execution power compared to Neon. We experimentally showed that WiDGET consumes 21% less power than the Neon for the same performance and achieves 8% power savings for 26% better performance than Neon.

This paper focused on the single thread capability of WiDGET. In the future, we will evaluate it in a multi-threaded context, generalizing power proportionality to energy proportionality. Many open questions still remain, including policies for dynamic EU provisioning both within and across instruction engines, guaranteeing service-level agreements in the hardware, and harnessing other shared resources, such as the L2 and off-chip bandwidth. We would also like to explore multithreading the instruction engines for further power savings. A case where this is valuable is multithreaded applications with high cache miss rates. Since instruction engines would spend most of the time idle waiting for the memory requests, it is more energy efficient to consolidate such threads onto a single instruction engine and turn off other instruction engines. In this case, the distributed instruction buffers within an EU could also be used for thread instruction buffers, providing another level of multithreading beyond what could be available in the IE. EUs may also be specialized to better accommodate different

types and phases of applications. An example is a deployment of a few EUs tuned for streaming memory accesses and a computation-intensive EU.

WiDGET is the first step towards a grand goal of a unified framework for extracting parallelism with high energy efficiency. Specifically, we would like to achieve an energy proportional design that reconfigures the cores to meet a power-performance target of a given workload mix.

7. ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation (NSF), with grants CCR-0324878, CNS-0551401, CNS-0720565, and CCF-0916725, as well as donations from Microsoft and Sun Microsystems/Oracle. The views expressed herein are not necessarily those of the NSF, Microsoft or Sun Microsystems/Oracle. Prof. Wood has a significant financial interest in Microsoft.

We thank Dan Gibson, Steven Kunkel, James Laudon, Charles P. Thacker, Philip M. Wells, and anonymous reviewers for their comments on the paper.

8. REFERENCES

[1] D. Albonesi, R., Balasubramonian, S. Dropsbo, S. Dwarkadas, F. Friedman, M. Huang, V. Kursun, G. Magklis, M. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. Cook, and S. Schuster. Dynamically tuning processor resources with adaptive processing. *IEEE Computer*, 36(2):49–58, Dec. 2003.

[2] G. M. Amdahl. Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Conference Proceedings*, pages 483–485, Apr. 1967.

[3] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *Proc. of the 27th Annual Intnl. Symp. on Computer Architecture*, June 2000.

[4] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12), 2007.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proc. of the 27th Annual Intnl. Symp. on Computer Architecture*, pages 83–94, June 2000.

[6] R. Canal, J.-M. Parcerisa, and A. Gonzalez. A Cost-Effective Clustered Architecture. In *Proc. of the Intnl. Conf. on Parallel Architectures and Compilation Techniques*, Oct. 1999.

[7] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-Power CMOS Digital Design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992.

[8] M. S. Floyd, S. Ghiasi, T. W. Keller, K. Rajamani, F. L. Rawson, J. C. Rubio, and M. S. Ware. System power management support in the IBM POWER6 microprocessor. *IBM Journal of Research and Development*, 51(6), 2007.

[9] G. Gerosa, S. Curtis, M. D'Addeo, B. Jiang, B. Kuttanna, F. Merchant, B. Patel, M. Taufique, and H. Samarchi. A Sub-2 W Low Power IA Processor for Mobile Internet Devices in 45 nm High-k Metal Gate CMOS. *IEEE Journal of Solid-State Circuits*, 44(1):73–82, 2009.

[10] J. González and A. González. Dynamic Cluster Resizing. In *Proceedings of the 21st International Conference on Computer Design*, 2003.

[11] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, March-April 2000.

[12] A. Hartstein and T. R. Puzak. Optimum Power/Performance Pipeline Depth. In *Proc. of the 36th Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2003.

[13] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, pages 33–38, July 2008.

[14] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *International Symposium on Low Power Electronics and Design*, pages 32–37, Aug. 2004.

[15] Intel and Core i7 (Nehalem) Dynamic Power Management, 2008.

[16] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core Fusion: Accomodating Software Diversity in Chip Multiprocessors. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[17] S. Keckler, D. Burger, K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. Govindan, P. Gratz, D. Gulati, H. H. amd C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, and P. Shivakumar. *Architecture and Implementation of the TRIPS Processor*. CRC Press, 2007.

[18] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable Lightweight Processors. In *Proc. of the 40th Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2007.

[19] H. S. Kim and J. E. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *Proc. of the 29th Annual Intnl. Symp. on Computer Architecture*, May 2002.

[20] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, Mar/Apr 2005.

[21] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. In *Proc. of the 31st Annual Intnl. Symp. on Computer Architecture*, pages 64–75, June 2004.

[22] G. Magklis, G. Semeraro, D. H. Albonesi, S. G. Dropsho, S. Dwarkadas, and M. L. Scott. Dynamic Frequency and Voltage Scaling for a Multiple-Clock-Domain Microprocessor. *IEEE Micro*, 23(6):62–68, Nov/Dec 2003.

[23] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[24] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, pages 92–99, Sept. 2005.

[25] D. Meisner, B. T. Gold, and T. F. Wenisch. PowerNap: Eliminating Server Idle Power. In *Proc. of the 14th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar. 2009.

[26] S. Palacharla and J. E. Smith. Complexity-Effective Superscalar Processors. In *Proc. of the 24th Annual Intnl. Symp. on Computer Architecture*, pages 206–218, June 1997.

[27] K. K. Rangan, G.-Y. Wei, and D. Brooks. Thread Motion: Fine-Grained Power Management for Multi-Core Systems. In *Proc. of the 36th Annual Intnl. Symp. on Computer Architecture*, June 2009.

[28] V. J. Reddi, B. Lee, T. Chilimbi, and K. Vaid. Web Search Using Small Cores: Quantifying the Price of Efficiency. *Technical Report MSR-TR-2009-105, Microsoft Research*, Aug. 2009.

[29] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, and J. Torrellas. Energy-Efficient Thread-Level Speculation on a CMP. *IEEE Micro*, 26(1), Jan/Feb 2006.

[30] A. Roth and G. S. Sohi. Register Integration: A Simple and Efficient Implementation of Squash Reuse. In *Proc. of the 33rd Annual IEEE/ACM International Symp. on Microarchitecture*, pages 223–234, Dec. 2000.

[31] P. Salverda and C. Zilles. Fundamental performance constraints in horizontal fusion of in-order cores. In *Proc. of the 14th IEEE Symp. on High-Performance Computer Architecture*, pages 252–263, Feb. 2008.

[32] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *Proceedings of the International Conference on Computer Graphics and Interactive Techniques*, 2008.

[33] T. Sha, M. M. K. Martin, and A. Roth. NoSQ: Store-Load Communication without a Store Queue. In *Proc. of the 39th Annual IEEE/ACM International Symp. on Microarchitecture*, pages 285–296, Dec. 2006.

[34] T. Shyamkumar, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, Hewlett Packard Labs, 2008.

[35] J. E. Smith. Decoupled Access/Execute Computer Architecture. In *Proc. of the 9th Annual Symp. on Computer Architecture*, pages 112–119, Apr. 1982.

[36] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *Proc. of the 22nd Annual Intnl. Symp. on Computer Architecture*, pages 414–425, June 1995.

[37] S. Tam, S. Rusu, J. Chang, S. Vora, B. Cherkauer, and D. Ayers. A 65nm 95W Dual-Core Multi-Threaded Xeon Processor with L3 Cache. In *Proc. of the 2006 IEEE Asian Solid-State Circuits Conference*, Nov. 2006.

[38] F. Tseng and Y. N. Patt. Achieving Out-of-Order Performance with Almost In-Order Complexity. In *Proc. of the 35th Annual Intnl. Symp. on Computer Architecture*, June 2008.

[39] Wisconsin Multifacet GEMS Simulator. http://www.cs.wisc.edu/gems/.

[40] B. Zhai, D. Blaauw, D. Sylvester, and K. Flaunter. Theoretical and Practical Limits of Dynamic Voltage Scaling. In *Proc. of the 41st Annual Design Automation Conference*, pages 868–873, June 2004.