Preprint – To appear in Proc. ACM/IEEE Int. Symp. on Computer Architecture (ISCA), Minneapolis, MN, USA, June 2014

Improving the Energy Efficiency of Big Cores

Kenneth Czechowski[*] Victor W. Lee[†] Ed Grochowski[†] Ronny Ronen[†]
Ronak Singhal[†] Richard Vuduc[*] Pradeep Dubey[†]
[*]Georgia Institute of Technology, Atlanta, GA
[†]Intel Corporation, Santa Clara, CA
kentcz@gatech.edu, victor.w.lee@intel.com

Abstract

Traditionally, architectural innovations designed to boost single-threaded performance incur overhead costs which significantly increase power consumption. In many cases the increase in power exceeds the improvement in performance, resulting in a net increase in energy consumption. Thus, it is reasonable to assume that modern attempts to improve single-threaded performance will have a negative impact on energy efficiency. This has led to the belief that "Big Cores" are inherently inefficient. To the contrary, we present a study which finds that the increased complexity of the core microarchitecture in recent generations of the Intel® Core™ processor have reduced both the time and energy required to run various workloads. Moreover, taking out the impact of process technology changes, our study still finds the architecture and microarchitecture changes —such as the increase in SIMD width, addition of the frontend caches, and the enhancement to the out-of-order execution engine— account for 1.2x improvement in energy efficiency for these processors. This paper provides real-world examples of how architectural innovations can mitigate inefficiencies associated with "Big Cores" —for example, micro-op caches obviate the costly decode of complex x86 instructions— resulting in a core architecture that is both high performance and energy efficient. It also contributes to the understanding of how microarchitecture affects performance, power and energy efficiency by modeling the relationship between them.

1. Introduction

Power and Energy have always had a significant impact on processor design. Yet, as the demand for energy efficient computing increases and rate of improvement from process technology slows, energy efficiency has become a first-class design constraint. Prior work has shown that increasing single-threaded performance through microarchitectural advances tends to increase power [13, 2]. This increase is due to the increased complexity of the core such as a larger instruction window and more aggressive use of speculation. However, this increase can be countered by reductions in energy due to improvements in process technology, architecture, and microarchitecture. This paper examines the improvements in each of these areas. We use measurements from six generations of Intel® Core™ processors running a variety of vectorizable workloads. We show that through the combination of these techniques, both the time and the energy required to run the workloads has been reduced.

This paper makes three contributions to the understanding of energy efficiency.

1. **Methodology**: We present a methodology for empirically measuring instruction-level energy efficiency on a modern processor. This includes techniques for isolating architectural features as well as a novel technique for studying the relationship between performance and power.
2. **Attribution**: By isolating key architectural features, we account for many of the variable costs which impact energy efficiency. This provides valuable insight into the driving forces behind energy efficiency on a real-world processor.
3. **Evolution of Energy Efficiency**: Using a longitudinal study of the Intel® Core™ processor, we track the impact architectural innovations have had on energy efficiency. This motivates a discussion about the future strategies for improving energy efficiency on a Big Core.

2. Methodology

Our goal is to understand the impact architectural features have on performance and energy efficiency of real-world processors. Ideally, a properly controlled experiment would involve designing and manufacturing processors, with different permutations of features, across several process technology nodes; however, this approach is infeasible. Instead, we rely on comparisons of existing processors. To control the variables we focus on successive generations of a single architecture family, where each processor has a similar base architecture with modest incremental changes in architecture features each generation. With the proper setup, these cross-generational comparisons provides a unique before-and-after evaluation of newly added architectural features.

This section provides more detailed descriptions of our experimental methodology and setup.

2.1. Processors

Table 1 lists the processors used in the study: Penryn (PNY), Nehalem (NHM), Westmere (WSM), Sandy Bridge (SNB), Ivy Bridge (IVB), and Haswell (HSW). In total, our data spans six years, six generations, four major microarchitecture revisions, and three process technology nodes.

We selected flagship processor models from the high-end enthusiast segment rather than trying to match features exactly. Fortunately, the clock speed, core count, and last level cache capacity are relatively similar. The L1 data cache is a constant 32 KB across all processors. The largest discrepancy is WSM, which, for business reasons, was designed with two extra cores. For consistency, we disabled those cores and treated WSM as a four-core processor. To account for the minor clock frequency difference, performance is measured in cycles rather than time. Finally, since all memory accesses in these experiments are confined to the L1 cache, slight differences in the rest of the cache hierarchy, mid-level cache (MLC), last-level cache (LLC), memory controller, and the rest of the uncore should not have a significant impact on results.

2.2. Architecture Features

The core features we studied include (i) the frontend caches, which include the loop cache and the micro-op cache (ii) the out-of-order resources, which include the execution units, issue port, and the number of reorder buffer entries and (iii) the SIMD execution unit and ISA extension. Details of these features are listed in Table 2.

2.3. Kernels

We use the Livermore Loops benchmark suite [24] (see Table 3), a collection of compute intensive kernels extracted from scientific applications used by the Lawrence Livermore National Laboratory, to evaluate the processors.[1] The Livermore Loops derive from actual high-performance computing applications, yet are small enough to instrument, study, and control manually. We also acknowledge that the kernels are not necessarily the optimal C code implementation of the computation, but are instead canonical examples of typical scientific code. Note, the original Livermore Loops benchmark set contains 24 kernels, however, we had to omit five kernels because they do not fit basic loop structure used in our experimental framework.

For our experiments, we compiled each of the loops with the Intel® C Compiler 13.0.0.079, extracted the assembly instructions from each loop nest, then made minor modifications to remove any dependence of the loop body on any explicit loop iteration variables. These modifications provide additional control over execution by ensuring all memory accesses hit in the L1 cache, allowing us to run the loop for an unlimited number of iterations, and controlling the input values of all operations. Figure 1 shows the original Livermore Loops C code, the compiled assembly, and the modified version of the assembly for Kernel 7. The lines in red highlight the modifications to the compiled assembly, which confine the memory access pattern to a smaller working set by keeping register *rdi* constant. Although it is not shown in the figure, we also initialize all of the registers and memory locations that are touched by the kernel to ensure that dynamic behavior remains constant each iteration. Figure 1 also contains a table with a sample of the performance, power, and energy data collected for Kernel 7.

The resulting kernels have between 12 and 183 instructions inside a single for loop, which we run for several billion iterations. This enables us to make very precise measurements of performance and power. We also use performance counters and simulators to analyze runtime behavior.

For illustration purposes, we also added an additional kernel to the original nineteen kernels: Kernel 20 – Peak floating-point throughput. This kernel is designed to attain the theoretical peak floating-point throughput. Both a vector multiply (mulpd) and vector add instruction (addpd) are issued each cycle. There are no memory accesses or auxiliary instructions in this kernel. We do not include Kernel 20 when calculating averages in the results section.

2.4. Experimental Platform

Our results are based on empirical measurements of CPU power, which are measured by instrumenting the 12 volt rail that feeds the voltage regulator which in turn feeds the processor. To account for minor voltage fluctuations, both voltage and current are measured to calculate power. A National Instruments® cDAQ-9174 with a 9229 module is used to sample power at a rate of $2KHz$. This device is managed by a second workstation to ensure that data acquisition does not affect the target machine.

Each system runs a stock Ubuntu 12.04 Server installation – no additional attempts were made to optimize the operating system. The processors are set to run at factory prescribed frequency with Intel® Turbo Boost Technology and Hyperthreading features disabled. The benchmarks were compiled using the Intel® Composer XE version 2013.0.079 tool chain [15].[2]

Preliminary experiments have shown that a rise in processor temperature can increase power by as much as five watts. To minimize this impact, benchmarks are run continuously for a five minute warmup period to allow the temperature to stabilize before starting the experiment. Tests are conducted by running the benchmark continuously for five minutes, immediately following the five minute warmup period. Power and performance values are averaged across the entire five minute trial. Repeated runs of the same experiment show the experimental precision of power measurements to be within

[1] Note, all floating-point calculations within these kernels are based on double-precision, not single-precision, floating-point types.

[2] Intels compilers may or may not optimize to the same degree for non-Intel processors for optimizations that are not unique to Intel processors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Processor-dependent optimizations in this product are intended for use with Intel processors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel processors. Please refer to the applicable product User Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804.

Model	Core X9650	Core i7 975	Core i7 980X	Core i7 2700K	Core i7 3770K	Core i7 4770K
Code Name	Penryn (PNY)	Nehalem (NHM)	Westmere (WSM)	Sandy Bridge (SNB)	Ivy Bridge (IVB)	Haswell (HSW)
Release Cycle	Tick	Tock	Tick	Tock	Tick	Tock
Core Microarchitecture	Core	Nehalem	Nehalem	Sandy Bridge	Sandy Bridge	Haswell
Process Node	45 nm	45 nm	32 nm	32 nm	22 nm	22 nm
Frequency	3.00 GHz	3.33 GHz	3.33 GHz	3.5 GHz	3.5 GHz	3.5 GHz
Cores	4	4	6	4	4	4
LLC	12 MB	8 MB	12 MB	8 MB	8 MB	8 MB
TDP	130 W	130 W	130 W	95 W	77 W	84 W
Release Date	Q4'07	Q2'09	Q1'10	Q1'11	Q2'12	Q2'13

Table 1: The six processor models used in this study.

Kernel #7 (Equation of State)

Livermore Loops C Code:

```
for ( k=0 ; k<n ; k++ ) {
        x[k] = u[k] + r*( z[k] + r*y[k] ) +
              t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) +
              t*( u[k+6] + r*( u[k+5] + r*u[k+4] ) ) );
}
```

	Cycles Per Iteration	CPU Power (Watts)	Energy Per Iteration
PNY	23.0	60.5	116 nJ
NHM	22.5	73.2	124 nJ
WSM	22.5	50.5	85 nJ
SNB	18.3	69.0	90 nJ
IVB	18.3	34.3	45 nJ
HSW	17.3	40.1	50 nJ

Compiled Assembly:

```
innerloop:
vmulpd    ymm15, ymm2, [32+r15+rdi*8]
vmovupd   xmm7,  [8+r15+rdi*8]
vmovupd   xmm13,  [40+r15+rdi*8]
vmulpd    ymm3, ymm2,  [r12+rdi*8]
vmovupd   xmm6,  [24+r15+rdi*8]
vaddpd    ymm4, ymm3, [r10+rdi*8]
vmulpd    ymm5, ymm2, ymm4
vaddpd    ymm0, ymm5, [r15+rdi*8]
vinsertf128 ymm8, ymm7, [24+r15+rdi*8], 1
vinsertf128 ymm14, ymm13, [56+r15+rdi*8], 1
vmulpd    ymm9, ymm2, ymm8
vaddpd    ymm13, ymm14, ymm15
vaddpd    ymm10, ymm9, [16+r15+rdi*8]
vmulpd    ymm14, ymm2, ymm13
vmulpd    ymm12, ymm2, ymm10
vaddpd    ymm15, ymm14, [48+r15+rdi*8]
vmulpd    ymm4, ymm1, ymm15
vinsertf128 ymm11, ymm6, [40+r15+rdi*8], 1
vaddpd    ymm3, ymm11, ymm12
vaddpd    ymm5, ymm3, ymm4
vmulpd    ymm6, ymm1, ymm5
vaddpd    ymm0, ymm0, ymm6
vmovupd  [r13+rdi*8], ymm0
add    rdi, 1
cmp    rdi, r9
jb     innerloop
```

Modified Assembly:

```
innerloop:
vmulpd    ymm15, ymm2, [32+r15+rdi*8]
vmovupd   xmm7,  [8+r15+rdi*8]
vmovupd   xmm13,  [40+r15+rdi*8]
vmulpd    ymm3, ymm2,  [r12+rdi*8]
vmovupd   xmm6,  [24+r15+rdi*8]
vaddpd    ymm4, ymm3, [r10+rdi*8]
vmulpd    ymm5, ymm2, ymm4
vaddpd    ymm0, ymm5, [r15+rdi*8]
vinsertf128 ymm8, ymm7, [24+r15+rdi*8], 1
vinsertf128 ymm14, ymm13, [56+r15+rdi*8], 1
vmulpd    ymm9, ymm2, ymm8
vaddpd    ymm13, ymm14, ymm15
vaddpd    ymm10, ymm9, [16+r15+rdi*8]
vmulpd    ymm14, ymm2, ymm13
vmulpd    ymm12, ymm2, ymm10
vaddpd    ymm15, ymm14, [48+r15+rdi*8]
vmulpd    ymm4, ymm1, ymm15
vinsertf128 ymm11, ymm6, [40+r15+rdi*8], 1
vaddpd    ymm3, ymm11, ymm12
vaddpd    ymm5, ymm3, ymm4
vmulpd    ymm6, ymm1, ymm5
vaddpd    ymm0, ymm0, ymm6
vmovupd  [r13+rdi*8], ymm0
add      r8, 1
cmp      r8, r9
jb innerloop
```

Figure 1: This figure shows Kernel 7 and a sample of the data collected for this kernel. To conserve space, the assembly code listings contains the AVX version (26 instructions) instead of the longer SSE version (51 instructions); however, the data table contains values collected from the SSE version.

0.05 watts.

When measuring power, we run an instance of the kernel on each core to amplify the dynamic power consumption. Therefore, the measured power values presented in this paper represent the total power of the processor when each of the four cores are running an instance of the kernel. However, when we refer to IPC we are referring to statistics of an individual core.

2.5. Definition of Energy Efficiency

This paper is primarily focused on energy efficiency. Broadly speaking, the objective is to minimize the total amount of energy consumed by the processor to complete a particular computation. We calculate energy consumption (measured in joules) by measuring the average power of the processor (measured in watts; note, $1\ watt = 1\frac{joule}{sec}$) and multiplying by the duration of the computation (measured in seconds but reported in clock cycles). For example, the average energy consumed per iteration of Kernel 7 (see Figure 1) on HSW is[3]

$$(17.3\ cycles)(\frac{1 sec}{3.5 \times 10^9\ cycles})(10.0\ W) = 50\ nJ. \quad (1)$$

[3]To get the cost of a single iteration, we must also divide processor power by four since the benchmark is running simultaneously on each of the four cores (see Section 2.4).

	Core	Nehalem	Sandy Bridge	Haswell
L1 Bandwidth (load, store)	16, 16 Bytes per cycle	16, 16 Bytes per cycle	32, 16 Bytes per cycle	64, 32 Bytes per cycle
Instruction Cache	32KB L1 Icache	32KB L1 Icache	32KB L1 Icache, 1.5K uop cache	32KB L1 Icache, 1.5K uop cache
Reorder Buffer	96 entries	128 entries	168 entries	192 entries
Ins/Uop Queue	32 ins	28 uops	28 uops (56 on IVB)	56 uops
SIMD Extensions	SSE	SSE	AVX	AVX2

Table 2: Architectural Features

Kernel	Description
1	Hydro fragment
2	Incomplete Cholesky Conjugate Gradient
3	Inner product
4	Banded linear equations
5	Tri-diagonal elimination, below diagonal
6	General linear recurrence equations
7	Equation of state fragment
8	Integrate predictors
9	Difference predictors
10	First sum
11	First difference
12	2-D PIC (Particle In Cell)
13	1-D PIC (Particle In Cell)
14	ADI integration
15	2-D explicit hydrodynamics fragment
16	General linear recurrence equations
17	Discrete ordinates transport
18	Matrix-matrix product
19	2-D implicit hydrodynamics fragment
20	Peak floating-point throughput

Table 3: The 20 kernels evaluated in this study.

Energy efficiency is inversely proportional to energy consumption. It is generally expressed as performance over power,

$$\frac{1}{energy} = \frac{1}{duration \times power} = \frac{performance}{power}. \quad (2)$$

If performance is constant then the change in energy efficiency is proportional to the change in power. For example, based on the data in Figure 1, SNB and IVB have identical performance on Kernel 7. The ratio of average power shows that IVB is 2.0x more efficient than SNB.

In many situations, performance and energy efficiency are treated as competing goals since it is generally possible to improve performance at the expense of energy efficiency and vice versa. Occasionally, an artificial metric that incorporates both performance and energy efficiency, such as the energy delay product [12], is used to determine whether a trade-off between the two is beneficial. Fortunately, this paper does not explicitly encounter this trade-off and therefore does not resort to alternative metrics.

2.6. Register Scrambling

The analysis in this paper includes a novel technique, called register scrambling, to artificially alter the instruction-level dependencies of a kernel. As the name suggests, it involves randomly assigning new SSE/AVX register numbers to each instruction. For example, Figure 2 shows two examples of Kernel 1 with the registers scrambled. The instruction mix remains constant, but the performance and power change as the instruction-level parallelism changes. By generating several different "scrambles" of the kernel we are able to generate a regression that relates performance to power. Figure 3 shows the relationship between performance and power of Kernel 1 on HSW. This regression can be used to evaluate improvements in the core architecture by estimating how much a change in performance will affect the energy efficiency.

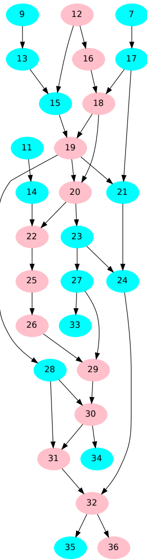
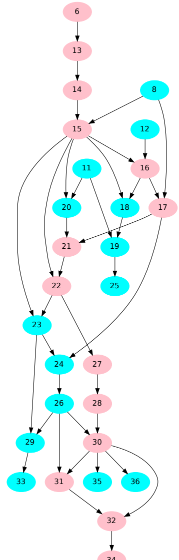
3. Experimental Results

Figure 4 presents the main results of this paper. Averaging across all of the Livermore Loop kernels, there is a 2.9x improvement in energy efficiency from PNY to HSW.

Furthermore, this section drills-down on the source of the improvements in energy efficiency by progressively removing the benefit of recent architectural features. This is completed as a five step process:

1. **SIMD Extensions**: we restrict the use of relevant ISA extensions that have been added since PNY: the AVX and AVX2 SIMD extensions.
2. **Frontend Innovations**: we prevent the processor from utilizing new frontend features. The primary innovations since PNY are the addition of a micro-op cache and improvements to the loop caches.
3. **Backend Innovations**: we estimate the impact performance improvements due to backend innovations —such as additional execution units and larger instruction windows—have on power and energy efficiency.
4. **22nm Process Technology Node**: we rollback the advantage of the 22nm process technology to estimate power and energy efficiency on the 32nm process technology node.
5. **32nm Process Technology Node**: we rollback the advantage of the 32nm process technology to estimate power and energy efficiency on the 45nm process technology node.

Figure 4 shows the improvement in energy efficiency after each step. For example, the HSW plot shows that the im-

Scramble #1

```
0  inloop:
1  movsd      xmm1, [88+r12+r9*8]
2  movsd      xmm1, [104+r12+r9*8]
3  movsd      xmm2, [120+r12+r9*8]
4  movsd      xmm2, [136+r12+r9*8]
5  movaps     xmm0, [80+r12+r9*8]
6  movhpd     xmm1, [96+r12+r9*8]
7  movaps     xmm2, [96+r12+r9*8]
8  movhpd     xmm3, [112+r12+r9*8]
9  movaps     xmm1, [112+r12+r9*8]
10 movhpd     xmm0, [128+r12+r9*8]
11 movaps     xmm0, [128+r12+r9*8]
12 movhpd     xmm3, [144+r12+r9*8]
13 mulpd      xmm1, xmm1
14 mulpd      xmm0, xmm0
15 mulpd      xmm1, xmm3
16 mulpd      xmm3, xmm3
17 mulpd      xmm2, xmm2
18 mulpd      xmm3, xmm2
19 mulpd      xmm1, xmm3
20 mulpd      xmm3, xmm1
21 addpd      xmm2, xmm1
22 addpd      xmm0, xmm3
23 addpd      xmm3, xmm3
24 addpd      xmm2, xmm3
25 mulpd      xmm0, [r15+r9*8]
26 mulpd      xmm0, [16+r15+r9*8]
27 mulpd      xmm3, [32+r15+r9*8]
28 mulpd      xmm1, [48+r15+r9*8]
29 addpd      xmm0, xmm3
30 addpd      xmm0, xmm1
31 addpd      xmm1, xmm0
32 addpd      xmm1, xmm2
33 movaps     [r11+r9*8], xmm3
34 movaps     [16+r11+r9*8], xmm0
35 movaps     [32+r11+r9*8], xmm1
36 movaps     [48+r11+r9*8], xmm1
37 add        r8, 1
38 cmp        r8, rbx
39 jb         inloop
```

Cycles per Iteration:	**31.51 (IPC=1.24)**
Average Power:	**37.32 watts**
Energy per Iteration:	**84.0 nJ**

Scramble #2

```
0  inloop:
1  movsd      xmm2, [88+r12+r9*8]
2  movsd      xmm0, [104+r12+r9*8]
3  movsd      xmm0, [120+r12+r9*8]
4  movsd      xmm3, [136+r12+r9*8]
5  movaps     xmm0, [80+r12+r9*8]
6  movhpd     xmm3, [96+r12+r9*8]
7  movaps     xmm0, [96+r12+r9*8]
8  movhpd     xmm2, [112+r12+r9*8]
9  movaps     xmm1, [112+r12+r9*8]
10 movhpd     xmm0, [128+r12+r9*8]
11 movaps     xmm1, [128+r12+r9*8]
12 movhpd     xmm0, [144+r12+r9*8]
13 mulpd      xmm3, xmm3
14 mulpd      xmm3, xmm3
15 mulpd      xmm3, xmm2
16 mulpd      xmm0, xmm3
17 mulpd      xmm2, xmm0
18 mulpd      xmm0, xmm3
19 mulpd      xmm0, xmm1
20 mulpd      xmm1, xmm3
21 addpd      xmm1, xmm2
22 addpd      xmm1, xmm3
23 addpd      xmm3, xmm1
24 addpd      xmm2, xmm3
25 mulpd      xmm0, [r15+r9*8]
26 mulpd      xmm2, [16+r15+r9*8]
27 mulpd      xmm1, [32+r15+r9*8]
28 mulpd      xmm1, [48+r15+r9*8]
29 addpd      xmm3, xmm2
30 addpd      xmm1, xmm2
31 addpd      xmm2, xmm1
32 addpd      xmm2, xmm1
33 movaps     [r11+r9*8], xmm3
34 movaps     [16+r11+r9*8], xmm2
35 movaps     [32+r11+r9*8], xmm1
36 movaps     [48+r11+r9*8], xmm1
37 add        r8, 1
39 cmp        r8, rbx
39 jb         inloop
```

Cycles per Iteration:	**19.65 (IPC=1.98)**
Average Power:	**42.02 watts**
Energy per Iteration:	**59.0 nJ**

Figure 2: An example of two "scrambled" versions of Kernel 1. This figure also includes sample data from HSW.

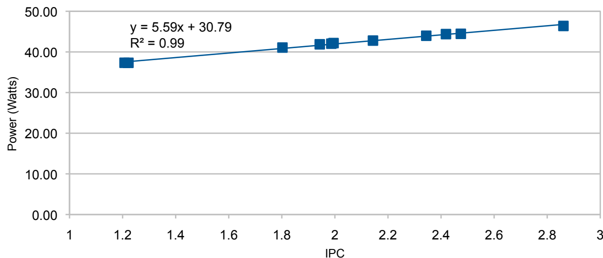


Figure 3: A regression of several scrambles of Kernel 1 on HSW.

provement in energy efficiency drops from 2.9x to 2.1x after removing the use of SIMD extensions (step 1). Similarly, after also removing the impact of frontend innovations (step 2), the improvement drops from 2.1x to 1.8x. After, removing the benefits of the SIMD Extensions, frontend, and backend as well as the process technology, we are left with the "base" energy efficiency. The "base" value represents the efficiency of HSW if it pays the overhead cost of implementing all of its architectural features —such as the micro-op cache and additional execution units— but does not benefit from any of them.

It is important to note that this process is conducted in an additive process – the changes made in previous steps are retained in the following steps. This methodological decision was made out of necessity since each processor needs to be running identical code in order to properly compare the frontends. Similarly, we need to neutralize changes in the frontend and ISA before comparing the backends.

The details of each step are described in the following subsections.

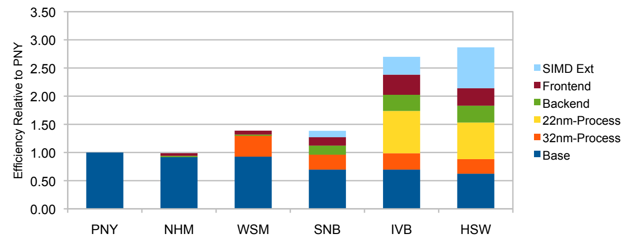


Figure 4: Average improvement in energy efficiency across the Livermore Loops kernels.

3.1. SIMD Extensions

The AVX extensions introduce three important features to the ISA that are relevant to the Livermore Loops: (1) four-wide SIMD vector instructions (double the SIMD width of SSE instructions) (2) non-destructive instructions, and (3) 256-bit load and store instructions (double the width of SSE instructions) [11]. In addition, the AVX2 extensions also provide fused multiply-add (FMA) instructions which double the peak floating-point throughput and gather instructions (e.g., vgatherdpd) for vectorizing non-adjacent memory accesses.

To evaluate the impact of the AVX and AVX2 extensions, we create multiple versions of each kernel. The SSE version — the baseline version— is created by constraining the compiler to only generate code that is supported by SSE4.1 machines, then extracting the resulting loop body. The AVX and AVX2 versions are generated by allowing the compiler to take advantage of the additional AVX instructions. We gauge the effectiveness of the extensions by comparing the performance and energy efficiency of the resulting versions.

Results from Kernel 20 (the peak floating-point throughput kernel) on HSW demonstrate the potential benefits of SIMD extensions. Doubling the SIMD width with AVX instructions doubles performance and only increases total power by 4.3%, which results in a 1.9x improvement in energy efficiency. Similarly, the use of the FMA instructions can double the performance and only increase power by 5.0%, which also nearly doubles energy efficiency. By fully exploiting both the FMA instructions and the wide SIMD width of the AVX instructions, HSW can achieve up to 6.3 GFlops/watt, a 7x improvement over the 0.9 GFlops/watt on PNY.

Unlike the ideal conditions, the AVX and AVX2 extensions have only a moderate impact on the Livermore Loops kernels. Eleven of the nineteen kernels (1, 2, 3, 4, 7, 8, 10, 12, 13, 14, and 18) benefit from the additional SIMD instructions – the other kernels have structural dependencies which make vectorization difficult. Of the kernels that do benefit, energy efficiency improves 4-56% and performance improves 7-83%. In each of these cases the improvement in performance is greater than the improvement in energy efficiency primarily because utilizing SIMD instructions often requires a little extra work to shuffle data into and out of the proper SIMD lanes. Interestingly, in two of the kernels (Kernel 4 and Kernel 9), the use of AVX instructions had a negative impact on both performance and energy efficiency even though the AVX version of the kernel uses fewer instructions. Averaging across all kernels, the AVX extensions delivers a 21% improvement in performance and a 16% improvement in energy efficiency. Utilizing both AVX and AVX2 instruction on HSW provide a 26% improvement in performance and a 21% improvement in energy efficiency over the SSE version.

3.2. Frontend Features

The frontend is responsible for fetching and decoding instructions to feed the execution engines in the backend. In an out-of-order architecture, the frontend is integrated with a branch prediction unit so it can fetch and decode instructions down speculative paths in order to keep the backend busy. In addition, modern processors are equipped with loop detectors and instruction caches to reduce the number of instructions that must be fetched and decoded, which improves both performance and energy efficiency. This section examines how the evolution of the frontend has impacted energy efficiency.

3.2.1. Loop Caches The loop cache exploits the temporal locality of instructions to reduce the burden on the frontend [20, 4]. When used, instructions from inside a loop nest are streamed directly from the Instruction Queue (IQ) or Micro-op Queue (MQ), skipping earlier stages of the pipeline. In PNY, the cache is located between the instruction predecode and the instruction decoders. In subsequent generations the cache is located after the decode units, allowing the frontend to power-down the instruction fetch and decode units when streaming from the MQ [26].[4] Ultimately, the loop cache improves performance by eliminating potential frontend bottlenecks and reduces power by eliminating redundant work.

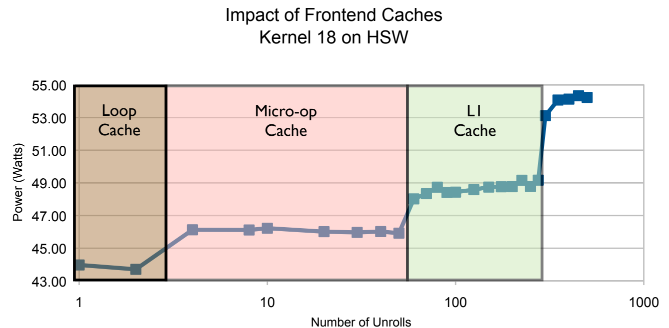


Figure 5: This figure demonstrates the results from manually unrolling Kernel 18 on HSW. Kernel 18 has 19 instructions, not counting the branch instruction. When unrolls=10 the loop nest contains 190 instructions. Performance remains constant until the instruction stream exceeds the capacity of the L1 instruction cache at unrolls =300.

3.2.2. Mico-op Cache In addition to the loop cache, SNB, IVB, and HSW also include a micro-op cache in the frontend, between the loop cache and the L1 instruction cache. It provides many of the benefits of the loop cache but has a higher capacity (1500 micro-ops versus the 56 micro-ops in the loop cache). Like the loop cache, the micro-op cache can reduce the number of instruction decodes by caching decoded instructions. However, unlike the loop cache, the micro-op cache exists earlier in the pipeline and therefore does not bypass the branch prediction unit. The results in Figure 5 and Figure 6 show that utilizing the loop or micro-op cache can reduce the power by several watts but the power saving advantage of the loop cache over the micro-op cache is relatively low.

We measure the impact of the loop cache and the micro-op cache by comparing the original kernel with a version that has been manually unrolled to the point that the instruction stream exceeds the capacity of the caches. In most cases the kernel only needed to be unrolled two to four times before the loop nest exceeded the capacity of the loop cache; however, some of the kernels are too large to fit in the loop cache even without unrolling. Table 2 lists IQ capacity of the different processors. Performance counters are used to verify the utilization of the loop cache.

[4] Since the loop cache is located before the decoders in PNY, it caches instructions in the Instruction Queue (IQ); whereas later generations cache decoded micro-ops in the Micro-op Queue (MQ).

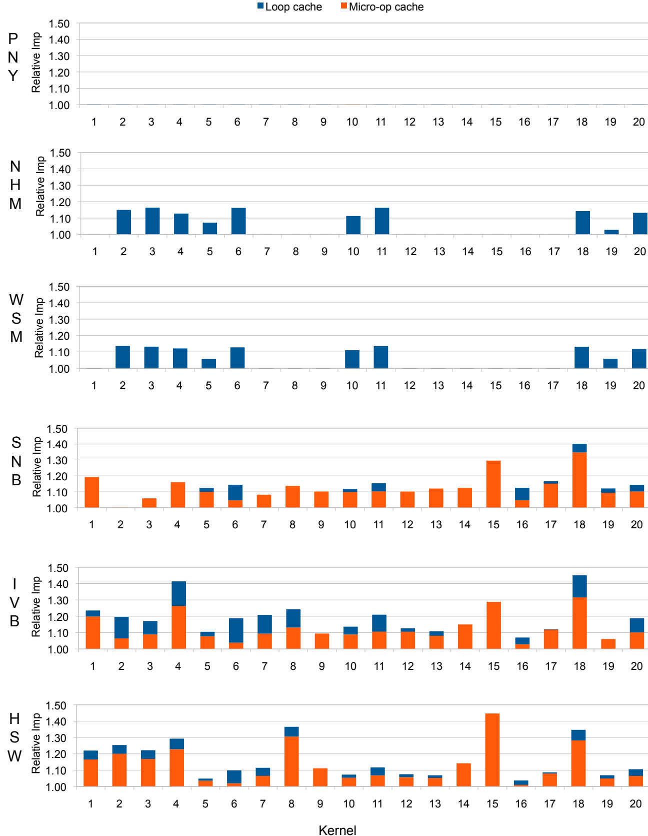


Figure 6: Improvement in energy efficiency from the micro-op and loop caches. Blue denotes the contribution of the loop caches and orange denotes the contribution of the micro-op caches.

Figure 6 shows the results from the loop unrolling. In the case of PNY, only one kernel (Kernel 11) fits into the loop cache because of the size of most kernels exceeds the limited capacity of the cache. The additional capacity in NHM and WSM increases the number of kernels that can utilize the loop cache to six. We also notice that locating the loop cache after the decode stage of the pipeline has increased the energy savings from 5% in PNY to 15% in NHM and WSM. Similarly, the larger MQ in SNB, IVB, and HSW further increases the number of kernels that utilize the loop cache; however, in this case the improvement in energy efficiency is relatively small because the unrolled kernels only fall back to the micro-op cache, still avoiding the fetch and decode phases of the frontend.

3.3. Backend Features

Comparing HSW to PNY with the SSE version of the Livermore Loops, there has been a 1.4x improvement in per-cycle performance which can be attributed to architectural improvements to the backend. While increasing performance is beneficial on its own, it also affects energy efficiency by reducing execution time and increasing power (recall Equation 2). In this section, we analyze the source of improvements in performance and energy efficiency of the backend.

3.3.1. Additional Execution Units Superscalar processors exploit instruction-level parallelism by issuing multiple instructions per cycle. It improves performance and also improves energy efficiency by reducing the execution time. Since PNY, several additional execution units have been added to the core microarchitecture. The most relevant additions are the second load unit added to SNB, which doubles the L1 load bandwidth, and several redundant execution units added to HSW.

3.3.2. Additional Out-of-order Scheduling Resources To take advantage of additional execution units, the out-of-order scheduling resources must be increased to augment the discovery and scheduling of independent instructions. Among these resources, the number of reorder buffer (ROB) entries directly affects the number of parallel instructions in flight. The ROB holds the operands necessary for the instructions' operations and hold the results before they are written back to the architectural registers. More ROB entries allow more concurrent instructions in the pipeline. The instruction window controls the number of instructions in the execution flow that the processor can analyze for parallel execution. A larger instruction window allows the processors to examine more instructions, which increases the chances of finding independent instructions to be executed concurrently. Table 2 shows the increase in the size of the reorder buffer and the instruction window from PNY to HSW.

3.3.3. Backend Experiments and Results Ideally, the proper approach to quantifying the benefit of the backend improvements is to defeature a processor to make its backend behave like the previous generation's backend. Unfortunately, we do not have the type of controls in modern processors to make this work. Instead, we approximate this approach by manually isolating changes in the backend. For these experiments we use the SSE version of the kernels and unroll them enough so the instruction streams do not fit into the micro-op caches, thereby eliminating affects from the SIMD extensions and frontend. Furthermore, to remove the impact of the manufacturing process, we compare processors from the same process node (i.e., NHM to PNY, SNB to WSM, and HSW to IVB).

Overall, backend features improved performance more than energy efficiency. From PNY to NHM, five of the 19 kernels improved performance, but only three improved energy efficiency. From WSM to SNB, 13 of the 19 kernels improved performance and only three improved energy efficiency. From IVB to HSW, 12 of the 19 kernels improved performance and only two improved energy efficiency. The reason behind these results is that backend improvements require a substantial increase in transistor count which leads to a significant increase in capacitance and power to toggle them.

Using performance counters and architecture simulators [17, 16], we study the reduction in pipeline stalls from one generation to the next. Based on this information, we can discern what feature is responsible for the change in performance.

From PNY to NHM, we found that the performance benefits mainly come from increased scheduling resources. From WSM to SNB, we found that the majority of the performance benefits come from the addition of a second load port. Many of the kernels are L1 bandwidth limited on NHM, therefore the additional port alleviates this performance bottleneck. From IVB to HSW, we found that several of the kernels that benefited from the additional load port in SNB now benefit from the additional scheduling resources as the L1 bandwidth is no longer a significant performance bottleneck.

Finally, we use Equation 2 to calculate the effect these performance improvements have on energy efficiency. Since these performance improvements will increase power due to the increase in utilization, we use the Register Scrambling technique (see Section 2.6) to map changes in performance to changes in power. For example, Kernel 1 has an IPC of 2.37 for PNY and 2.79 for HSW. Using the regression model for Kernel 1 on HSW (see Figure 3), we can determine that this 1.18x improvement in performance equates to a 1.05x increase in power (from 44.0 watts to 46.4 watts). Overall, this increase in power is countered with a more significant decrease in runtime which translates to a net 1.11x improvement in energy efficiency.

3.4. Process/Circuits Innovation

Process technology innovations have been the primary driver of improvements in energy efficiency over the past several decades. These advances enable manufacturers to produce smaller transistors that can operate a lower supply voltages and reduced capacitance, both of which reduce dynamic switching energy. Although, smaller transistors also have a higher leakage ratio, which contribute to higher overall static power dissipation.

We evaluate the impact of process technology improvements by comparing microarchitectures across process technology nodes: NHM with WSM and SNB with IVB. Since the core microarchitecture is largely unchanged, the change in power can be primarily attributed to improvements in circuits and process technology. Figure 7 shows a scatter plot comparing the average power of a kernel on NHM with the average power of the same kernel on WSM. The regression model suggests that moving from the 45nm to the 32nm process technology node has increased static power by 7.55 watts but reduced dynamic power by a factor of 0.57. The increase in the static power is partially due to higher leakage and partially due to a bigger die with two more cores in WSM – we believe the increase in static power would be lower if Intel had produced an equivalent processor with four rather than six cores. Figure 8 shows a similar trend for the progression from 32nm to 22nm process technology nodes. In this case, both the dynamic and static power reduced by a factor of 0.68 and by 10.48 watts respectively. The reduction in static power has been discussed as a benefit of the transition from planar transistors in the 32nm process node to the 3D Tri-Gate transistors which debuted in the 22nm process node [1]. The 40% reduction in dynamic power in both cases is in line with ITRS Roadmap projections [3].

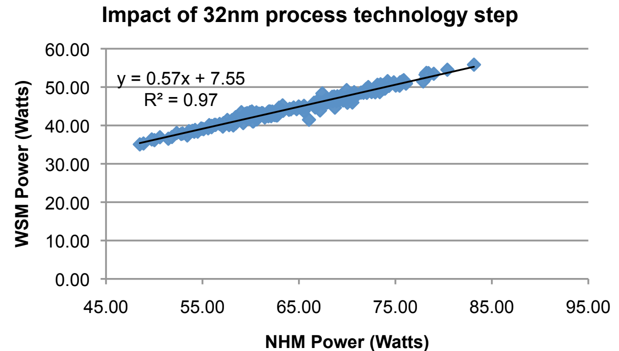


Figure 7: Improvement in energy efficiency attributed to 32nm process technology step.

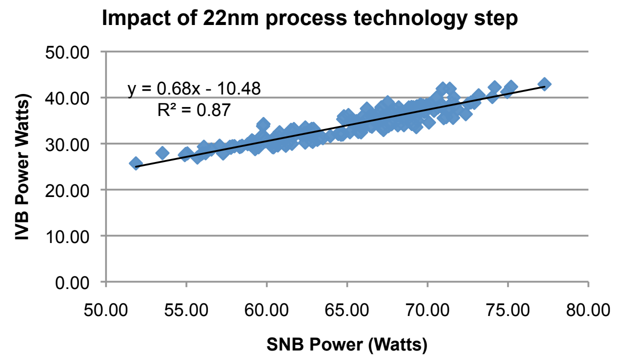


Figure 8: Improvement in energy efficiency attributed to 22nm process technology step.

4. Discussion

Taking out the impact of process technology changes, microarchitecture changes have increased per-cycle performance 1.9x and energy efficiency by 1.2x. Ignoring the overhead cost of implementing these features, 40% of the energy efficiency can be attributed to SIMD width, 35% to frontend features, and 25% to backend features.

This section section makes several additional observations about the evolution of energy efficiency in these processors.

4.1. Energy per instruction (EPI) depends highly on IPC

Efficiency is often gauged by calculating the effective energy per instruction (EPI). EPI makes the most sense as a metric when it remains constant, independent of IPC. However, because real machines expend energy at a certain minimum rate even when idle, actual EPI becomes a function of IPC. Figure 9 demonstrates the effective EPI of HSW as a function of

IPC. The effective cost of an instruction drops from 4.1 nJ at IPC=0.5 to 0.8 nJ when IPC=4. The plot also shows that IPC has a more dramatic impact on EPI than the actual instruction mix.

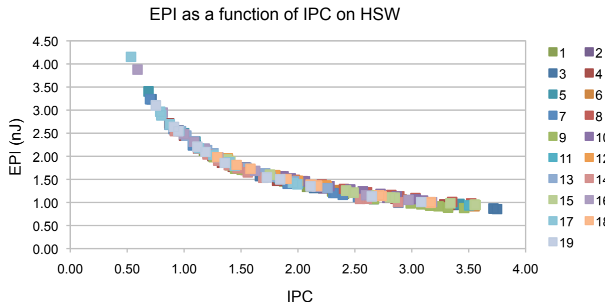


Figure 9: EPI as a function of IPC on HSW. Data samples come from ten "scrambles" of each of the Livermore Loops kernels. Data points are colored according to the kernel.

4.2. Fixed costs dominate variable costs

The relationship between IPC and EPI is the result of high static power which is independent of core utilization. We can approximate this overhead by modeling power as a function of IPC (see Figure 10). Table 4 lists a linear regression relating power to IPC on each of the processors. As the table shows, when IPC=2, HSW consumes 11.05 nJ per cycle. Of that cost, 8.31 nJ comes from a fixed overhead cost and 2.74 nJ comes from the variable cost of the instructions. As this data implies, the actual operation cost of an instruction (e.g., the floating-point arithmetic implied by a floating-point instruction) is only a small fraction of the total power of the CPU. This is a typical consequence of general purpose processors [25]. Looking at the trend from PNY to HSW, the variable costs drops with each processor generation, with substantial drops corresponding to the changes in the process technology node. The fixed costs drop during process technology node steps (NHM->WSM, SNB->IVB), but increase when subsequent microarchitectures are introduced (PNY->NHM, WSM->SNB, IVB->HSW).

Comparing HSW to PNY, a larger fraction of the total energy is spent on fixed costs. Two reasons contribute to this trend. First, as modern processors integrate more cores together on a single chip, the size of the uncore grows [22]. In our experiment, the uncore is not exercised, but it is not clock gated either. Second, increasing core performance has a cost in area and power. However, the transfer from variable costs to fixed costs is not necessarily bad. For example, the introduction of the micro-op cache increases the fixed cost of the core by adding a cache, but it also reduces the variable cost by reducing the number of instruction decodes and fetches.

Furthermore, comparing HSW to IVB reveals how the new features in HSW affect both the fixed and variable costs. The microarchitectural evolution from IVB to HSW has increased the fixed cost by 22%. Unless HSW is able to exploit these features to improve performance, at IPC=2.0 IVB will be 14% more efficient than HSW. To overcome this deficit, HSW must sustain an IPC of 2.38 to match the energy efficiency of IVB at IPC=2.0 or rely on SIMD extensions to make each instruction more productive.

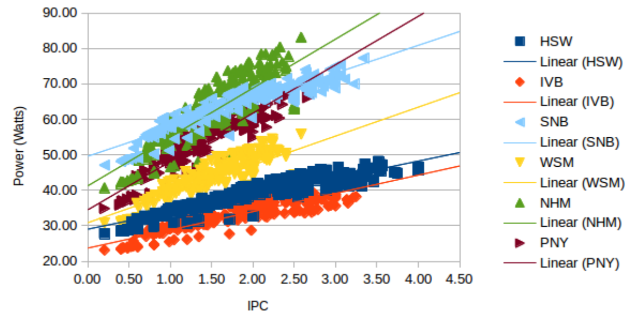


Figure 10: Power as a function of IPC. Data samples come from ten "scrambles" of each of the Livermore Loops kernels.

4.3. Performance improvements exceed power increases

According to the "base" value in Figure 4, the increased complexity of the HSW core has increased the overhead cost by nearly 80% when compared to PNY. To avoid a net increase in energy consumption, improvements in performance should be greater than 80%. In the case of the Livermore Loops, the average performance increases 90%.

4.4. Frontend features reduce the tax of complex instructions

As mentioned earlier, as the complexity of the core increases, more energy is devoted to the fixed costs. However, the increase in fixed cost does not necessarily reduce energy efficiency. For example, the frontend caches increase the fixed cost, but ultimately improve energy efficiency by dramatically reducing amount of energy spent fetching and decoding instructions.

When compared to the Reduced Instruction Set Computing (RISC), Complex Instruction Set Computing (CISC) is generally considered less efficient. CISC does not have uniform instruction length, which adds significant complexity to the instruction fetch and decode logic. Based on the results in section 3.2 we can approximate the power of the fetch and decode units by comparing the power when instructions are streamed from the micro-op cache —which caches decoded micro-ops and therefore can bypass the fetch and decode stages— with the power when the instruction are streamed from the L1 instruction cache. Figure 5 shows that on HSW, the fetch and decode overhead consumes about five watts of power which is roughly 12.5% of the average 40 watts of power consumed by HSW when running the Livermore Loops kernels. Our

	Power Model	Energy Per Cycle (IPC=2)	Fixed Cost Per Cycle (IPC=2)	Variable Cost Per Cycle (IPC=2)	Fixed / Total (IPC=2)
HSW	4.79(IPC) + 29.08	11.05 nJ	8.31 nJ	2.74 nJ	75%
IVB	5.14(IPC) + 23.74	9.72 nJ	6.78 nJ	2.94 nJ	70%
SNB	7.81(IPC) + 49.63	18.64 nJ	14.81 nJ	4.46 nJ	76%
WSM	8.14(IPC) + 30.89	13.97 nJ	9.14 nJ	4.82 nJ	65%
NHM	13.79(IPC) + 41.30	20.39 nJ	12.23 nJ	8.16 nJ	60%
PNY	13.63 (IPC) + 34.54	20.60 nJ	11.51 nJ	9.09 nJ	56%

Table 4: Fixed vs Variable cost analysis of CPU power. The power model is based on a linear regression of the data shown in Figure 10.

results also show that 16 of the 19 kernels fit into the loop cache on HSW and all of the kernels fit in the micro-op cache, which suggest that most of the inefficiencies associated with the CISC decode tax can be eliminated with a well designed frontend.

4.5. SIMD extensions increase the productivity of each instruction with minimal impact on power

It is a well known that SIMD computing is energy efficient. Our results show that under the ideal conditions vector instruction can increase performance with only a nominal increase in dynamic energy consumption. However, as SIMD width increases, its applicability diminishes (see Section 3.1). Even when there is sufficient data parallelism, the additional work necessary to shuffle data into the proper SIMD lanes can limit the advantage of wide SIMD instructions. For example, algorithms with indirect accesses or random accesses require separate scalar operations to load and pack data into these SIMD lanes, which could take away most of the performance and energy efficiency benefit offered by SIMD. Auxiliary instructions —such as the gather instruction, which loads non-continuous data into SIMD registers— are crucial to attaining the full potential of vectorization. Just like the frontend caches which provide the benefit of reducing complex instructions overhead, the gather instructions help make wider SIMD useful for a broader pool of applications.

4.6. High performance computing vs energy efficient computing

Modern semiconductor manufacturing technology enables processors to operate over a wide range of frequency and supply voltages. The processor can operate at a high voltage to support a high clock frequency or a low voltage to reduce power. In our experiments the voltage and frequency used are near the top end of the supply voltage range, which sacrifices energy efficiency for high single-threaded performance. In this regard, our results may not capture the most energy efficiency way of using the processors; it is possible for these processors to be more efficient operating at a lower frequency and voltage. We therefore caution readers to consider the supply voltage of the processor before comparing the energy efficiency across different classes of processors.

5. Related Work

Traditionally, research has focused on the power consumption of individual functional units, yet there has been a growing demand for processor-level analysis of energy efficiency.

At the application level, there have been several studies comparing the energy efficiency of different processors. This has included long-term historical trends [19, 13] as well as comparisons of competing platforms [9, 8, 7]. The demand for energy efficient computing has even motivated the Green500 list, which has become an industry benchmark for comparing the energy efficiency of supercomputers [10]. Of course, the vast differences between these systems (performance levels, ISA, process and manufacturing technology, code quality, etc) makes it difficult to draw any definitive conclusions about the true impact of the underlying microarchitecture.

Bottom-up models and cycle-accurate simulators have been used to evaluate architectural design decisions that affect power [6, 14, 27, 18]. Unfortunately, it is often difficult to validate the relative contribution of the individual components in the underlying model when only the total power of the physical processor can be observed. Furthermore, as the complexity of modern processors swells, it is becoming increasingly difficult to construct a bottom-up model that can accurately capture all relavent processor features.

Alternatively, several groups have studied power consumption by developing instruction-based power models [23, 5, 21]. This approach focuses on correlating hardware performance counters with observed power measurements. A regression analysis is used to assign energy costs to each architectural event. This can then be used to predict power consumption of an arbitrary application based solely on performance counter values. The primary weakness of this approach is its inability to directly account for the full context within which an instruction is executed. For example, as our paper shows, the energy consumed by an instruction that is streamed from a loop cache can be significantly different from one that instead exercises the instruction fetch and decode units, yet the performance counter events only give a limited view of the internal state of the processor core.

6. Future Work and Conclusion

Due to the complexity of modern processors, we had to limit the scope of this project to deliver meaningful insights. We identify several limitations that can be addressed further in future work:

- **Beyond the core**: Our study focuses on the core architecture and thereby tries to minimize the impact of the memory hierarchy, uncore, and system-level components. We acknowledge that these components are an important factor in the overall efficiency of a modern application. Studying the core in isolation provides a foundation for tackling the broader problem.
- **Representative workloads**: The Livermore Loops benchmark suite provides a number of floating-point heavy computations with a variety of instruction-level dependencies. It is well suited for the experiments in our study, but it is not necessarily representative of modern or future workloads. In particular, we want to draw attention to two important characteristics that are absent in our benchmarks: first, the single-loop structure of our benchmarks eliminates any performance advantage of a sophisticated branch predictor because nearly all branches are taken; second, by design all memory accesses hit the L1 cache, neglecting the uncore and higher levels of the memory hierarchy and thereby eliminate dynamic variation in instruction latency. Studying more representative workloads is a natural next step.
- **Scaling voltage and frequency**: Core voltage and clock frequency can have a tremendous impact on energy efficiency. In our experiments, the processors were configured to run at the default voltage and frequency without dynamic scaling. With a specific application in mind, it would be interesting to study how voltage and frequency scaling impact energy efficiency on a fixed architecture.
- **Comparing competing processors**: The quantitative results of this study are specific to the Intel® Core™ processor, yet the conclusions can be applied in general. For a more general purpose study, this approach can also be extended to compare among different processor families involving more dramatic microarchitectural differences, such as a comparison with the Intel® Atom™ or Xeon Phi™ or even comparing processors with different ISAs.
- **Optimizing software for energy efficiency**: Beyond the quantitative numbers presented in this study, this work also demonstrates how carefully controlled experiments can be used to isolate individual architectural features for the purposes of empirically measuring energy consumption. In many cases, key architectural features, such as the loop cache, are transparent from a performance perspective and are thus overlooked. We believe this methodology provides the level of precision necessary for exploring the impact software implementation decisions have on energy efficiency. For example, an evaluation of compiler heuristics can lead to compile time optimizations tailored specifically for reducing energy consumption.

Following the hackneyed business adage, "if you can't measure it, you can't manage it", we firmly believe that long-term improvements in energy efficiency within the field of high performance computing depend on rigorous evaluations of progress. In this spirit, our paper provides an in-depth assessment of energy efficiency on recent generations of the Intel® Core™ processor. Our results indicate that advances in manufacturing and circuits have been the dominate contributor to the improvements in energy efficiency, but architectural innovations have had a positive impact on energy efficiency as well.

7. Acknowledgements

We thank the anonymous reviewers for their suggestions. This work was supported in part by the Defense Advanced Research Projects Agency, under a Computer Science Study Group grant, as well as by the National Science Foundation (NSF) under CAREER award number 0953100. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF or DOE.

References

[1] Khaled Ahmed and Klaus Schuegraf, "Transistor wars," *Spectrum, IEEE*, vol. 48, no. 11, pp. 50–66, 2011.

[2] Juan L Aragón, José González, and Antonio González, "Power-aware control speculation through selective throttling," in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*. IEEE, 2003, pp. 103–112.

[3] SI Association *et al.*, "International technology roadmap for semiconductors," *Semiconductor Industry Association, Tech. Rep*, 2010.

[4] Nikolaos Bellas, Ibrahim Hajj, Constantine Polychronopoulos, and George Stamoulis, "Energy and performance improvements in microprocessor design using a loop cache," in *Computer Design, 1999.(ICCD'99) International Conference on*. IEEE, 1999, pp. 378–383.

[5] R. Bertran, M. Gonzàlez, X. Martorell, N. Navarro, and E. Ayguadé, "Counter-based power modeling methods: Top-down vs. bottom-up," *The Computer Journal*, 2012.

[6] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2. ACM, 2000, pp. 83–94.

[7] Eric S Chung, Peter A Milder, James C Hoe, and Ken Mai, "Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?" in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2010, pp. 225–236.

[8] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 2011, pp. 365–376.

[9] Hadi Esmaeilzadeh, Ting Cao, Xi Yang, Stephen Blackburn, and Kathryn McKinley, "What is happening to power, performance, and software?" *Micro, IEEE*, vol. 32, no. 3, pp. 110–121, 2012.

[10] W. Feng, X. Feng, and R. Ce, "Green supercomputing comes of age," *IT professional*, vol. 10, no. 1, pp. 17–23, 2008.

[11] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo, "Intel avx: New frontiers in performance improvements and energy efficiency," *Intel white paper*, 2008.

[12] Ricardo Gonzalez and Mark Horowitz, "Energy dissipation in general purpose microprocessors," *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 9, pp. 1277–1284, 1996.

[13] Ed Grochowski, Ronny Ronen, John Shen, and Hong Wang, "Best of both latency and throughput," in *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*. IEEE, 2004, pp. 236–243.

[14] Z. Herczeg, Á. Kiss, D. Schmidt, N. Wehn, and T. Gyimóthy, "Xeemu: An improved xscale power simulator," *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, pp. 300–309, 2007.

[15] Intel, "Intel composer xe 2013," 2013.

[16] *Intel® Architecture Code Analyzer*, Intel Corp., June 2012. Available: http://software.intel.com/en-us/articles/intel-architecture-code-analyzer

[17] *Intel® 64 and IA-32 Architectures Software Developer's Manuals*, Intel Corp., September 2013. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[18] A.B. Kahng, B. Li, L.S. Peh, and K. Samadi, "Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration," in *Proceedings of the conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2009, pp. 423–428.

[19] Jonathan G Koomey, Stephen Berard, Marla Sanchez, and Henry Wong, "Implications of historical trends in the electrical efficiency of computing," *Annals of the History of Computing, IEEE*, vol. 33, no. 3, pp. 46–54, 2011.

[20] Lea Hwang Lee, Bill Moyer, and John Arends, "Instruction fetch energy reduction using loop caches for embedded applications with small tight loops," in *Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on*. IEEE, 1999, pp. 267–269.

[21] A.W. Lewis, N.F. Tzeng, and S. Ghosh, "Runtime energy consumption estimation for server workloads based on chaotic time-series approximation," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 3, p. 15, 2012.

[22] Gabriel Loh, "The cost of uncore in throughput-oriented many-core processors," in *Workshop on Architectures and Languages for Throughput Applications*. Citeseer, 2008, pp. 1–9.

[23] J.C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A.C. Snoeren, and R.K. Gupta, "Evaluating the effectiveness of model-based power characterization," in *USENIX Annual Technical Conf*, 2011.

[24] Frank H McMahon, "The livermore fortran kernels: A computer test of the numerical performance range," Lawrence Livermore National Lab., CA (USA), Tech. Rep., 1986.

[25] Ofer Shacham, Omid Azizi, Megan Wachs, Wajahat Qadeer, Zain Asgar, Kyle Kelley, John P Stevenson, Stephen Richardson, Mark Horowitz, Benjamin Lee *et al.*, "Rethinking digital design: Why design must change," *Micro, IEEE*, vol. 30, no. 6, pp. 9–24, 2010.

[26] Ronak Singhal, "Inside intel next generation nehalem microarchitecture," in *Hot Chips*, vol. 20, 2008.

[27] S. Thoziyoor, N. Muralimanohar, J.H. Ahn, and N.P. Jouppi, "Cacti 5.1," *HP Laboratories, April*, vol. 2, 2008.