Looking Back on the Language and Hardware Revolutions: Measured Power, Performance, and Scaling *

Hadi Esmaeilzadeh[†] Ting Cao[‡] Xi Yang[‡] Stephen M. Blackburn[‡] Kathryn S. McKinley[§]

[†]University of Washington [‡]Australian National University [§]The University of Texas at Austin

hadianeh@cs.washington.edu {Ting.Cao,Xi.Yang,Steve.Blackburn}@anu.edu.au mckinley@cs.utexas.edu

Abstract

This paper reports and analyzes *measured* chip power and performance on five process technology generations executing 61 diverse benchmarks with a rigorous methodology. We measure representative Intel IA32 processors with technologies ranging from 130nm to 32nm while they execute sequential and parallel benchmarks written in native and managed languages. During this period, hardware and software changed substantially: (1) hardware vendors delivered chip multiprocessors instead of uniprocessors, and independently (2) software developers increasingly chose managed languages instead of native languages. This quantitative data reveals the extent of some known and previously unobserved hardware and software trends. Two themes emerge.

(I) *Workload:* The power, performance, and energy trends of native workloads do not approximate managed workloads. For example, (a) the SPEC CPU2006 native benchmarks on the i7 (45) and i5 (32) draw significantly less power than managed or scalable native benchmarks; and (b) managed runtimes exploit parallelism even when running single-threaded applications. The results recommend architects always include native and managed workloads when designing and evaluating energy efficient hardware.

(II) *Architecture:* Clock scaling, microarchitecture, simultaneous multithreading, and chip multiprocessors each elicit a huge variety of power, performance, and energy responses. This variety and the difficulty of obtaining power measurements recommends exposing on-chip power meters and when possible structure specific power meters for cores, caches, and other structures. Just as hardware event counters provide a quantitative grounding for performance innovations, power meters are necessary for optimizing energy.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Native & Managed Languages; Performance; Power

General Terms Experimentation, Languages, Performance, Power, Measurement

* This work is supported by ARC DP0666059 and NSF CSR-0917191. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.

Copyright © 2011 ACM 978-1-4503-0266-1/11/03…$10.00

1. Introduction

Quantitative performance analysis serves as a foundation for computer system design and innovation. In their now classic paper [9, 10], Emer and Clark noted that "A lack of detailed timing information impairs efforts to improve performance." They pioneered the quantitative approach by characterizing the instruction mix and cycles per instruction on real timesharing workloads. In fact, they surprised their reviewers by demonstrating the VAX-11/780 was a .5 MIPS machine, not a 1 MIPS machine! Using this data, their team, academics, and other industrial architects subsequently used a more principled approach to improving performance, or in other words [10]: "Boy, you ruin all our fun—you have data." This paper extends the quantitative approach to measured power on modern workloads. This work is timely because the past decade heralded the era of power and energy (power × execution time) constrained computer architecture design. A lack of detailed power measurements is impairing efforts to reduce energy consumption on real modern workloads.

From 2003 to 2010, technology shrank from 130nm to 32nm, following Moore's law. However, physical on-chip power limits and design complexity forced architects to stop using clock scaling as the primary means of improving performance. In this period, Dennard scaling slowed significantly [6]; reductions in process technology are no longer yielding both power and performance gains at historical rates; and wire delay hit its physical limit. For example, single-cycle cross-chip access times are not possible with typical chip areas. Because of technology constraints, computing entered the Chip MultiProcessor (CMP) era, in which architects are delivering more processors on each chip. On a broader scale, explosive demand for power and energy efficient large-scale computing [13] and mobile devices continues unabated. Consequently, power is a first-order design constraint in all market segments.

A commensurate explosion of software applications make these devices useful. Demands such as complexity management, reliability, correctness, security, mobility, portability, and ubiquity have pushed developers in many market segments away from native compiled ahead-of-time programming languages. Developers are increasingly choosing *managed* programming languages, which provide safe pointer disciplines, garbage collection (automatic memory management), extensive standard libraries, and portability through dynamic just-in-time compilation. For example, web services are increasingly using PHP on the server side and JavaScript on the client side. Java and its virtual machine are now mature technologies and are the dominant choice in markets as diverse as financial software and cell phone applications. This software trend, which is independent of the CMP hardware trend, motivates including managed workloads in architecture analysis.

This paper examines power, performance, and scaling in this period of software and hardware changes. We use eight representa-

tive Intel IA32 processors from five technology generations introduced in 2003 through 2010 that range from 130nm to 32nm. These processors have an isolated processor power supply on the motherboard with a stable voltage. We use a Hall effect sensor to measure the power supply current, and hence processor power. We calibrate and validate our sensor data. We find that actual power measurements vary widely with benchmarks. Furthermore, relative performance, power, and energy are not well predicted by core count, clock speed, or reported Thermal Design Power (TDP), i.e., the nominal amount of power the chip is designed to dissipate without exceeding the maximum junction temperature.

We execute 61 diverse sequential and parallel benchmarks written in native C, C++, and Fortran languages and in the managed Java language. We draw these benchmarks from SPEC CINT2006, SPEC CFP2006, PARSEC, SPECjvm, DaCapo 06-10-MR2, DaCapo 9.12, and pjbb2005. We use Java as our representative of managed languages because it is the most widely used managed language [33], has publicly available real-world sophisticated benchmarks, and has mature Virtual Machine (VM) technology. We classify the benchmarks into four groups and weight the groups equally: native non-scalable benchmarks (Native Non-scalable), native scalable benchmarks (Native Scalable), Java non-scalable benchmarks (Java Non-scalable), and Java scalable benchmarks (Java Scalable).

We explore the energy impact of a variety of hardware features using hardware configuration to perform controlled experiments. We explore the effects of simultaneous multithreading (SMT), core count (CMP), clock frequency, die shrink, gross microarchitectural changes, Turbo Boost,[1] software parallelism, and workload. We perform a historical analysis and a Pareto energy efficiency analysis, which identifies the most power and performance efficient designs in this architecture configuration space of real processors. We make all our data publicly available to encourage others to use it and perform further analysis. The individual benchmark results, experiments, and analysis described in this paper are in the ACM Digital Library in csv and excel formats as a companion to this paper [12]. To the best of our knowledge, this study is the first systematic exploration of power and performance across technology generations using measured processor power, controlled configuration, and diverse workloads.

This quantitative data reveals the extent, with precision and depth, of some known workload and hardware trends and some previously unobserved trends. We call out four workload and nine architecture findings from our data and analysis. Two themes emerge.

Workload Findings *The power, performance, and energy trends of native workloads do not approximate managed workloads well.* For example, (a) the SPEC CPU2006 native benchmarks executing on the i7 (45) and i5 (32) are outliers with respect to power because they draw significantly less power than managed or scalable native benchmarks; (b) the addition of SMT *slows down* non-scalable Java on the Pentium 4 (130), and (c) managed runtimes exploit parallelism, even when managed applications do not. For example, single-threaded Java workloads run on average about 10% faster and up to 60% faster on two cores when compared to one core. This result is not due to better code from more aggressive just-in-time compilation on unutilized cores. This speedup comes directly from parallelism in the VM and reductions in VM and application interference when the VM performs its computation and data accesses elsewhere. Native single-threaded workloads *never* experience performance or energy improvements from CMPs or SMT, and sometimes consume a small amount of additional power on CMPs. While measuring and simulating managed workloads does require additional methodologies, prior work resolves them (see Sections 2 and 5). *These results recommend that architects always include native and managed workloads when designing and evaluating energy efficient designs.*

Architectural Findings *A huge variety of processor power, performance, and energy responses due to features such as clock scaling, microarchitecture, Turbo Boost, SMT, and CMP reveal a complex and poorly understood energy efficiency design space*. Consider these three sample findings. (a) Halving the clock rate of the i5 (32) increases its energy consumption around 4%, whereas it decreases the energy consumption of the i7 (45) and Core 2D (45) by around 60%, i.e., running the i5 (32) at its *peak* clock rate is as energy efficient as running it as its lowest, whereas running the i7 (45) and Core 2D (45) at their *lowest* clock rate is substantially more energy efficient than their peak. (b) Two pairs of our processors observe the effect of a die-shrink. On the Core and Nehalem families, the die shrink is remarkably effective at reducing energy consumption, *even when controlling for clock speed*. (c) We disable and enable SMT and find that on more modern processors it is a remarkably energy efficient mechanism for exploiting software parallelism. Although it was originally designed for wide-issue out-of-order processors, SMT provides the most energy benefits for the dual-issue in-order Atom (45). Modern processors include power management techniques that monitor power sensors to minimize power usage and boost performance, for example. However, these sensors are not currently exposed. *The wide variety of performance and power responses to workload and architectural features, and the difficulty of obtaining power measurements recommends exposing on-chip power meters and when possible, structure-specific power meters for cores, caches, and other structures. Coupling these measurements with hardware event performance counters will provide a quantitative basis for optimizing power and energy for future system design.*

Measurement is key to understanding and optimization.

2. Methodology

This section describes our benchmarks, compilers, Java Virtual Machines, operating system, hardware, power measurement methodologies, and performance measurement methodologies.

2.1 Benchmarks

The following methodological choices in part prescribe our choice of benchmarks. (1) *Individual benchmark performance and average power:* We measure execution time and average power of individual benchmarks in isolation and aggregate them by workload type. While multi-programmed workload measurements, such as SPECrate can be valuable, the methodological and analysis challenges they raise are beyond the scope of this paper. (2) *Language and parallelism:* We systematically explore native / managed, and scalable / non-scalable workloads. We create four benchmark groups in the cross product and weight each group equally.

- **Native Non-scalable**: C, C++ and Fortran single-threaded benchmarks from SPEC CPU2006.
- **Native Scalable**: Multithreaded C and C++ benchmarks from PARSEC.
- **Java Non-scalable**: Single and multithreaded benchmarks that do not scale well from SPECjvm, DaCapo 06-10-MR2, DaCapo 9.12, and pjbb2005.
- **Java Scalable**: Multithreaded benchmarks from DaCapo 9.12, selected because their performance scales similarly to Native Scalable on the i7 (45).

Native and managed applications embody different tradeoffs between performance, reliability, portability, and deployment. In this

[1] Intel Turbo Boost technology automatically increases frequency beyond the default frequency when the chip is operating below power, current, and temperature thresholds [19].

Grp	Src	Name	Time	Description
Native Non-scalable	SI	perlbench	1037	Perl programming language
		bzip2	1563	bzip2 Compression
		gcc	851	C optimizing compiler
		mcf	894	Combinatorial opt/singledepot vehicle scheduling
		gobmk	1113	AI: Go game
		hmmer	1024	Search a gene sequence database
		sjeng	1315	AI: tree search & pattern recognition
		libquantum	629	Physics / Quantum Computing
		h264ref	1533	H.264/AVC video compression
		omnetpp	905	Ethernet network simulation based on OMNeT++
		astar	1154	Portable 2D path-finding library
		xalancbmk	787	XSLT processor for transforming XML
	SF	gamess	3505	Quantum chemical computations
		milc	640	Physics/quantum chromodynamics (QCD)
		zeusmp	1541	Physics/Magnetohydrodynamics based on ZEUS-MP
		gromacs	983	Molecular dynamics simulation
		cactusADM	1994	Cactus and BenchADM physics/relativity kernels
		leslie3d	1512	Linear-Eddy Model in 3D computational fluid dynamics
		namd	1225	Parallel simulation of large biomolecular systems
		dealII	832	PDEs with adaptive finite element method
		soplex	1024	Simplex linear program solver
		povray	636	Ray-tracer
		calculix	1130	Finite element code for linear and nonlinear 3D structural applications
		GemsFDTD	1648	Solves the Maxwell equations in 3D in the time domain
		tonto	1439	Quantum crystallography
		lbm	1298	Lattice Boltzmann Method for incompressible fluids
		sphinx3	2007	Speech recognition
Native Scalable	PA	blackscholes	482	Prices options with Black-Scholes PDE
		bodytrack	471	Tracks a markerless human body
		canneal	301	Minimizes the routing cost of a chip design with cache-aware simulated annealing
		facesim	1230	Simulates human face motions
		ferret	738	Image search
		fluidanimate	812	Fluid motion physics for realtime animation with SPH algorithm
		raytrace	1970	Uses physical simulation for visualization
		streamcluster	629	Computes an approximation for the optimal clustering of a stream of data points
		swaptions	612	Prices a portfolio of swaptions with the Heath-Jarrow-Morton framework
		vips	297	Applies transformations to an image
		x264	265	MPEG-4 AVC / H.264 video encoder
Java Non-scalable	SJ	compress	5.3	Lempel-Ziv compression
		jess	1.4	Java expert system shell
		db	6.8	Small data management program
		javac	3.0	The JDK 1.0.2 Java compiler
		mpegaudio	3.1	MPEG-3 audio stream decoder
		mtrt	0.8	Dual-threaded raytracer
		jack	2.4	Parser generator with lexical analysis
	D6	antlr	2.9	Parser and translator generator
		bloat	7.6	Java bytecode optimization and analysis tool
	D9	avrora	11.3	Simulates the AVR microcontroller
		batik	4.0	Scalable Vector Graphics (SVG) toolkit
		fop	1.8	Output-independent print formatter
		h2	14.4	An SQL relational database engine in Java
		jython	8.5	Python interpreter in Java
		pmd	6.9	Source code analyzer for Java
		tradebeans	18.4	Tradebeans Daytrader benchmark
		luindex	2.4	A text indexing tool
	JB	pjbb2005	10.6	Transaction processing, based on SPECjbb2005
Java Scalable	D9	eclipse	50.5	Integrated development environment
		lusearch	7.9	Text search tool
		sunflow	19.4	Photo-realistic rendering system
		tomcat	8.6	Tomcat servlet container
		xalan	6.9	XSLT processor for XML documents

Table 1. Benchmark Groups; Source: SI: SPEC CINT2006, SF: SPEC CFP2006, PA: PARSEC, SJ: SPECjvm, D6: DaCapo 06-10-MR2, D9: DaCapo 9.12, and JB: pjbb2005; Source, Name, and reference run times in seconds.

	Execution Time		Power	
	average	max	average	max
Average	1.2%	2.2%	1.5%	7.1%
Native Non-scalable	0.9%	2.6%	2.1%	13.9%
Native Scalable	0.7%	4.0%	0.6%	2.5%
Java Non-scalable	1.6%	2.8%	1.5%	7.7%
Java Scalable	1.8%	3.7%	1.7%	7.9%

Table 2. Aggregate 95% confidence intervals for measured execution time and power, showing average and maximum error across all processor configurations, and all benchmarks.

setting, it is impossible to meaningfully separate language from workload. We therefore offer no commentary on the *virtue* of a language choice, but rather, reflect the measured reality of two workload classes that are ubiquitous in today's software landscape.

We draw 61 benchmarks from six suites to populate these groups. We weight each group equally in our aggregate measurements; see Section 2.6 for more details on aggregation. We use Java to represent the broader class of managed languages. Table 1 shows the benchmarks, their groupings, the suite of origin, the *reference running time* (see Section 2.6) to which we normalize our results, and a short description. In the case of native benchmarks, all single-threaded benchmarks are non-scalable and all parallel multithreaded native benchmarks are scalable on up to eight hardware contexts, the maximum we explore. By scalable, we mean that adding hardware contexts improves performance. Bienia et al. show that the PARSEC benchmarks scale up to 8 hardware contexts [2]. To create a comparable group of scalable Java programs, we put multithreaded Java programs that do not scale well in the non-scalable category.

Native Non-scalable Benchmarks We use 27 C, C++, and Fortran codes from the SPEC CPU2006 suite [31] for Native Non-scalable and all are *single-threaded*. The 12 SPEC CINT benchmarks represent compute-intensive integer applications that contain sophisticated control flow logic, and the 15 SPEC CFP benchmarks represent compute-intensive floating-point applications. These native benchmarks are compiled ahead-of-time. We chose Intel's *icc* compiler because we found that it consistently generated better performing code than *gcc*. We compiled all of the Native Non-scalable benchmarks with version 11.1 of the 32-bit Intel compiler suite using the `-o3` optimization flag, which performs aggressive scalar optimizations. This flag does not include any automatic parallelization. We compiled each benchmark once, using the default Intel compiler configuration, without setting any microarchitecture-specific optimizations, and used the *same* binary on all platforms. We exclude 410.bwaves and 481.wrf because they failed to execute when compiled with the Intel compiler. Three executions are prescribed by SPEC. We report the mean of these three successive executions. Table 2 shows that aggregate 95% confidence intervals are low for execution time and power: 1.2% and 1.5% respectively.

Native Scalable Benchmarks The Native Scalable benchmarks consists of 11 C and C++ benchmarks from the PARSEC suite [2]. The benchmarks are intended to be diverse and forward looking parallel algorithms. All but one uses POSIX threads and one contains some assembly code. We exclude freqmine because it is not amenable to our scaling experiments, in part, because it does not use POSIX threads. We exclude dedup from our study because it has a large working set that exceeds the amount of memory available on the 2003 Pentium 4 (130). The multithreaded PARSEC benchmarks include *gcc* compiler configurations, which worked correctly. The icc compiler failed to produce correct code for many of the PARSEC benchmarks with similar configurations. We used the PARSEC default gcc build scripts with gcc version 4.4.1. The gcc scripts use `-O3` optimization. We leave systematic comparisons using both

icc and gcc to future work. We report the mean of five successive executions of each benchmark. We use five executions, which as Table 2 shows, gives low aggregate 95% confidence intervals for execution time and power: 0.9% and 2.1% on average.

Java Non-scalable Benchmarks The Java Non-scalable group includes benchmarks from SPECjvm, both releases of DaCapo, and pjbb2005 that do not scale well. It includes both single-threaded and multithreaded benchmarks. SPECjvm is intended to be representative of client-side Java programs. Although the SPECjvm benchmarks are over ten years old and Blackburn et al. have shown that they are simple and have a very small instruction cache and data footprint [4], many researchers still use them. The DaCapo Java benchmarks are intended to be diverse, forward-looking, and non-trivial [4, 32]. The benchmarks come from major open source projects under active development. Researchers have not reported extensively on the 2009 release, but it was designed to expose richer behavior and concurrency on large working sets. We exclude tradesoap because its heavy use of sockets suffered from timeouts on the slowest machines. We use pjbb2005, which is a fixed-workload variant of SPECjbb2005 [30] that holds the workload, instead of time, constant. We configure pjbb2005 with 8 warehouses and 10,000 transactions per warehouse. We include the following multithreaded benchmarks in Java Non-scalable: pjbb2005, avrora, batik, fop, h2, jython, pmd, and tradebeans. As we show below, these applications do not scale well. Section 2.2 discusses the measurement methodology for Java. Table 2 indicates low aggregate 95% confidence intervals for execution time and power: 1.6% and 1.5%.

Java Scalable Benchmarks We include the multithreaded Java benchmarks in Java Scalable that scale similarly to the Native Scalable benchmarks. Figure 1 shows the scalability of the multithreaded Java benchmarks. The five most scalable are: sunflow, xalan, tomcat, lusearch and eclipse, all from DaCapo 9.12. Together, they speed up on average by a factor of 3.4 given eight hardware contexts compared to one context on the i7 (45). Our Native Scalable benchmarks scale better on this hardware, improving by a factor of 3.8. Although eliminating lusearch and eclipse would improve average scalability, it would reduce the number of benchmarks to three, which we believe is insufficient. Table 2 shows low aggregate 95% confidence intervals for execution time and power: 1.8% and 1.7%.

2.2 Java Virtual Machines and Measurement Methodology

We use Java as the managed language representative in part because of its mature Java Virtual Machine (JVM) technology, which includes high performance garbage collection, profiling, and dynamic optimizations. We report Oracle (Sun) HotSpot build 16.3-b01 Java 1.6.0 Virtual Machine. We did some additional experiments with Oracle JRockit build R28.0.0-679-130297 and IBM J9 build pxi3260sr8. Their average performance is similar to HotSpot, but individual benchmarks vary substantially. We observe aggregate power differences of up to 10% between JVMs [12]. Exploring the influence of the native compilers and JVMs on power and energy is an interesting avenue for future research.

To measure both Java Non-scalable and Java Scalable, we follow the recommended methodologies for measuring Java [5, 14]. We use the `-server` flag and fix the heap size at a generous 3× the minimum required for each benchmark. We did not set any other JVM flags. We report the fifth iteration of each benchmark within a single invocation of the JVM to capture steady state behavior. This methodology avoids class loading and heavy compilation activity that often dominates the early phases of execution. The fifth iteration may still have a small amount of compiler activity, but has sufficient time to create optimized frequently executed code. We perform this process twenty times and report the mean. Table 2 reports the measured error. We require twenty invocations to generate a statistically stable result because the adaptive JIT compilation and garbage collection induce non-determinism. In contrast to the compiled ahead-of-time native configurations, Java compilers may dynamically produce microarchitecture-specific code.

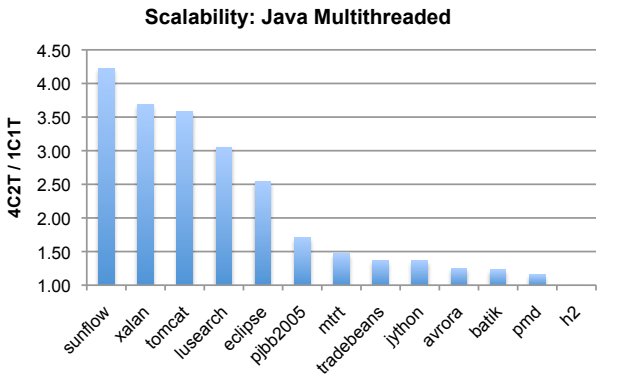


Figure 1. Scalability of Java multithreaded benchmarks on i7 (45).

2.3 Operating System

We perform all the experiments using 32-bit Ubuntu 9.10 Karmic with the 2.6.31 Linux kernel. We use a 32-bit OS and compiler builds because the 2003 Pentium 4 (130) does not support 64-bit. Exploring the impact of word size is also interesting future work.

2.4 Hardware Platforms

We use eight IA32 processors, manufactured by Intel using four process technologies (130nm, 65nm, 45nm, and 32nm), representing four microarchitectures (NetBurst, Core, Bonnell, and Nehalem). Table 3 lists processor characteristics: uniquely identifying sSpec number, release date / price; CMP and SMT (*n*C*m*T means *n* cores, *m* SMT threads per core), die characteristics; and memory configuration. Intel sells a large range of processors for each microarchitecture—the processors we use are just samples within that space. Most of our processors are mid-range desktop processors. The release date and release price in Table 3 provides the context regarding Intel's placement of each processor in the market. The two Atoms and the Core 2Q (65) *Kentsfield* are extreme points at the bottom and top of the market respectively.

2.5 Power Measurements

In contrast to whole system power studies [3, 20, 22], we measure on-chip power. Whole system studies measure AC current to an entire computer, typically with a clamp ammeter. To measure on-chip power, we must isolate and measure DC current to the processor on the motherboard, which cannot be done with a clamp ammeter. We use *Pololu's ACS714* current sensor board, following prior methodology [26]. The board is a carrier for *Allegro's* $\pm 5A$ *ACS714* Hall effect-based linear current sensor. The sensor accepts a bidirectional current input with a magnitude up to $5A$. The output is an analog voltage ($185mV/A$) centered at $2.5V$ with a typical error of less than 1.5%. The sensor on i7 (45), which has the highest power consumption, accepts currents with magnitudes up to $30A$.

Each of our experimental machines has an isolated power supply for the processor on the motherboard, which we verified by examining the motherboard specification and confirmed empirically. This requirement precludes measuring for example the Pentium M, which would have given us a 90nm processor. We place the sensors on the $12V$ power line that supplies only the processor. We experimentally measured voltage and found it was very stable, varying less than 1%. We send the measured values from the current sen-

Processor	µArch	Processor	sSpec	Release Date	Price USD	CMP SMT	LLC B	Clock GHz	nm	Trans M	Die mm^2	VID Range V	TDP (W)	FSB MHz	B/W GB/s	DRAM Model
Pentium 4	NetBurst	Northwood	SL6WF	May '03	—	1C2T	512K	2.4	130	55	131	—	66	800	—	DDR-400
Core 2 Duo E6600	Core	Conroe	SL9S8	Jul '06	$316	2C1T	4M	2.4	65	291	143	0.85 - 1.50	65	1066	—	DDR2-800
Core 2 Quad Q6600	Core	Kentsfield	SL9UM	Jan '07	$851	4C1T	8M	2.4	65	582	286	0.85 - 1.50	105	1066	—	DDR2-800
Core i7 920	Nehalem	Bloomfield	SLBCH	Nov '08	$284	4C2T	8M	2.7	45	731	263	0.80 - 1.38	130	—	25.6	DDR3-1066
Atom 230	Bonnell	Diamondville	SLB6Z	Jun '08	$29	1C2T	512K	1.7	45	47	26	0.90 - 1.16	4	533	—	DDR2-800
Core 2 Duo E7600	Core	Wolfdale	SLGTD	May '09	$133	2C1T	3M	3.1	45	228	82	0.85 - 1.36	65	1066	—	DDR2-800
Atom D510	Bonnell	Pineview	SLBLA	Dec '09	$63	2C2T	1M	1.7	45	176	87	0.80 - 1.17	13	665	—	DDR2-800
Core i5 670	Nehalem	Clarkdale	SLBLT	Jan '10	$284	2C2T	4M	3.4	32	382	81	0.65 - 1.40	73	—	21.0	DDR3-1333

Table 3. The eight experimental processors and key specifications.

	Speedup Over Reference								Power (W)							
Processor	NN	NS	JN	JS	Avg^w	Avg^b	Min	Max	NN	NS	JN	JS	Avg^w	Avg^b	Min	Max
Pentium 4	0.91 *6*	0.79 *7*	0.80 *6*	0.75 *7*	0.82 *6*	0.85 *6*	0.51 *6*	1.25 *6*	42.1 *7*	43.5 *6*	45.1 *7*	45.7 *6*	44.1 *6*	43.5 *7*	34.5 *7*	50.0 *6*
Core 2 Duo E6600	2.02 *5*	2.10 *5*	1.99 *5*	2.04 *5*	2.04 *5*	2.03 *5*	1.40 *4*	2.85 *5*	24.3 *5*	26.6 *4*	26.2 *5*	28.5 *4*	26.4 *5*	25.6 *5*	21.4 *5*	32.3 *4*
Core 2 Quad Q6600	2.04 *4*	3.62 *3*	2.04 *4*	3.09 *3*	2.70 *3*	2.41 *4*	1.39 *5*	4.67 *3*	50.7 *8*	61.7 *8*	55.3 *8*	64.6 *8*	58.1 *8*	55.2 *8*	45.6 *8*	77.3 *7*
Core i7 920	3.11 *2*	6.25 *1*	3.00 *2*	5.49 *1*	4.46 *1*	3.84 *1*	2.16 *2*	7.60 *1*	27.2 *6*	60.4 *7*	37.5 *6*	62.8 *7*	47.0 *7*	39.1 *6*	23.4 *6*	89.2 *8*
Atom 230	0.49 *8*	0.52 *8*	0.53 *8*	0.52 *8*	0.52 *8*	0.51 *8*	0.39 *8*	0.75 *8*	2.3 *1*	2.5 *1*	2.3 *1*	2.4 *1*	2.4 *1*	2.3 *1*	1.9 *1*	2.7 *1*
Core 2 Duo E7600	2.48 *3*	2.76 *4*	2.49 *3*	2.44 *4*	2.54 *4*	2.53 *3*	1.45 *3*	3.71 *4*	19.1 *3*	21.1 *3*	20.5 *3*	22.6 *3*	20.8 *3*	20.2 *3*	15.8 *3*	26.8 *3*
Atom D510	0.53 *7*	0.96 *6*	0.61 *7*	0.86 *6*	0.74 *7*	0.66 *7*	0.41 *7*	1.17 *7*	3.7 *2*	5.3 *2*	4.5 *2*	5.1 *2*	4.7 *2*	4.3 *2*	3.4 *2*	5.9 *2*
Core i5 670	3.31 *1*	4.46 *2*	3.18 *1*	4.26 *2*	3.80 *2*	3.56 *2*	2.39 *1*	5.42 *2*	19.6 *4*	29.2 *5*	24.7 *4*	29.5 *5*	25.7 *4*	23.6 *4*	16.5 *4*	38.2 *5*

Table 4. Average performance and power characteristics. The rank for each measure is indicated in small italics.

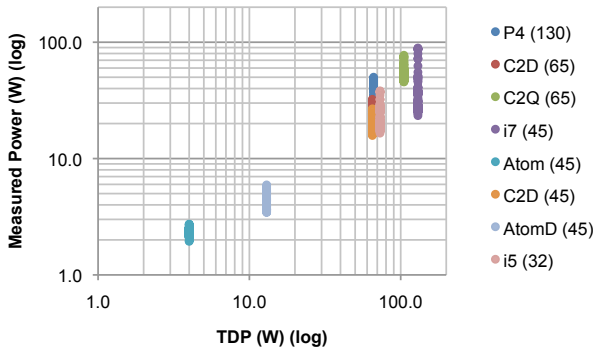


Figure 2. Measured benchmark power for each processor.

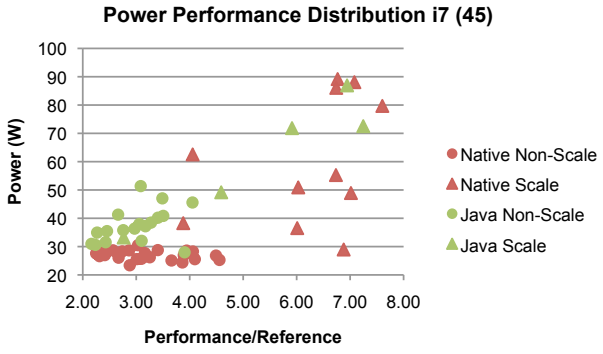


Figure 3. Benchmark power and performance on the i7 (45).

sor to the measured machine's USB port using *Sparkfun's Atmel AVR Stick*, which is a simple data-logging device. We use a data-sampling rate of 50Hz. We execute each benchmark, log its measured power values, and then compute the average power consumption over the duration of the benchmark.

To calibrate the meters, we use a current source to provide 28 reference currents between 300mA and 3A, and for each meter record the output value (an integer in the range 400-503). We compute linear fits for each of the sensors. Each sensor has an R^2 value of 0.999 or better, which indicates an excellent fit [12]. The measurement error for any given sample is about 1%, which reflects the fidelity of the quantization (103 points).

Thermal Design Power (TDP) TDP is the nominal thermal design power for a processor, i.e., the nominal amount of power the chip is designed to dissipate without exceeding the maximum transistor junction temperature. Table 3 indicates the TDP for each of our processors. Because measuring real processor power is difficult and TDP is readily available, TDP is often substituted for real measured power. Figure 2 shows that this substitution is problematic. It plots measured power for each benchmark on each stock processor as a function of TDP, on a logarithmic scale. Note that for these benchmarks TDP is strictly higher than actual power, and that measured power varies greatly among the benchmarks. This variation is highest on the i7 (45) and i5 (32), which likely reflects the advanced power management integrated into these processors [29]. For example, on the i7 (45), measured power varies between 23W for 471.omnetpp and 89W for fluidanimate! The smallest variation between maximum and minimum is on the Atom (45), but even this is around 30%. Manufacturers sometimes report the same TDP for a family of microarchitectures. For example, Core 2D (65) and Core 2D (45) have the same TDP of 65W as shown in Table 3, yet their measured power differs by around 40-50% as shown in Figure 2. In summary, while TDP loosely correlates with power consumption, it *does not* provide a good estimate for: (1) maximum power consumption of individual processors; (2) comparing among processors; or (3) approximating benchmark-specific power consumption.

2.6 Reference Execution Time, Reference Energy, and Aggregation

As is standard, we weight each benchmark equally within each workload group, since the execution time of the benchmark is not necessarily an indicator of benchmark importance. Furthermore, we want to represent each of our benchmark groups equally. These goals require (1) a reference execution time and a reference energy value for normalization, and (2) an average of the benchmarks in

each workload group. Since the average power of a benchmark is not directly biased by execution time, we use it directly. We also normalize energy to a reference, since $energy = power \times time$.

Table 1 shows the reference running time we use to normalize the execution time and energy results. To avoid biasing performance measurements to the strengths or weaknesses of one architecture, we normalize individual benchmark execution times to its average execution time executing on four architectures. We choose the Pentium 4 (130), Core 2D (65), Atom (45), and i5 (32) to capture all four microarchitectures and all four technology generations in this study. The reference energy is the average power on these four processors times the average runtime. Given a power and time measurement, we compute energy and then normalize it to the reference energy.

Table 1 shows that the native workloads tend to execute for much longer than the managed workloads. Measuring their code bases is complicated because of the heavy use of libraries by the managed languages and by PARSEC. However, some native benchmarks are tiny and many PARSEC codes are fewer than 3000 lines of non-comment code. These estimates show that the size of the native and managed application code bases alone (excluding libraries) does not explain the longer execution times. There is no evidence that native execution times are due to more sophisticated applications; instead these longer execution times are likely due to more repetition.

The averages equally weight each of the four benchmark groups. We report results for each group by taking the arithmetic mean of the benchmarks within the group. We use the mean of the four groups for the overall average. This aggregation avoids bias due to the varying number of benchmarks within each group (from 5 to 27). Table 4 shows the measured performance and power for each of the processors and each of the benchmark groups. The table indicates the weighted average (Avg^w), which we use throughout the paper, and for comparison, the simple average of the benchmarks (Avg^b). The table also records the highest and lowest performance and power measures seen on each of the processors.

2.7 Benchmark Power / Performance Diversity

Figure 3 shows the range of power and performance among our benchmarks and among our workload groups. Native / managed is differentiated by color and scalable / non-scalable is differentiated by shape for each benchmark. The graph plots performance of the i7 (45) normalized to the reference performance on the x-axis, and power on the y-axis. Unsurprisingly, the scalable benchmarks perform the best and consume the most power, since the i7 (45) has eight hardware contexts. Non-scalable benchmarks however exhibit a wide range of performance and power characteristics as well. Overall, the benchmarks exhibit a diversity of power and performance characteristics.

2.8 Processor Configuration Methodology

We evaluate the eight stock processors and configure them for a total of 45 processor configurations. We produce power and performance data for each benchmark that corresponds to Figure 3 for each configuration [12]. To explore the influence of architectural features, we control for clock speed and hardware contexts. We selectively down-clock the processors, disable cores, disable simultaneous multithreading (SMT), and disable Turbo Boost [19]. Intel markets SMT as Hyper-Threading [18]. The stock configurations of Pentium 4 (130), Atom (45), Atom D (45), i7 (45), and i5 (32) include SMT (Table 3). The stock configurations of the i7 (45) and i5 (32) include Turbo Boost, which automatically increases frequency beyond the base operating frequency when the core is operating below power, current, and temperature thresholds [19]. We control each variable via the BIOS. We experimented with operating system configuration, which is far more convenient, but it was not sufficiently reliable. For example, operating system scaling of hardware contexts often caused power consumption to increase as hardware resources were decreased! Extensive investigation revealed a bug in the Linux kernel [23]. We use all the means at our disposal to isolate the effect of various architectural features using stock hardware, but often the precise semantics are undocumented. Notwithstanding such limitations, these processor configurations help quantitatively explore how a number of features influence power and performance in real processors.

3. Feature Analysis

We organize the results into two sections. This first section explores the energy impact of hardware features through controlled experiments. The second section explores historical trends and performs an energy and performance Pareto efficiency analysis at the 45nm technology node. We present two pairs of graphs for feature analysis experiments as shown in Figure 4 for example. The top graph compares relative power, performance, and energy as an average of the four workload groups. The bottom graph breaks down energy by workload group. In these graphs, higher is better for performance. Lower is better for power and energy. Because of the volume of data and analysis, we cannot present all our data and sometimes refer to data that we gathered, but do not present. In these cases, we cite the complete, on line data as appropriate [12]. We organize the analysis by calling out, labeling, and numbering summary points as ARCHITECTURE FINDINGS and WORKLOAD FINDINGS.

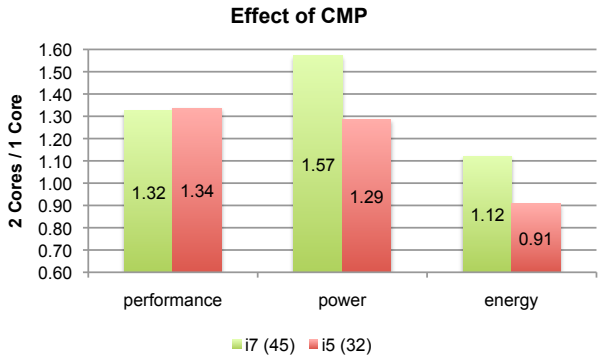
3.1 Chip Multiprocessors

We measure the effect of enabling one or two cores. We disable Simultaneous Multithreading (SMT) to maximally expose thread-level parallelism to the Chip MultiProcessor (CMP) hardware feature. We also disable Turbo Boost because its power and performance behavior is affected by the number of idle cores. (Section 3.6 explores Turbo Boost.) Figure 4(a) shows the average power, performance, and energy effects of moving from one core to two cores for the i7 (45) and i5 (32) processors. Figure 4(b) breaks down the energy effect as a function of benchmark group. While average energy is reduced by 9% on the i5 (32), it is increased by 12% on the i7 (45). Figure 4(a) shows that the source of this difference is that the i7 (45) experiences twice the power overhead for enabling a core as the i5 (32), while producing roughly the same performance improvement.

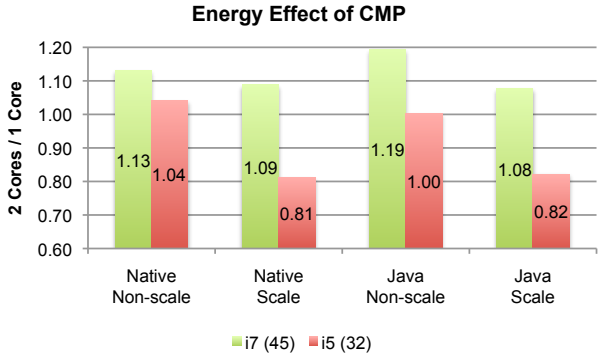
> ARCHITECTURE FINDING 1. *When comparing one core to two, enabling a core is not consistently energy efficient.*

Figure 4(b) shows that Native Non-scalable and Java Non-scalable suffer the most energy overhead with the addition of another core on the i7 (45). As expected, performance for Native Non-scalable is unaffected [12]. However, turning on an additional core for Native Non-scalable leads to a power increase of 4% and 14% respectively for the i5 (32) and i7 (45), translating to energy overheads.

More interesting is that Java Non-scalable does not incur energy overhead on the i5 (32). Careful examination reveals that the *single-threaded* Java Non-scalable experience performance gains from CMP on both processors. Figure 6 shows the scalability of the single-threaded subset of Java Non-scalable on the i7 (45), with SMT disabled, comparing one and two cores. We were very surprised that most of the single-threaded Java workloads exhibit measurable speedups with a second core. On the i5 (32), the Java speedups offset the power overhead of enabling additional cores. Although these Java benchmarks themselves are single-threaded, the JVMs on which they execute are not.

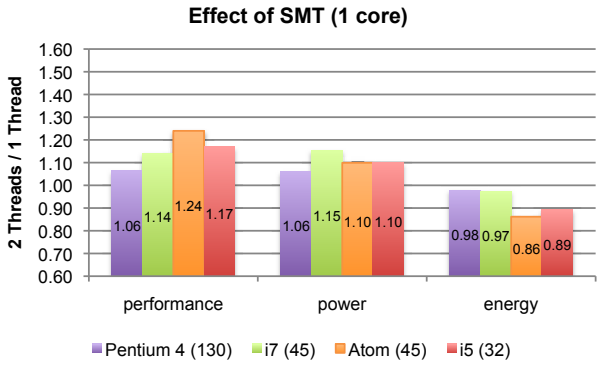


(a) Average impact of doubling cores.

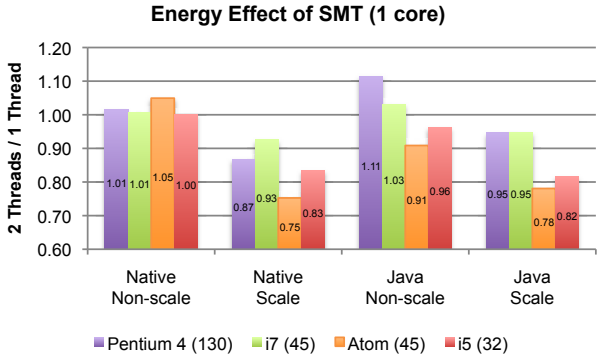


(b) Workload energy impact of doubling cores.

Figure 4. Comparing two cores to one *without* SMT or Turbo Boost.



(a) Average impact of two-way SMT.



(b) Workload energy impact of two-way SMT.

Figure 5. Two-way SMT impact with respect to a single core.

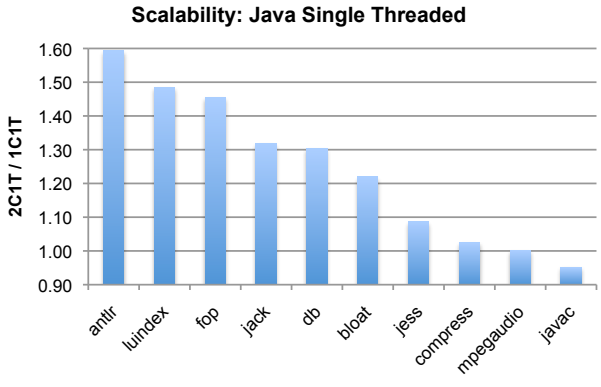


Figure 6. CMP impact for single-threaded Java.

WORKLOAD FINDING 1. *The JVM often induces significant amounts of parallelism into the execution of single-threaded Java benchmarks.*

Since the JVM's runtime services, such as profiling, just-in-time (JIT) compilation, and garbage collection, are concurrent and parallel, the JVM provides substantial scope for parallelization, even within these ostensibly sequential applications. To understand these effects better, we instrumented the HotSpot JVM to separately count cycles and retired instructions for the JVM and application. These experiments isolate the sources of parallelism from HotSpot to the JIT compiler and garbage collector.

Most benchmarks spend around 90-99% of their time in the application thread, but for example, antlr spends as much as 50% of its time in the JVM. Although db spends 95% of its instructions in single-threaded application code, it experiences a 30% improvement when it uses two hardware contexts on the i7 (45). Performance counter measurements show that memory system performance improvements were the cause of this surprising result. The DTLB experiences a factor of 2.5 fewer misses when more cores were available. Our hypothesis is that when the garbage collector executes on other cores it dramatically reduces the collector displacement effect on application data in local caches. These memory and cache behaviors are clearly significant but they are very subtle and need further exploration.

3.2 Simultaneous Multithreading

We measured the effect of simultaneous multithreading (SMT) [34] by disabling SMT at the BIOS on the Pentium 4 (130), Atom (45), i5 (32), and i7 (45). Each processor supports two-way SMT. On the i5 (32) and i7 (45) multiprocessors, we use only one core to ensure that SMT is the sole opportunity for thread-level parallelism. We disable Turbo Boost since it may vary the clock rate. Each processor is otherwise in its stock configuration. Figure 5(a) shows the overall power, performance, and energy impact of enabling SMT on a single core. Singhal states that the small amount of logic that is exclusive to SMT consumes very little power [29]. Nonetheless, this logic is integrated, so SMT does contribute to

total power even when disabled. These results therefore slightly underestimate the power cost of SMT. The performance advantage of SMT is significant. Notably, on the i5 (32) and Atom (45), SMT improves average performance significantly without much cost in power, leading to net energy savings.

ARCHITECTURE FINDING 2. *SMT delivers substantial energy savings for the i5 (32) and Atom (45).*

Given that SMT was motivated and continues to be motivated by the challenge of filling issue slots and hiding latency in wide issue superscalars [29, 34], it appears counter intuitive that performance on the dual-issue Atom (45) should benefit so much more from SMT than the quad-issue i7 (45) and i5 (32). One potential explanation is that the in-order Atom (45) is more restricted in its capacity to fill issue slots. Compared to other in-order processors, the Atom (45) has a relatively deep pipeline. Compared to the other processors in this study, the Atom (45) has much smaller caches. These features accentuate the need to hide latency, and therefore the value of SMT.

The performance improvements on the Pentium 4 (130) due to SMT are half to one third that of the newer processors, and consequently there is no net energy advantage. This result is not so surprising given that the Pentium 4 (130) is among the first commercial implementations of SMT. Furthermore, the i5 (32) and i7 (45) have more issue slots to fill and their much larger cache capacities and memory bandwidth better sustain the demands of SMT.

WORKLOAD FINDING 2. *On the Pentium 4 (130), SMT degrades performance for Java Non-scalable.*

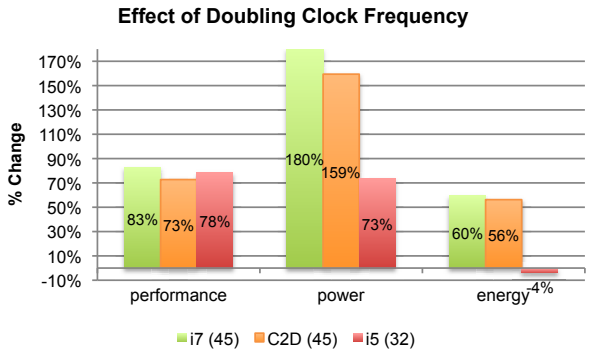
SMT is particularly unhelpful to Java on the Pentium 4 (130) as shown in Figure 5(b). It offers substantially lower performance improvements on Java Scalable and a performance *degradation* on Java Non-scalable [12], the latter leading to an 11% energy overhead. Figure 5(b) shows that, as expected, Native Non-scalable benchmarks experience very little energy overhead due to enabling SMT, whereas Figure 4(b) shows that enabling a core incurs a significant power and thus energy penalty. The scalable benchmarks unsurprisingly benefit most from SMT.

The effectiveness of SMT is impressive on recent processors as compared to CMP, particularly given its 'very low' die footprint [29]. Compare Figure 4 and 5. SMT provides less performance improvement than CMP—SMT adds about half as much performance as CMP on average, but incurs much less power cost—SMT adds just a quarter of the power of CMP on the i7 (45) and one third the power on the i5 (32). These factors lead to greater energy savings on the i7 (45) and i5 (32). These results on the modern processors show SMT in a much more favorable light than in Sasanka et al.'s model-based comparative study of the energy efficiency of SMT and CMP [28].

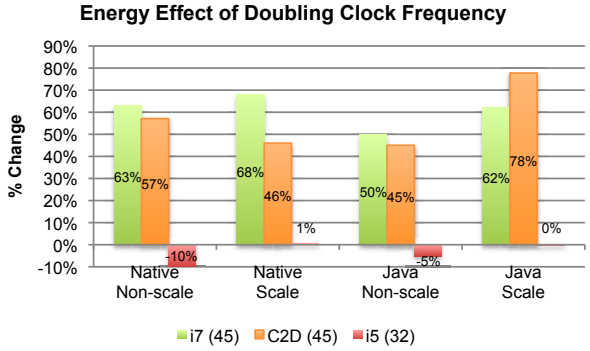
3.3 Clock Scaling

We vary the processor clock on the i7 (45), Core 2D (45), and i5 (32) between their minimum and maximum settings and measure the effect on power and performance. The range of clock speeds are: 1.6 to 2.7GHz for i7 (45); 1.6 to 3.1GHz for Core 2D (45); and 1.2 to 3.5GHz for i5 (32). We uniformly disable Turbo Boost to produce a consistent clock rate for comparison; Turbo Boost may vary the clock rate, but only when the clock is set at its highest value. Each processor is otherwise in its stock configuration. Figures 7(a) and 7(b) express changes in power, performance, and energy with respect to doubling in clock frequency over the range of clock speeds to normalize and compare across architectures.

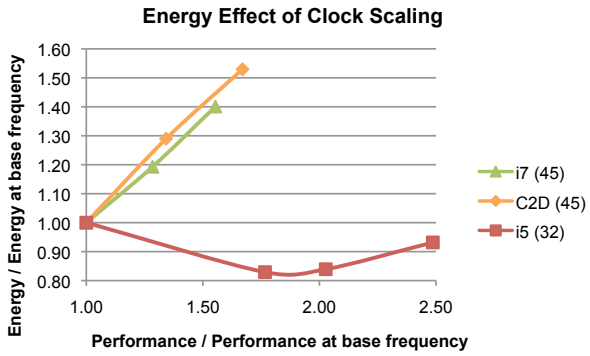
The three processors experience broadly similar increases in performance of around 80%, but *power differences vary substantially, from 70% to 180%*. On the i7 (45) and Core 2D (45), the performance increases require disproportional power increases—conse-



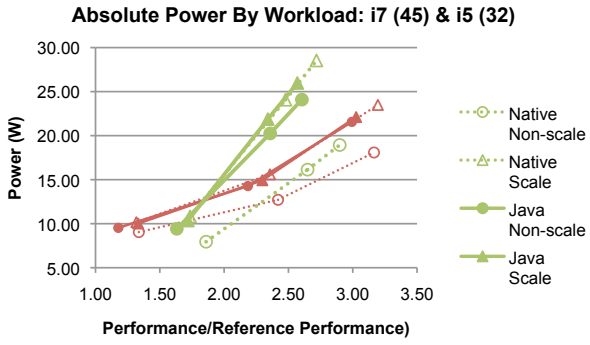
(a) Average impact of doubling clock.



(b) Workload energy impact of doubling clock.



(c) Average energy performance curve, each point is a clock speed.



(d) Absolute power (Watts) and performance on the i7 (45) (green) and i5 (32) (red) by benchmark group, each point is a clock speed.

Figure 7. The impact of clock scaling in stock configurations.

quently energy consumption increases by about 60% as the clock is doubled. The i5 (32) is starkly different—doubling its clock leads to a slight energy reduction.

> ARCHITECTURE FINDING 3. *The i5 (32) does not increase energy consumption as the clock increases, in contrast to the i7 (45) and Core 2D (45).*

Figure 7(c) shows that this result is consistent across the range of i5 (32) clock rates. A number of factors may explain why the i5 (32) performs relatively so much better at its highest clock rate: (a) the i5 (32) is a 32nm process, while the others are 45nm; (b) the power-performance curve is non-linear and these experiments may observe only the upper (steeper) portion of the curves for i7 (45) and Core 2D (45); (c) although the i5 (32) and i7 (45) share the same microarchitecture, the second generation i5 (32) likely incorporates energy improvements; (d) the i7 (45) is substantially larger than the other processors, with four cores and a larger cache.

> WORKLOAD FINDING 3. *The power / performance behavior of Native Non-scalable differs from the three other workload groups.*

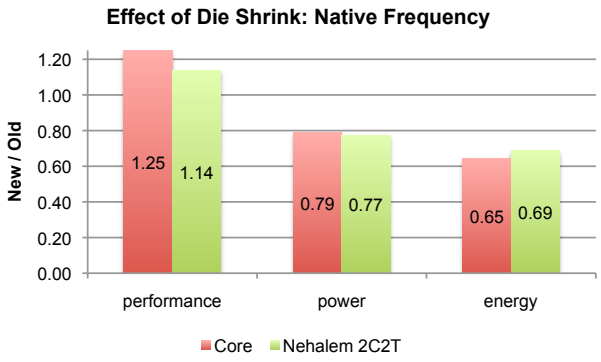
Figure 7(b) shows that doubling the clock on the i5 (32) roughly maintains or improves energy consumption of all benchmark groups, with Native Non-scalable improving the most. For the i7 (45) and Core 2D (45), doubling the clock raises energy consumption. Figure 7(d) shows that Native Non-scalable has a different power / performance behavior compared to the other workloads and that this difference is largely independent of clock rate. The Native Non-scalable benchmarks draw less power overall, and power increases less steeply as a function of performance increases. Native Non-scalable (SPEC CPU2006) is the most widely studied workload in the architecture literature, but it is the outlier. These results reinforce the importance of including scalable and managed workloads in energy evaluations.

3.4 Die Shrink

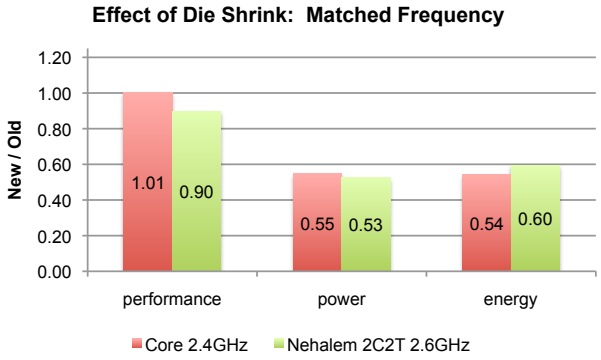
We used processor pairs from the Core (Core 2D (65)/ Core 2D (45)) and Nehalem (i7 (45)/ i5 (32)) microarchitectures to explore die shrink effects. These stock hardware comparisons are imperfect because they are not straightforward die shrinks. To limit the differences, we control for hardware parallelism by limiting the i7 (45) to two cores, and control for clock speed by running both Cores at 2.4GHz and both Nehalems at 2.66GHz. We also run them at their native speeds. Nonetheless, we cannot control for the cache size and other differences. For the Core, it appears that the die shrink was fairly straightforward, except that the Core 2D (45) uses a 3MB triple-port cache, whereas the Core 2D (65) uses a 4MB dual-port cache. Nehalem's changes are more extensive and include a reduction in the number of cores, the size of the cache, a more limited DMI interconnect instead of QPI, the integration of a PCIe controller in the i5 (32), and the inclusion of a GPU on a separate 45nm die within the same package. The GPU is not exercised by any of our benchmarks, but is nonetheless included in our power measurements. Notwithstanding these caveats, these architectures present an opportunity to compare power and performance across process technologies.

> ARCHITECTURE FINDING 4. *A die shrink is remarkably effective at reducing energy consumption, even when controlling for clock frequency.*

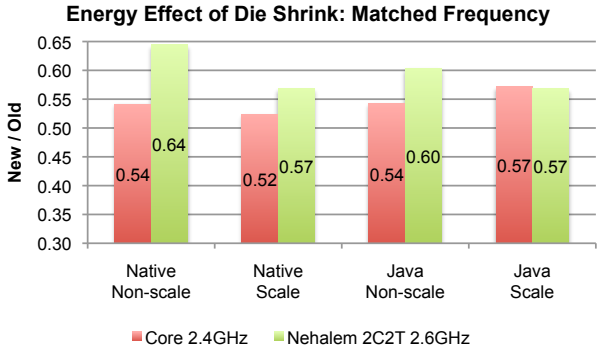
Figure 8(a) shows the power and performance effects of the die shrinks with the stock clock speeds for all the processors. The newer processors are significantly faster at their higher stock clock speeds and significantly more power efficient. Figure 8(b) shows



(a) Average impact of die shrink with native clocks.



(b) Average impact of a die shrink with matched clocks.



(c) Workload energy impact of a die shrink with matched clocks.

Figure 8. The impact of a die shrink for Core and Nehalem microarchitecture families.

the same experiment, but down clocking the newer processors to match the frequency of their older peers. Down clocking the new processors improves their relative power and energy advantage even further. Note that as expected, the die shrunk processors offer no performance advantage once the clocks are matched, indeed the i5 (32) performs 10% slower than the i7 (45). However, power consumption is reduced by 47%. This result is consistent with expectations, given the lower voltage and reduced capacitance at the smaller feature size.

> ARCHITECTURE FINDING 5. *Moving from 45nm to 32nm repeated the energy improvements of the previous generation.*

Figures 8(a) and 8(b) reveal a striking similarity in the power and energy savings between the Core (65nm / 45nm) and Nehalem (45nm / 32nm) die shrinks. This data suggests that Intel was able to maintain the same rate of energy reduction across each of these generations. ITRS predicted a 9% increase in frequency and 20% reduction in power from 45nm to 32nm [21]. Figure 8(a) is more encouraging, showing a 14% increase in performance and 23% reduction in power accompanying the 26% increase in stock frequency from the i7 (45) to i5 (32).

3.5 Gross Microarchitecture Change

We explore the power and performance effect of gross microarchitectural change through a series of comparisons where we can match architectural features such as processor clock, degree of hardware parallelism, process technology, and cache size. The processors in this study represent only one or two examples of the many processors built in each family. For example, Intel sells over sixty 45nm Nehalems ranging in price from around $190 to over $3700. However, we chose mainstream processors at similar price points for the most part. These microarchitecture comparisons are imperfect, but they provide broader perspective on the processors and workloads.

Figure 9 compares the Nehalem i7 (45) with with the NetBurst Pentium 4 (130), Bonnell Atom D (45), and Core 2D (45) microarchitectures, and it compares the Nehalem i5 (32) with the Core 2D (65). Each comparison configures the Nehalems to match the clock speed, number of cores, and hardware threads of the other architecture. It is pleasing, although unsurprising, to see the i7 (45) performing 2.6× faster than the Pentium 4 (130), while consuming just one third the power, when controlling for clock speed and hardware parallelism. Much of the 50% power improvement is attributable to process technology advances (Architecture Finding 4). However, this comparison does not control for memory speed, nor for three generations of process technology scaling.

> ARCHITECTURE FINDING 6. *Controlling for hardware parallelism and clock speed, Nehalem performs about 14% better than Core.*

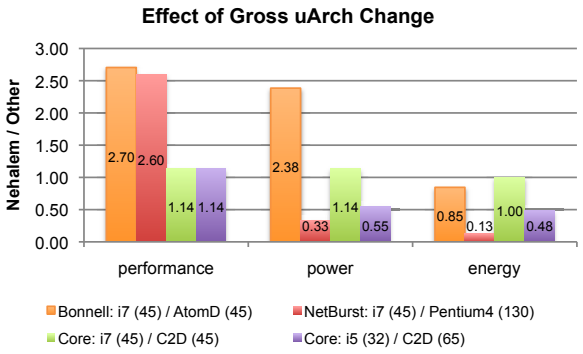
Both the i7 (45) and i5 (32) comparisons to the Core show that the move from Core to Nehalem yields a small 14% performance improvement. This finding is not inconsistent with Nehalem's stated primary design goals, i.e., delivering scalability and memory performance. Power increases by 14% when we hold process technology constant (i7 (45) / Core 2D (45)) and reduces 45% when we shift two technology generations (i5 (32) / Core 2D (65)).

> ARCHITECTURE FINDING 7. *Controlling for technology, hardware parallelism, and clock speed, the Nehalem has similar energy efficiency to Core and Bonnell.*

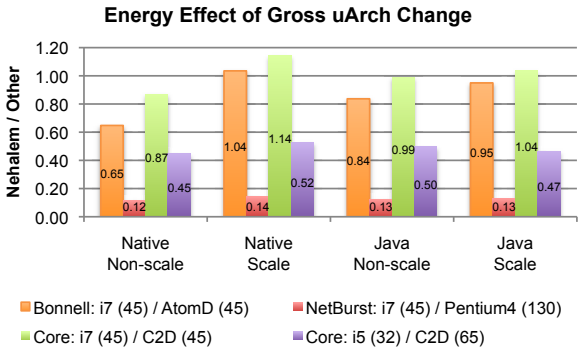
The comparisons between the i7 (45) and Atom D (45) and Core 2D (45) hold process technology constant at 45nm. All three processors are remarkably similar in energy consumption. This outcome is all the more interesting because the i7 (45) is disadvantaged since it uses fewer hardware contexts here than in its stock configuration. Furthermore, the i7 (45) integrates more services on-die, such as the memory controller, that are off-die on the other processors and thus outside the scope of the power meters. The i7 (45) improves upon the Core 2D (45) and Atom D (45) with a more scalable, much higher bandwidth on-chip interconnect, that is not heavily exercised by our workloads. It is impressive that despite all of these factors, the i7 (45) delivers similar energy efficiency to these two 45nm peers.

3.6 Turbo Boost Technology

Intel Turbo Boost Technology on Nehalem processors over-clocks cores under the following conditions [19]. With Turbo Boost enabled, *all* cores can run one "step" (133MHz) faster if temperature,



(a) Average impact of a gross μarch change.



(b) Workload energy impact of a gross μarch change.

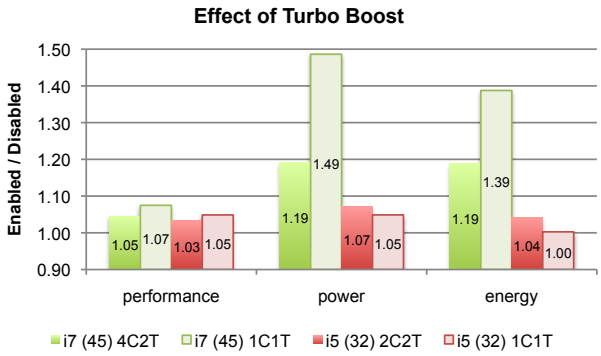
Figure 9. The impact of gross microarchitectural change.

power, and current conditions allow. When only one core is active, Turbo Boost may clock it an additional step faster. Turbo Boost is only enabled when the processor executes at its default highest clock setting. This feature requires on-chip power sensors which are currently not exposed to the programmer. We verified empirically on the i7 (45) and i5 (32) that all cores ran 133MHz faster with Turbo Boost. When only one core was active, the core ran 266MHz faster. Since the i7 (45) runs at a lower clock (2.67GHz) than the i5 (32) (3.46GHz), it experiences a relatively larger boost.

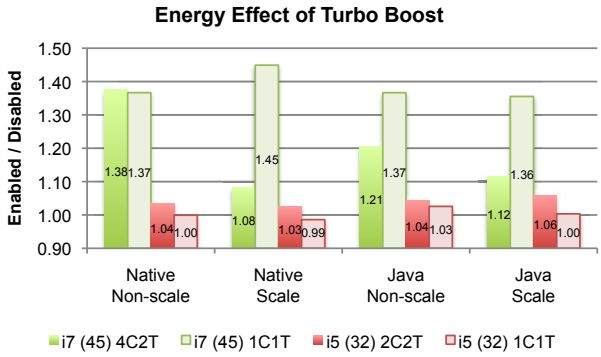
> ARCHITECTURE FINDING 8. *Turbo Boost is not energy efficient on the i7 (45).*

Figure 10(a) shows the effect of disabling Turbo Boost at the BIOS on the i7 (45) and i5 (32) in their stock configurations (dark) and when we limit each machine to a single hardware context (light). With the single hardware context, Turbo Boost will increment the clock by two steps if thermal conditions permit. The actual performance changes are well predicted by the clock rate increases. The i7 (45) clock step increases are 5 and 10%, and the actual performance increases are 5 and 7%. The i5 (32) clock step increases are 4 and 8%, and the actual performance increases are 3 and 5%. However, the i7 (45) responds with a substantially higher power increase and consequent energy overhead, while the i5 (32) is essentially energy-neutral.

Figure 10(b) shows that when all hardware contexts are available (dark), the non-scalable benchmarks consume relatively more energy than scalable benchmarks on the i7 (45) in its stock configuration. Because the non-scalable native and sometimes Java utilize only a single core, Turbo Boost will likely increase the clock by an additional step. Figure 10(a) shows that this technique is power-hungry on the i7 (45).

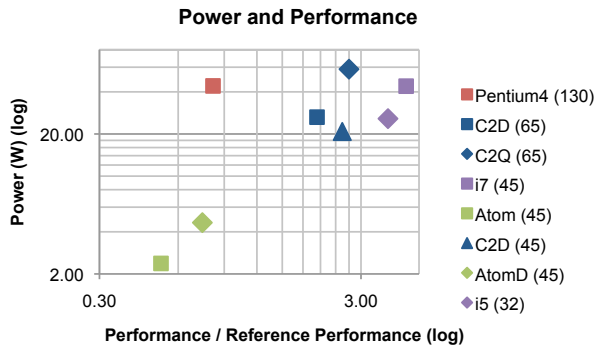


(a) Average impact of Turbo Boost.

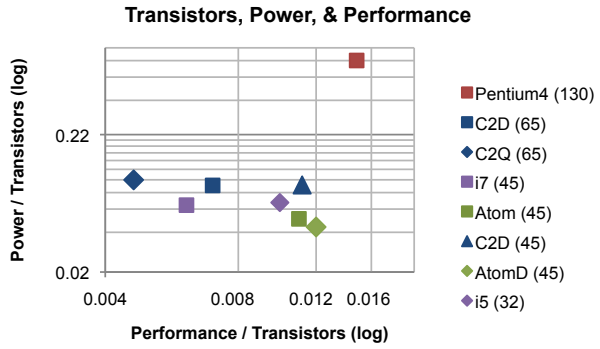


(b) Workload energy impact of Turbo Boost.

Figure 10. Workload impact of enabling Turbo Boost.



(a) Power / performance tradeoff by processor.



(b) Power / performance tradeoff as a function of transistors.

Figure 11. Power / performance tradeoff by processor.

4. Perspective

To give a broader view, this section: (1) examines the energy tradeoffs made by each processor over time, (2) examines the energy tradeoffs as a function of transistors count, and (3) conducts a Pareto energy efficiency analysis for our benchmarks running on the 45nm processors. This section presents processor performance relative to the reference performance for each benchmark shown in Table 1 and described in Section 2.6. We use the same methodology for energy, but because power is not dependent on benchmark length, we present measured power directly.

4.1 Historical Overview

Figure 11(a) plots the power and performance for each processor in their stock configuration relative to the reference performance, using a log / log scale. Both graphs in Figure 11 use the same color for all the experimental processors in the same family. The shapes encode release age: a square is the oldest, the diamond is next, and the triangle is the youngest, smallest technology in the family.

While mobile devices have historically optimized for power, general purpose processor design has not. Several results stand out that illustrate that power is now a first-order design goal and trumps performance in some cases. (1) The Atom (45) and Atom D (45) are designed as low power processors for a different market, however they do run all these benchmarks, and indeed they are the most power efficient processors. Compared to the Pentium 4 (130), they degrade performance modestly and reduce power enormously, consuming as little as one twentieth the power. (2) Comparing one generation between 65nm and 45nm with the Core 2D (65) and Core 2D (45) shows only a 25% increase in performance, but a 35% drop in power. (3) Comparing the most recent two generations with the i7 (45) and i5 (32), the figure shows that the i5 (32) delivers about 15% less performance, while consuming about 40% less power. This result has three root causes: (a) the i7 (45) has four cores instead of two on the i5 (32), (b) since half the benchmarks are scalable multithreaded benchmarks, the software parallelism benefits more from the additional two cores, increasing the advantage to the i7 (45), and (c) the i7 (45) has significantly better memory performance. Comparing the Core 2D (45) to the i5 (32) where the number of processors are matched, the i5 (32) delivers 50% better performance, while consuming around 25% more power than the Core 2D (45).

Contemporaneous comparisons also reveal the tension between power and performance. For example, the contrast between the Core 2D (45) and i7 (45) shows that the i7 (45) delivers 75% more performance than the Core 2D (45), but this performance is very costly in power with an increase of nearly 100%. These processors thus span a wide range of energy tradeoffs within and across the generations. Overall, these result indicate that optimizing for both power and performance is so far considerably more challenging than optimizing for performance alone.

ARCHITECTURE FINDING 9. *Power per transistor is relatively consistent within a microarchitecture family.*

Figure 11(b) explores the effect of transistors on power and performance by dividing them by the number of transistors in the *package* for each processor. We include all transistors because our power measurements occur at the level of the package, not the die. This measure is rough and will downplay results for the i5 (32) and Atom D (45), each of which have GPUs within the package.

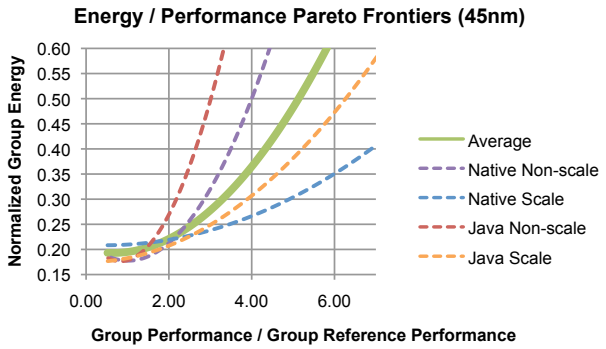


Figure 12. Energy / performance Pareto frontiers for our selection of 45nm processors executing at both stock and at clock and hardware context scaled configurations.

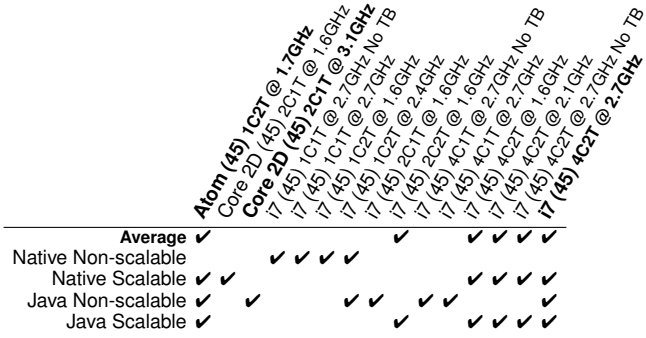


Table 5. Pareto efficient processor configurations for each benchmark group. Stock configurations are indicated in bold.

Even though our benchmarks do not exercise the GPUs, we include them in the totals because the GPU transistor counts on the Atom D (45) are undocumented. Note the similarity between the Atom (45), Atom D (45), Core 2D (45), and i5 (32), which at the bottom right of the graph, are the most efficient processors by the transistor metric. Even though the i5 (32) and Core 2D (45) have five to eight times more transistors than the Atom (45), they all eek out very similar performance and power per transistor.

The left-most processors yield the smallest amount of performance per transistor. Among these processors, the Core 2D (65) and i7 (45) yield the least performance per transistor. These two machines correspond to the ones with the largest (8MB) caches among our set. This result is consistent with Patt's observation that very large caches are a waste of transistors [27], but this result does not hold for all workloads.

The Pentium 4 (130) is perhaps most remarkable—it yields the most performance per transistor and consumes the most power per transistor by a considerable margin. Consider applying the die shrink parameters from Finding 4 to the Pentium 4 (130) design across four generations from 130nm to a 32nm. The resulting microarchitecture would reduce power four fold and increase performance two fold, sliding it down and to the right on the graph.

Performance per transistor is inconsistent across microarchitectures, but power per transistor is consistent. Power per transistor correlates well with microarchitecture, regardless of technology generation.

4.2 Pareto Analysis at 45nm

The Pareto frontier defines a set of choices that are most Pareto efficient in a tradeoff space, and identifies the most optimal choices within the space. Prior research uses the Pareto frontier to explore tradeoffs of power and performance using *models* and then derives potential architectural designs on the frontier [1, 17]. Instead of models, we present a Pareto frontier derived from *measured performance and power* on contemporary real processors. We hold the process technology constant and use the four 45nm processors: Atom (45), Atom D (45), Core 2D (45), and i7 (45). We expand the number of processors from four to twenty-nine by configuring the number of hardware contexts (SMT and CMP), by clock scaling, and by disabling Turbo Boost. We use the same data as analyzed in Section 3. We use the twenty-five non-stock configurations as proxies for alternative design points. We explore processor configurations that are most efficient in the performance × energy space.

The set of Pareto efficient choices is determined by plotting all choices on an energy / performance scatter graph, and then identifying those choices that are not dominated in performance or energy efficiency by any other choice [12]. Visually these configurations are the bottom-right-most choices on the graphs in a figure such as Figure 12.

Table 5 shows the Pareto efficient processor configurations on the frontier for each of the benchmark groups and the average. Fifteen of the twenty-nine processor configurations are represented in Table 5, the remaining fourteen processors, including all four Atom D (45) configurations, are not Pareto efficient for any of the five groupings. As shown in previous sections, the energy and performance behavior of each benchmark group differs and consequently, the selection of most efficient choices differs for each group. Notice further that: (1) Native Non-scalable shares only one choice with any other group, (2) Java Scalable shares the same choices as the average, and (3) of the eleven choices on the Java Non-scalable and Java Scalable frontiers, only two are common to both.

It is interesting to note that Native Non-scalable does not include the Atom (45) in its frontier. This finding contradicts Azizi et al., who conclude that 2-wide in-order cores and 2-wide out of order cores are Pareto optimal designs with respect to the Native Non-scalable benchmarks [1]. Instead we find that all of the Pareto efficient points for Native Non-scalable are various configurations of the 4-wide out of order i7 (45).

Figure 12 shows the resulting Pareto frontier curves for each benchmark group. The curve for a group is constructed by plotting the energy and performance of the group on each processor and fitting a polynomial curve through those processors that are Pareto efficient according to Table 5.

WORKLOAD FINDING 4. *Pareto analysis shows energy efficient architectural design is very sensitive to workload.*

The curves for each benchmark group deviate substantially from the average. Even when the groups share points in common, the respective points are usually in different places on the graph because each group exhibits a different energy / performance tradeoff. Comparing the scalable and non-scalable benchmarks at 0.40 normalized energy on the y-axis, it is impressive to see how the architectures we evaluate are able to very effectively exploit software parallelism, pushing the curves to the right, increasing performance from about 3 to 7 while holding energy constant. This measured behavior confirms prior model-based observations about the role of software parallelism in extending the energy / performance curve to the right [1, 17].

5. Related Work

The processor design literature is full of performance measurement and analysis, a tradition that began in the 1980s [9, 10]. Unfortunately, despite the growing importance of power, power measurements are relatively rare and new [20].

Measured Power Isci and Martonosi introduce a coordinated measurement approach that combines real total power using a clamp ammeter with performance counters for per unit power estimation [20]. They measure total power for an Intel Pentium 4 on the SPEC CPU2000 benchmark suite. Bircher and John [3] perform a detailed study of power and performance on AMD quad core Opteron and Phenom processors. They measure power using a series resistor, sampling the voltage across the resistor at 1KHz. Our work is complimentary. While the prior work takes a very close look at power on two processor cores and off-core resources, we examine trends across microarchitectures and technology generations.

Le Sueur and Heiser study the energy impact of dynamic voltage and frequency scaling (DVFS) on three server-class AMD processors fabricated at 130nm, 90nm, and 45nm using one memory-bound (181.mcf) SPEC CPU2000 benchmark [22]. They measure the whole system power with a RAM disk to prevent the inclusion of hard disk power in the measurements. They conclude that as the technology shrinks to 45nm the energy effectiveness of scaling down the frequency diminishes due to the increase of static power in lower voltages. We measure many benchmarks, processors, and use configuration to understand microarchitecture sensitivity.

Fan et al. study accurately estimating whole system power for large scale data centers [13]. They find that even the most power-consuming workloads draw less than 60% of the 'nameplate' peak power consumption. We measure chip power and support their results by showing that TDP does not predict chip measured power well on a range of workloads.

Power Models and Simulation Li et al. explore the design space of chip multiprocessors under various area and thermal constraints [24]. They combine decoupled trace-driven cache simulation and cycle-accurate execution-driven multicore simulation with detailed single-core simulation to achieve high accuracy in power and performance estimation for chip multiprocessors. While Li et al.'s work uses simulation and is prospective, ours uses direct measures and is retrospective.

Azizi et al. introduce a joint circuit-architecture design space exploration framework [1]. They create statistical architectural models using simple simulations and combine these models with circuit-level energy-performance tradeoff functions to populate the single-core energy and performance design space using a subset of SPEC CPU benchmarks. They use Verilog models synthesized using a CMOS 90nm standard-cell library to generate the circuit-level energy and delay tradeoff functions. The framework uses Pareto optimal tradeoff curves for cycles per instruction versus energy per instruction to determine microarchitectural parameters, circuit implementations, and operating voltage and frequency given performance and energy constraints. Our Pareto analysis uses measurements and is historical rather than predictive.

Charles et al. study and characterize the behavior of Turbo Boost technology using a Core i7 (45nm) processor with SPEC CPU2006 and a subset of PARSEC and multithreaded BLAST benchmarks [8]. They analyze how the rate of memory accesses and the overall processor load affects Turbo Boost activation. They estimate the energy impact of Turbo Boost using a power model derived from the time spent at different frequencies. Their findings are similar to ours—they find Turbo Boost is costly in energy, but we use measurements rather than models. Our Turbo boost study extends theirs by considering two processors, i7 (45) and i5 (32), and a diverse set of workloads.

Thermal Design Power (TDP) TDP is widely used in the literature as an estimate of chip power consumption. Horowitz et al. study energy across different process technologies while using TDP to estimate power and SPECmark to estimate performance for understanding CMOS scaling [17]. Hempstead et al. introduce an early stage modeling framework, Navigo, for CMP architecture exploration across future process technologies from 65nm down to 11nm [16]. Navigo models voltage and frequency scaling based on ITRS [21] and Predictive Technology Models [25]. As the starting point, Navigo uses reported TDP for power and SPECmarks for performance and then predicts the power and performance of processors in future technologies. Chakraborty uses TDP as the power envelope for studying CMP power consumption trends in future technologies [7]. Perhaps it is already well known that TDP is a poor estimate for actual chip power consumption, but our data suggests these studies need to be reconsidered.

We published preliminary data and analysis of TDP, CMP scaling, and historical trends without the Native Scalable benchmarks in a workshop paper [11]. The data and methodology here are more complete and rigorous. For example, the data and analysis include aggregation by benchmark group, Native Scalable benchmarks, more samples of all native benchmarks, sensor calibration, sensor validation, and extensive architecture configuration. Furthermore, all the feature analysis in Section 3 and the Pareto analysis are new here.

Methodology Although the results show conclusively that managed and native workloads have different responses to architectural variations, perhaps this result is not or should not be surprising. However, very few academic publications with processor measurements or simulated designs use Java or any other managed workloads. The evaluation methodologies for real processors that we use here, however, are well developed [5, 14]. Objections to using managed languages in architectural design include lack of simulator support and simulation time. However, prior work shows how to modify simulators to handle Java [4]. The modest execution times of DaCapo were intended to ease experimentation on real processors and simulators, while also providing substantial and diverse workloads [4]. Some additional methodologies are needed to simulate managed workloads, but prior work has addressed these issues as well [15].

6. Conclusion

As far as we are aware, this paper is the first to quantitatively study measured power and performance at the chip level across hardware generations using single threaded and multithreaded, native and managed workloads. The volume of data and results do not lend themselves to concise conclusions, but they do offer three recommendations. Manufacturers: (1) Expose on-chip power meters to the community. Researchers: (2) Use both managed and native workloads. (3) Measure power and performance to understand and optimize power, performance, and energy.

Acknowledgements

A number of people have generously provided us with assistance. We thank Bob Edwards at ANU for helping fabricate and calibrate the current sensors. We thank Daniel Frampton for managing and configuring the machines, and for his feedback. We thank Katherine Coons, Pradeep Dubey, Jungwoo Ha, Laurence Hellyer, Daniel Jiménez, and Bert Maher for their comments and suggestions. We thank our ASPLOS shepherd, David Patterson, and John Hennessy for their feedback and encouragement.

References

[1] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis. In *ACM/IEEE International Symposium on Computer Architecture*, pages 26–36, 2010.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, January 2008.

[3] W. L. Bircher and L. K. John. Analysis of dynamic power management on multi-core processors. In *ACM International Conference on Supercomputing*, pages 327–338, Island of Kos, Greece, 2008.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, Oct. 2006.

[5] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffman, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake up and smell the coffee: Evaluation methodologies for the 21st century. *Communications of the ACM*, 51(8):83–89, Aug. 2008.

[6] M. Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE SSCS Newsletter*, pages 11–13, Winter 2007.

[7] K. Chakraborty. *Over-provisioned Multicore Systems*. PhD thesis, University of Wisconsin-Madison, 2008.

[8] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova. Evaluation of the Intel® Core™ i7 Turbo Boost feature. In *IEEE International Symposium on Workload Characterization*, pages 188–197, 2009.

[9] J. S. Emer and D. W. Clark. A characterization of processor performance in the VAX-11/780. In *ACM/IEEE International Symposium on Computer Architecture*, pages 301–310, 1984.

[10] J. S. Emer and D. W. Clark. Retrospective: A characterization of processor performance in the VAX-11/780. *ACM 25 Years ISCA: Retrospectives and Reprints 1998*, pages 274–283, 1998.

[11] H. Esmaeilzadeh, S. M. Blackburn, X. Yang, and K. S. McKinley. Power and performance of native and Java benchmarks on 130nm to 32nm process technologies. In *Sixth Annual Workshop on Modeling, Benchmarking, and Simulation*, June 2010.

[12] H. Esmaeilzadeh, T. Cao, X. Yang, S. M. Blackburn, and K. S. McKinley. Source materials in ACM Digital Library for: Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–332, Mar. 2011. URL `http://doi.acm.org/10.1145/1950365.1950402`.

[13] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *ACM/IEEE International Symposium on Computer Architecture*, pages 13–23, San Diego, CA, 2007.

[14] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*, pages 57–76, 2007.

[15] J. Ha, M. Gustafsson, S. M. Blackburn, and K. S. McKinley. Microarchitectural Characterization of Production JVMs and Java Workloads. In *IBM CAS Workshop*, Feb. 2008.

[16] M. Hempstead, G.-Y. Wei, and D. Brooks. Navigo: An early-stage model to study power-contrained architectures and specialization. In *Workshop on Modeling, Benchmarking, and Simulations*, June 2009.

[17] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, and K. Bernstein. Scaling, power, and the future of CMOS. In *Proceedings of International Electron Devices Meeting*, pages 7–15, December 2005.

[18] Intel Corporation. Intel Hyper-Threading Technology, 2011. URL `http://www.intel.com/technology/platform-technology/hyper-threading`.

[19] Intel Corporation. Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors. White Paper, Nov. 2008.

[20] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *IEEE International Symposium on Microarchitecture*, pages 93–104, December 2003.

[21] ITRS Working Group. International technology roadmap for semiconductors, 2011. URL `http://www.itrs.net`.

[22] E. Le Sueur and G. Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Workshop on Power Aware Computing and Systems*, Vancouver, Canada, Oct. 2010.

[23] S. H. Li. Linux kernel bug 5471, 2011. URL `https://bugzilla.kernel.org/show_bug.cgi?id=5471`.

[24] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron. CMP design space exploration subject to physical contraints. In *International Symposium on High Performance Computer Architecture*, pages 17–28, Feb 2006.

[25] Nanoscale Integration and Modeling (NIMO) Group. Predictive technology model, 2011. URL `http://ptm.asu.edu`.

[26] V. Pallipadi and A. Starikovskiy. The ondemand governor: Past, present and future. In *Proceedings of Linux Symposium*, volume 2, pages 223–238, July 2006.

[27] Y. Patt. Future microprocessors: Multi-core, mega-nonsense, and what we must do differently moving forward. Distinguished Lecture at UIUC, (April 2010), 2011. URL `http://www.parallel.illinois.edu/presentations/2010_04_30_Patt_Slides.pdf`.

[28] R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The energy efficiency of CMP vs. SMT for multimedia workloads. In *ACM International Conference on Supercomputing*, pages 196–206, Malo, France, 2004.

[29] R. Singhal. Inside Intel next generation Nehalem microarchitecture. Intel Developer Forum (IDF) presentation (August 2008), 2011. URL `http://software.intel.com/file/18976`.

[30] Standard Performance Evaluation Corp. SPEC Benchmarks, 2010. URL `http://www.spec.org`.

[31] Standard Performance Evaluation Corporation. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Newsletter, Computer Architecture News*, 34(4), September 2006.

[32] The DaCapo Research Group. The DaCapo Benchmarks, beta-2006-08, 2006. URL `http://www.\-dacapo\-bench.\-org`.

[33] TIOBE Software. TIOBE Programming Community Index for January 2011, 2011. URL `http://www.tiobe.com`.

[34] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ACM/IEEE International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, June 1995.