Karlsruher Institut für Technologie

# System Support for Distributed Energy Management in Modular Operating Systems

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie
genehmigte

Dissertation
von

Jan Stoess

aus Karlsruhe

| | |
|---|---|
| Tag der mündlichen Prüfung: | 5. Februar 2010 |
| Hauptreferent: | Prof. Dr. Frank Bellosa<br>Karlsruher Institut für Technologie |
| Korreferent: | Prof. Dr. Gernot Heiser<br>University of New South Wales |

# Abstract

Energy management has become a challenge for modern computing environments that needs to be addressed by all involved components, including the operating system. At the same time, the trend in operating-system design is moving away from monolithic to modular structures; modern operating systems often come as a small kernel and a set of unprivileged service modules atop. Their custom operating-system abstractions render them extensible; their integrated virtualization capabilities retain compatibility to existing applications. Nonetheless, most existing energy-management schemes are tailored to monolithic operating systems, where software and hardware can be directly controlled to meet thermal or energy constraints. A modern operating system, however, consists of multiple components, and direct or centralized energy management is unfeasible.

This thesis proposes a novel approach for managing energy in modular operating systems. Our approach strives to enable energy awareness and energy management if the resource-management subsystem is distributed and scattered among operating-system modules rather than being centralized and monolithic. There are four key achievements: a model for modularization-aware energy management; the support for exposed and distributed energy accounting and allocation; the use of different energy-management interaction protocols; and, finally, the support for virtualization of energy effects.

We have implemented a prototype of our approach for a modular, virtualization-capable microkernel operating system. Our prototype supports processor and disk energy management, at the level of physical and virtual devices. To that end, it features distributed and exposed mechanisms for accounting and allocation of processor and disk energy both to complete virtual machines and to individual virtualized applications. Experiments show that the prototype accurately accounts and allocates processor and disk energy consumption to different notions of applications at runtime. Our mechanisms enable extensible and easily adaptable energy policies. Overheads for processor energy management may be dramatic for micro-benchmarks, but percolate to application level at a more moderate level. Overheads for combined processor and disk management are limited to an increase in processor utilization. Our experiments also reveal that there is an interdependency between accuracy and performance of energy-management mechanisms; using different management protocols, our prototype enables developers of energy policies to choose themselves the particular point in the trade-off space.

# Zusammenfassung

Über Jahrzehnte hinweg waren Leistung und Leistungssteigerung die bestimmenden Ziele bei der Entwicklung von Rechensystemen. In jüngster Zeit hat sich allerdings eine weitere Herausforderung dazugesellt: die Energieeffizienz. Der Wunsch nach energieeffizienteren Computern hat zwei Hauptursachen: Erstens erfreuen sich mobile Rechensysteme wie tragbare Telefone oder andere Kleincomputer einer zunehmenden Verbreitung, was Computerhersteller vor die Herausforderung stellt, leistungsfähige Computergeräte mit begrenztem Energievorrat zu entwickeln. Zweitens hat die Computerindustrie insgesamt mit einem stetig steigenden Energiehunger der Rechensysteme (insbesondere der Serversysteme) und einer damit verbundenen Kostenexplosion zu kämpfen. Der wachsende Energiehunger ist vor allem auf die gesteigerten Energiedichten in modernen Rechnerarchitekturen zurückzuführen; der dadurch entstehende Bedarf an aufwändiger Kühlung vergrößert den Energiebedarf entsprechend zusätzlich.

Nicht überraschend herrscht daher mittlerweile ein breiter Konses in Forschung und Technik, dass Energieeffizienz ein Hauptkriterium bei der Entwicklung von Computeranlagen sein sollte. Im Bereich der mobilen Computergeräte steht dabei naturgemäß der Wunsch im Vordergrund, möglichst lange Batterielaufzeiten zu ermöglichen. Bei Standrechnern und Serversystemen wiederum stellt die Reduktion der Energiekosten das Hauptziel dar. Das allgemeingehaltene Ziel der Energieeffizienz zergliedert sich somit in mehrere Teilziele, von denen die folgenden vier als maßgeblich bezeichnet werden können: erstens sollten Computersysteme natürlich weniger Leistung aufnehmen, sowohl um die Nutzbarkeit der Mobilgeräte zu erhöhen als auch um Energiekosten und deren ökologische Auswirkungen zu reduzieren; darüber hinaus sollten zweitens Computersysteme aber auch möglichst die Betriebstemperaturen gering halten, weil sich dadurch Bedarf und Kosten der Kühlung in Grenzen halten lassen, was erneut den Energieverbrauch der Rechenanlage verringert. Drittens sollten Computersysteme nach Möglichkeit das Auftreten von Leistungs- und Temperaturspitzen vermeiden, weil Kühlung und Elektizitätsversorgung von Computern oftmals nach Maximalbedarf und Maximaltemperaturen entworfen werden; jedes Vermeiden von Spitzenbedarf erlaubt daher eine bessere Anpassung der Strom- und Kühlinfrastruktur an tatsächliche, durchschnittliche Gegebenheiten. Viertens schließlich sollten Computersysteme eine anwendungsgetriebene Form des Energiemanagements betreiben, da nur der

Miteinbezug von Software und Anwendungen die Ziele der Energieeffizienz mit anderen, ebenso wichtigen Anforderungen an Rechenanlagen wie Dienstgüte, Benutzerfairness, oder bedarfsgerechter Kostenabrechnung in Einklang zu bringen erlaubt.

Traditionell hat man sich dem Ziel energieeffizienter Computer meist auf der Ebene der Hardware und Anlagen gewidmet, beispielsweise durch energieeffiziente Rechnerarchitekturen oder durch thermodynamisch effizientere Kühlanlagen. Tatsächlich sollte dieses Ziel aber auf allen Ebenen der Rechenanlagen angegangen werden; insbesondere das Betriebssystem trägt hierbei eine Schlüsselrolle, da diesem die Laufzeitkontrolle sowohl der Hardware als auch der Software untersteht. Ein betriebsystemseitiges Energiemanagement kann somit dafür sorgen, dass das Erreichen von Energieeffienz global und dynamisch erfolgt, also alle Hardwaregeräte und Anwendungen umfasst und zur Laufzeit stattfindet. In der Tat hat sich eine Vielzahl von wissenschaftlichen Untersuchungen dem Problem des betriebssystemseitigen Energiemanagements verschrieben. Generell stützt man sich hier zumeist auf Laufzeitmechanismen zur Überwachung und Steuerung sowohl der Geräte und ihrer Energiezustände, als auch der Anwendungen und deren Auswirkungen auf den Energieverbrauch. Mit Hilfe solcher Mechanismen versucht man sodann, Leistungs- und Nutzbarkeitsanforderungen mit Energiezielen unter einen Hut zu bringen; üblicherweise bedeutet das entweder, ein gegebenes Mass an Rechenleistung und Dienstgüte zur Verfügung zustellen, das aber mit möglichst wenig Energie, mit möglichst geringer Temperatur und mit möglichst wenig Energie- und Temperaturspitzen. Oder man konzentriert sich umgekehrt darauf, im Rechensystem gewisse Auflagen bezüglich Energieverbrauch oder Temperatur nicht zu überschreiten, dabei aber ein Maximum an Leistung und Dienstgüte herauszuholen.

Neben der gesteigerten Wichtigkeit des betriebssystemseitigen Energiemanagements lässt sich noch ein zweiter wichtiger Trend in der Betriebsystemforschung und -praxis beobachten: der Trend zu modularen Betriebssystemstrukturen. Der traditionelle Betriebssystemaufbau unterscheidet lediglich zwischen Nutzer- und Kernbereich: Anwendungen laufen im Nutzerbereich, während die gesamte Funktionalität des Betriebssystems monolithisch im privilegierten Kernbereich verbleibt. Waren frühe Betriebssysteme verhältnismäßig klein und berschränkt in ihren Funktionen, so sind mit zunehmender Leistungsfähigkeit der Rechenanlagen auch Größe und Komplexität des Betriebssystems beträchtlich gewachsen: ein moderner Betriebssystemkern wie Windows or Linux hat einen Quellcodeumfang von Millionen von Zeilen Code und umfasst eine unüberschaubare Anzahl von Funktionen und Diensten. Die wachsende Anzahl von Anwendungsszenarien, welcher sich moderne Betriebsysteme ausgesetzt sehen — so wird beispielsweise Linux heutzutage sowohl in mobilen Kleinstgeräten als auch in Rechenzentren eingesetzt —, hat ihren Teil zur wachsenden Komplexität beigetragen. Die Kombination aus monolitischem Betriebssystemaufbau einerseits und wachsender Betriebssystemkomplexität anderseits führt jedoch zu beträchtlichen strukturellen Problemen. Als wichtigste Probleme sind die starken Einschränkungen hinsichtlich Flexibilität und Ausfallsicherheit zu nennen, welche sich aus der gemeinsamen Unterbringung aller Betriebssystemfunktionen in einem einzigen privilegierten Kern und aus dem Fehlen von Modulgrenzen innerhalb dieses Kerns ergeben.

Zur Behebung der Probleme monolitschen Betriebssystemaufbaus hat die Forschung eine Vielzahl unterschiedlicher Lösungsansätze beigetragen; die vorliegende Arbeit befasst sich mit einer Klasse solcher Lösungen, welche als modulares Betriebssystem bezeichnet wird. Kurz zusammengefasst besteht der Ansatz modularer Betriebssysteme darin, einzelne Betriebssystemdienste in Softwaremodulen zu kapseln, welche ausschließlich über wohldefinierte Schnittstellen interagieren. Zur Eindämmung von Fehlern können Module in unterschiedlichen Schutzdomänen untergebracht werden und verschiedene Privilegienstufen zugewiesen bekommen. Als Konzept und Forschungsansatz existieren modulare Betriebssysteme seit geraumer Zeit; erst vor kurzem jedoch haben modulare Betriebssysteme auch zunehmend praktische Anwendung gefunden. Typische Ausprägungen bestehen aus einem kleinen Betriebssystemkern sowie mehreren Betriebsystemdiensten, wobei letztere deprivilegiert im Nutzerbereich laufen. Für Anpassbarkeit und Erweiterbarkeit des Betriebsystems sorgen anwendungsspezifische Betriebsystemabstraktionen und -dienste; für die Kompatibilität zu existierenden Anwendungen steht meist eine integrierte Virtualisierungsschicht zur Verfügung.

## Problemstellung

Zwar gibt es eine Vielzahl existierender Untersuchungen und Lösungsansätze aus der Forschung, welche sich dem betriebssystemseitigen Energiemanagement widmen. Jedoch haben sich diese Ansätze im Wesentlichen auf Standardbetriebssysteme mit monolitschem Aufbau konzentriert. Solche monolithischen Kerne vereinigen alle Betriebsystemfunktion in einem einzigen Kern, welcher zudem volle und exklusive Kontrolle sowohl über alle Hardwaregeräte als auch über alle Anwendungen ausübt. Die Folge ist jedoch, dass das Energiemanagement typischerweise gleichermaßen zentralistisch und monolitisch organisisert ist. Im monolitischen Kern ergeben sich dadurch auch keinerlei Probleme, kann doch dieser dank privilegiertem Direktzugriff auf alle Geräte und Anwendungen das gesamte Ressourcenmanagement unmittelbar nach den gewünschten Energiekriterien ausrichten.

Jedoch ist solcherart zentralistisches Energiemanagement, so passend sich es auch in den monolitischem Betriebssystemaufbau einfügen mag, grundsätzlich ungeeignet für modulare Betriebssysteme. Modulare Betriebssysteme lassen die einfache Strukturierung in Anwendungs- und Kernbereich hinter sich und bauen das Gesamtsystem stattdessen als eine Menge einzelner verteilter Module auf unterschiedlichen Hierarchiestufen auf, typischerweise unter Bereitstellung eines sehr kleinen privilegierten Kerns sowie einer Vielzahl von darüberliegenden Dienstmodulen. Mit einer solchen Struktur verteilen sich die für das Energiemanagement relevanten Kontrollfunktionen und Datenstrukturen naturgemäß auf viele unterschiedliche Module. So verfügt der Kern über Direktzugriff auf elementare Hardwarefunktionen; alle anderen Informationen und Kontrollfunktionen, einschließlich der Gerätetreiber, sind jedoch in die verschiedenen Dienstmodule im Nutzerbereich ausgelagert. Diese wiederum haben eng begrenzte Privilegien hinsichtlich der Hardware, oft mit Zugriffsberechtigung auf nur jeweils einzelne Geräte pro Modul. Gleichermaßen gestaltet sich die Anwendungskontrolle: hat der Kern Kontrolle über maschinennahe Anwendungensabstraktionen und deren Verhalten, so sind höherwertige oder komplexere Arten von Anwendungen erst in höheren Schichten des modularen Betriebssystems zu finden.

In solch einer Umgebung ist direkt und zentralistisch organisiertes Energiemanagement schlechterdings unmöglich, da sich Kontrolle und Information über den Energiezustand des Systems auf mehrere isolierte Module in unterschiedlichen Betriebssystemschichten verteilen. Diese Verteilung führt zu drei wesentlichen Problemen für das Energiemanagement:

Erstens findet, durch die Verteilung und Modulgrenzen bedingt, kein zentrales Ressourcenmanagement mehr statt, sodass sich die für das Energiemanagement wichtigen Informationen — Zustand und Verhalten von Hardware und Anwendungen in Bezug auf Energieverbrauch und Energieeffekte — nunmehr isoliert voneinander in vielen unterschiedlichen Modulen befinden, wobei sich einzelne Module und Informationsteile hinsichtlich inneren Aufbaus und Semantik unterscheiden können. Damit ein Energiemanagement stattfinden kann, müssen diese Informationen nun explizit ermittelt, transportiert und zusammengeführt werden. Durch Virtualisierungsschichten und sonstige Kompatibilitätsanforderungen ergeben sich dabei zusätlizche Einschränkungen beim Entwurf der zum Transport und Zusammenführen notwendigen Schnittstellen und Protokolle.

Zweitens ergibt sich in modularen Betriebssystemen ein zusätzlicher Bedarf an Kommunikation und Datenaustausch. Die am Energiemanagement beteiligten Dienste und Funktionen verteilen sich nunmehr auf verschiedene Module, welche durch geeignete Schutzmechanismen (etwa Hardware-Addressräume) voneinander abgegrenzt werden. Der Aufbau solcher Schutzgrenzen hat seinen Preis: sind im monolitischen Betriebssystem andere Dienste und Funktionen direkt, also durch Prozeduraufrufe und Speicherzugriffsoperationen erreichbar, so müssen für diesen Vorgang im modularen Betriebssystem spezielle Kommunikationsprimitive benutzt werden, um die Schutzdomänen entsprechend geordnet zu durchqueren. Für ein effektives Energiemanagement müssen die dabei entstehenden Zusatzkosten und die zusätzliche Komplexität möglichst schon beim Entwurf miteinbezogen werden.

Drittens schließlich führt der modulare Betriebssystemaufbau zu einer drastischen Zunahme von Abhängigkeiten und Wechselwirkungen innerhalb des Energiemanagements. Hat der Monolith lediglich zwischen einer Anwendungs- und einer Hardwareschicht unterschieden, so ergeben sich im modularen Betriebssystem komplexe Pfade, welche von den verschiedenden Arten von Anwendungen durch die jeweiligen Systemdienste hindurch hinab zu den energieverbrauchenden, jeweils aber von unterschiedlichen Treibern verwalteten Geräten führen. Soll Energiemanagement jedoch die Anwendungsebene miteinbeziehen, so müssen die Abhängigkeiten und Wechselwirkungen der Ressourcen- und Energieverbrauchspfade ebenfalls in den Managementprozess eingebunden werden.

Wie bereits angesprochen, existiert eine Vielzahl von Vorschlägen und Ansätzen aus der Forschung zur Steigerung der Energieeffizienz aus dem Betriebssystem heraus; keiner dieser Ansätze hat sich jedoch, wie die vorliegende Arbeit, mit dem Problem befasst, wie die systemseitige Unterstützung für Energiemanagement in modular aufgebauten Betriebssystemen zu bewerkstelligen ist. Die allermeisten Ansätze zum betriebssystemseitigen Energiemanagement konzentrieren sich auf herkömmliche, monolithische Betriebssystemstrukturen und lassen das Problem der Modularisierung unbeachtet. Zwar ist zu erwarten, dass große Teil dieser Ansätze, inbesondere die Algorithmen und Strategien, prinzipiell auch in modularen Betriebssysteme Anwendung finden können. Jedoch setzt dies eine geeignete Unterstützung seitens des Betriebssystems voraus.

Wesentlich weniger wissenschaftliche Untersuchungen setzen sich dagegen mit Energiemanagement für ein Betriebssystem, welches einer Art Modularisierung unterzogen ist, auseinander. Eine dieser Untersuchungen hat sich mit der Frage befasst, wie Energieabrechnung und Energie-Preiskalkulationen in einem sogenannten vertikal strukturierten Betriebssystem gewährleistet werden können. Solch vertikal strukturierte Betriebssysteme können durchaus als modularisiert bezeichnet werden, obwohl die Modularisierung nur in eine Richtung erfolgt. Während die vertikale Unterteilung die Energieabrechnung erleichtert — Abrechnungsdaten fallen nur auf unterster Schicht an und können jeweils direkt der darüberliegenden Schichten in Rechnung gestellt werden — wird sie der Komplexität anderer modularer Betriebssysteme (zum Beispiel mikrokernbasierter Systeme), welche keine strikt vertikale Unterteilung aufweisen, hinsichtlich des Energiemanagement nicht gerecht. Eine Reihe weiterer Untersuchungen hat sich mit der Frage befasst, wie Energiemanagement in virtualisierten Betriebsystemumgebungen durchgeführt werden kann, eine weitere, popouläre Ausprägung modularer Betriebssysteme. Hinsichtlich ihrer Ziele und Erkennnisse stimmen diese Ansätze mit der vorliegenden Arbeit oftmals überein, konzentrieren sich al-

lerdings im Wesentlichen ausschließlich auf Virtualisierungssysteme und lassen wichtige Merkmale anderer modularer Betriebssysteme außer Acht. Schließlich existiert eine weitere Reihe Untersuchungen zum Problem, wie Energiemanagementfunktionen in einem mikrokernbasierten Betriebssystem bereitgestellt werden können. Typischerweise befassen sich diese Untersuchungen speziell mit mobilen Endgeräten. Sie zielen daher hauptsächlich darauf ab, Batterielaufzeiten zu erhöhen, schenken aber anderen Aspekten wie der geeigneten Energieabrechnung oder -budgetierung weniger Aufmerksamkeit. Des weiteren lassen diese Ansätze die Frage ausser Acht, wie sich Energiemanagement ändert, sobald eine Virtualisierungsschicht ins modulare Betriebssystem eingebunden wird.

## Ansatz

Die vorliegende Arbeit stellt ein neuartiges Konzept zur systemseitigen Unterstützung für Energiemanagement in modularen Betriebssystemumgebungen vor, da solche zunehmend in Forschung und Praxis Verbreitung finden. Ziel der Arbeit ist es, energiegewahres und energieeffizientes Ressourcenmanagement angesichts verteilter und verstreuter Resourcenmanagementprozesse im Modularbetriebssystem zu ermöglichen. Die von uns angestrebte Unterstützung sollte genügend Flexibilität besitzen, um der Vielzahl existierender Energiemanagementalgorithmen und -strategien gerecht zu werden; sie sollte die Effizienz des Betriebssystems erhalten, trotz existierender Modulgrenzen und erhöhtem Kommunikationsaufwand; und sie sollte die Vorteile modularer Betriebssysteme auch für allfällig integrierte Energiemanagementsubsysteme aufrechterhalten, auf der anderen Seite aber weiterhin Kompatibilität gegenüber existierenden Anwendungen bewahren können. Zusammengefasst sind die wesentlichen Beiträge dieser Arbeit:

**Modell für Modularisierungsgewahres Energiemanagement** Der erste Beitrag unser Arbeit besteht in einem Modell, welches das Betriebssystem als eine Menge von Modulen begreift und den Prozess des Energiemanagements als eine rückgekoppelte Schleife, deren Kontrollfluss sich über eines oder mehrere solcher Module erstreckt. Des weiteren fußt das Modell auf der Einheit Energie als alleiniger Abstraktion zur Formulierung und Realisierung des Managementprozesses, da nur Energieeinheiten über Geräte- und Anwendungsgrenzen hinweg verteilbar, unterteilbar und konvertierbar sind.

**Exponierte und Verteilte Energieabrechnung** Der zweite Beitrag besteht in einem verteilten Energieabrechnungsverfahren, welches die von den Computersystem verbrauchte Energie akkurat auf die Anwendungen zurückzuführen erlaubt. Das Verfahren ist verteilt konzipiert, so dass es sich über Modulgrenzen hinweg in der Lage zeigt, auch mit komplexen Ressourcen- und

Energieverbrauchspfaden und den entsprechenden Abhängigkeiten zurechtzukommen.

**Exponierte und Verteilte Energieallokation** Der dritte Beitrag besteht in einem Verfahren zur exponierten und verteilten Energieallokation, welches für die jeweiligen Ressourcenverwaltungsmodule geeignete Allokationsmechanismen zur Verfügung stellt und exportiert, so dass diese von den in anderen Modulen liegenden Entscheidungsprozessen und Strategien des Energiemanagements dazu verwendet werden können, den Ressourcen- und Energieverbrauch so aus der Ferne zu steuern, dass die gegebenen Energieeffizienzziele erreicht werden können.

**Interaktionsprotokolle für das verteilte Energiemanagement** Der vierte Beitrag besteht in einer Ausforschung zweier Protokolle für die Abwicklung der für das verteilte Energiemanagement notwendigen Kommunikation. Unsere Beobachtung ist dabei, dass die Art der Kommunikation Auswirkungen sowohl auf die zeitliche Genauigkeit als auch auf die Effizienz des Energiemanagementprozesses hat. Es werden daher zwei verschiedene Protokolle ausgeleuchtet, welche sich hinsichtlich der Zeitdauer zwischen Anfallen der zur Kommunikation vorgesehenen Energiemanagement-Informationseinheiten und der eigentlichen Übertragung dieser Einheiten unterscheiden.

**Virtualisierung von energetischen Effekten** Der fünfte und letzte Beitrag besteht in der geeigneten Emulation energetischer Effekte von Hardwaregeräten, um den Energiemanagementprozess nicht nur auf der Ebene der Hardware sondern auch auf der Ebene virtueller Geräte zu gewährleisten. Moderne Betriebssysteme setzen immmer häufiger Virtualisierungsschichten ein, um ihre Kompatibilitätsprobleme zu lösen. Eine Virtualisierung von Energieeffekten einzelner Geräte bietet daher den Vorteil, dass ein modulares Betriebssystem gleichzeitig abwärtskompatibel einerseits und offen für anwendungsspezifische Optimierungen im Energiemanagement andererseits zu sein vermag, da Optimierungen in Richtung Energieeffizienz nunmehr schrittweise in die virtualisierten Ressourcenmamagementsubsysteme eingeführt werden können.

## Validierung

Zur Validierung des Ansatzes wurde für die vorliegende Arbeit ein Prototyp entwickelt, welcher die Gültigkeit der Konzepte anhand eines mikrokernbasierten Mehrmodulbetriebssystems überprüft. Konkret fußt der Prototyp auf einer Implementierung der L4-Mikrokernreihe und der Implementierung mehrerer Systemdienste im Nutzerraum; insbesondere umfasst der Prototyp auch eine Virtualisierungsschicht, welche para-virtualisierte Linux 2.6.9 Gastbetriebssysteme unterstützt. Das Gesamtsystem läuft auf Computern der IA-32 Architektur. Durch die von L4 bereitgestellten Grundabstraktionen lässt sich das System sehr leicht erweitern; durch die mitgelieferte Virtualisierung bleibt es dabei gleichzeitig abwärtskompatibel gegenüber Linux-Anwendungen. Insgesamt stellt dieses Mehrmodulbetriebssystem einen typische Vertreter eines modernen modularen Systems dar, wie man es auch in ähnlicher Form in der Praxis finden könnte; es bietet somit eine ideale Plattform zur Validierung der vorgeschlagenen Energiemanagementkonzepte.

Die prototypische Implementierung der vorgeschlagenen systemseitigen Unterstützung für das Energiemanagement fand dabei in verschiedenen Modulen und auf mehreren Ebenen des mikrokernbasierten Modularbetriebssystems statt; unterstützt wird zur Zeit das Energiemanagement für Prozessoren und Festplatten. Die Prozessorverwaltung wird dabei direkt von L4 übernommen, während Festplattenlaufwerksdienste von speziellen Treiberprogrammen im Nutzerbereich bereitgestellt werden. Der Prototyp unterstützt derzeit Energiemanagement auf der Ebene physischer Geräte und auf der Ebene virtualisierter Geräte. Energieabrechnung und Energieallokation erfolgen in verteilter Art und Weise und umfassen dabei sowohl virtuelle Maschinen als auch einzelne virtualisierte Anwendungen innerhalb einer virtuellen Maschine. Die prototypische Energiemanagementinfrastruktur ist erweiterbar gestaltet und erlaubt die Anpassung oder den Austausch einzelner Energiemanagementstrategien und deren Implementierungen, ohne dass dabei die Managementmechanismen ausgetauscht werden müssten.

Die Auswertung der prototypischen Implementierung wurde auf einer Standrechner der IA-32 Architektur durchgeführt, welcher mit einem Pentium D 830 mit 3 GHz Taktfrequenz, einer Intel 945G Hauptplatine mit 2 GByte Hauptspeicher, sowie einer 160 GByte großen Maxtor Diamond Max Plus 9 IDE Festplatte bestückt ist. Hinsichtlich der Auswertung standen zwei Hauptaspekte im Vordergrund: die Wirksamkeit der Energieabrechnungs- und Energieallokationsmechanismen sowie die allfälligen Leistungseinbußen, welche die Benutzung dieser Mechanismen nach sich zieht. Die Genauigkeit der internen Energieabrechnung wurde jeweils mit Hilfe externer Messungen des Energieverbrauchs von Prozessor und Festplatte überprüft, welche mit einem digitalen Erfassungsgerät von National Instruments erfolgte.

Die Wirksamkeit der Energiemanagementmechanismen wurde anhand verschiedener Experimente zur Genauigkeit der Energieabrechnung und Energieallokation überprüft. Dabei wurden sowohl das Energiemanagement ganzer virtueller Maschinen als auch einzelner virtualisierter Anwendungen unter die Lupe genommen. Experimente mit der Energieabrechnung eines virtualisierten Festplattendienst förderten zu Tage, dass der Pfad von virtuellen Anwendungen hinab zu den phyischen Geräten getreu und in hoher zeitlicher Auflösung nachverfolgt werden kann, und individuelle Energieverbrauchsdaten für die jeweils beteiligten Module und Anwendungen ermitteln werden können, welche überdies kumulativ mit dem extern gemessenen Energieverbrauch von Festplatte und Prozessor übereinstimmen. Weitere Experimente mit der Energieallokation für Prozessor und Festplatte brachten ans Licht, dass die vorgeschlagenen Mechanismen eine individuelle Budgetierung einzelner Anwendungen und virtueller Maschinen seitens ihres Energievebrauchs erlauben. Wiederum stimmten intern gemessene Energiebudgets mit dem jeweils extern gemessenen Energieverbrauch von Festplatte und Prozessor überein.

Mögliche Leistungseinbussen wurden anhand weiterer Experimente ermittelt, welche die Effizienz des Prototypen mit der des originalen mikrokernbasierten Modulbetriebssystems ohne Unterstützung für Energiemanagement verglichen. Die Resultate untermauern den postulierten Zusammenhang zwischen der Art der Kommunikation, der zeitlichen Genauigkeit und der Effizienz des Energiemanagementprozesses. So ergab sich im Falle des Prozessor-Energiemanagements eine dramatische, zehnfach große Leistungseinbuße für ein ausgewähltes Microbenchmark-Szenario, was sich aber weit moderater auf der Anwendungsebene niederschlug, wo sich die Leistungseinbußen je nach gewünschter Genauigkeit der Energiemanagementmechanismen zwischen 0 und 36 Prozent für ausgewählte Benchmarks bewegten. Im Falle des kombinierten Energiemanagements von Prozessor und Festplatte konnte keine Änderung des Festlattendurchsatzes festgestellt werden und die Leistungseinbußen blieben auf den erhöhten Bedarf an Prozessorleistung beschränkt. Der erhöhte Rechenbedarf bewegte sich dabei absolut gesehen auf niedriger Ebene, war aber relativ gesehen durchaus signifikant: so stieg der Bedarf von 5.1 auf maximal 7.6 Prozent, was einem relativen Anstieg um maximal 49 Prozent entspricht. Wiederum hingen die tatsächlichen Werte von der gewünschten Genauigkeit der Energiemanagementmechanismen ab. Im Lichte der vorläufigen, prototypischen Implementierung kann man davon ausgehen, dass eine Optimierung der Mechanismen eine weitere Reduktion der Leistungseinbußen mit sich zöge. Festzuhalten ist aber auch, dass das Postulat der negativen Korrelation zwischen Genauigkeit und Effizienz des Energiemanagementprozesses untermauert weden konnte, was im Gegenzug wiederum für das in dieser Arbeit vorgestellte Konzept zur flexiblen Gestaltung der Kommunikation im Energiemanagementprozess spricht, welche eine Anpassung des Kommunikationsprotokolls an die jeweiligen Energiestrategien erlaubt.

Portions of this work were previously published in the following papers and articles:

- Jan Stoess, Christoph Klee, Stefan Domthera, and Frank Bellosa. Transparent, Power-Aware Migration in Virtualized Systems. In *Proceedings of the GI/ITG Fachgruppentreffen Betriebssysteme*, Karlsruhe, Germany, October 2007.

- Jan Stoess. Towards Effective User-Controlled Scheduling for Microkernel-Based Systems. In *ACM SIGOPS Operating Systems Review*, 41(4), July, 2007.

- Jan Stoess, Christian Lang, and Frank Bellosa. Energy Management for Hypervisor-Based Virtual Machines. In *Proceedings of the 2007 USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

- Jan Stoess, Christian Lang, and Marcus Reinhardt. Energy-aware Processor Management for Virtual Machines. In *Poster session of the 1st ACM SIGOPS EuroSys Conference*, Leuven, Belgium, April 2006

# Contents

# List of Figures

# Chapter 1

# Introduction

For decades, research and industry have focused on performance as the determining factor for designing computers. Recently, however, energy efficiency has become another serious challenge for computing environments. The desire for energy-efficient computing has two root causes. First, there is an ongoing trend in computing towards mobility, and computer designers are confronted with the challenge to deliver ever-increasing performance under conditions of constrained energy. Second, computer designers face a rapid growth in energy demands of computer systems, particularly in server systems, and with it an explosion of associated costs. The main reason for such high energy demands is the increasing power density of today's computer architectures; cooling requirements aggravate the energy demands additionally.

It has therefore become general consensus among researchers and practitioners that energy efficient computer design is a first-class architectural design constraint. For mobile computing platforms, the most important objective of energy management is to extend battery lifetime; in contrast, the common goal for server systems lies in reducing operational costs and in overcoming the limitations of higher power densities. Hence, "energy-efficient" computing has several different goals, among which we have identified four to play key roles: First, computers should *consume less power*, to enhance usability of mobile devices, and to reduce the general electricity costs and the ecological footprint of computing; second, computers operate at *lower temperatures*, to prevent the cooling requirements and costs from skyrocketing; third, computers should avoid *power and temperature peaks* where possible, as avoiding such peaks allows cooling infrastructure and electricity capacities to be designed for average scenarios, rather than to be over-provisioned for cumulated worst-case scenarios. Fourth, computers should provide a notion of *application-centric energy management*, as involving applications allows energy management to be combined with other notions such as quality-of-service, fairness, and billing, all of which are crucial for computer users.

Traditionally, computer designers have striven for energy efficiency mostly at the hardware and facilities layer, for instance, by designing low-power architectures or by improving air conditioners and data-center thermodynamics. Energy-efficient computing, however, is a challenge that needs to be addressed by all involved components, including the operating system. In fact, the operating system plays a crucial part in the design methodology, as it is responsible for controlling both computer hardware and applications at run time. Implementing energy management at operating-system level thus enables global and dynamic energy management, encompassing all hardware devices and software activities together.

As of now, there exists a large bottom of approaches to involve operating systems in computer energy management. Such approaches typically employ various monitoring and control loops for hardware devices and their power or heat states; likewise, such approaches implement some form of monitoring and control of applications, as those are ultimately responsible for device load and energy consumption. Based thereon, some management policy then strives to align utility interests with energy-efficiency goals; that is, the policy follows the goal of delivering performance at a given (or maximum) level, but with the lowest power consumption, the lowest heat generation, or the fewest power and heat spikes possible. Or, conversely, the policy tries to satisfy requested power or heat constraints while providing as much service and as much performance as possible.

At the same time energy efficiency has become a challenge for modern computing and operating systems, there has been a a growing trend in the recent past, in research and practice, towards modular operating-system designs with a small kernel base. Traditional operating-system design separated the software stack into two layers: user level and kernel level. Applications ran concurrently at user level, while the operating system encompassed all privileged and service functionality in a single kernel image. Early operating-system kernels were rather small in size, and their functionality was rather limited. With increasing capabilities and capacities of computers, however, kernel size and complexity increased dramatically; a modern monolithic kernel is built from millions of lines of source code, containing a huge conglomerate of complex and entangled functions.

A growing diversity of application domains has contributed to the size and complexity growth, as modern operating systems are typically designed to run in different deployments such as the server, desktop, and embedded space. In combination with large kernel sizes and complexity, however, monolithic design leads to severe structural problems. The two most burdensome among them are: limited flexibility, which stems from the lack of properly defined interfaces and module boundaries in a single kernel image; and limited reliability, which is caused by the monolithic kernel structure, where all kernel code, including device drivers and other, potentially error-prone parts run within the same, privileged kernel domain.

To alleviate the problems of monolithic kernel design, researchers have proposed and explored different solutions. In this work, we focus on a specific subset of those approaches, which we refer to as the paradigm of *modular operating systems*. In short, a modular operating system performs all its tasks in independent modules, which interact via dedicated and well-designed interfaces only. Modules are confined in isolated domains with different privileges, permitting flexible system design and containment of faults. Concept-wise, modular operating systems are a long-standing idea. Only recently, however, have modular operating systems emerged to become prevalent also in practice, and have become of widespread use in different deployments such as the embedded-systems or the server space. Typical embodiments come as a small kernel and a set of service modules running atop, in deprivileged user level. They provide custom abstractions to make the system extensible to new application scenarios; at the same time, they have integrated virtualization capabilities to remain compatible to existing applications.

## 1.1 The Problem

There has been a considerable research interest in involving operating systems in the energy management of a computer system. So far, however, most of these approaches have been targeting standard, monolithic operating-system structures. Monolithic kernels pack all their functionality and data in a single image; they have full and exclusive control over both hardware devices and applications. As a fundamental consequence, their energy-management subsystem becomes centralized and monolithic as well. Centralized operating-system energy management is well suited for monolithic kernels: in charge of controlling all devices and the whole application flow in the system, the kernel level can directly monitor and control both software and hardware in order to meet the thermal and energy constraints imposed by the management policies.

Modular operating systems, however, go beyond the simple application–kernel world in their structure, rendering direct and centralized energy management unfeasible. Modular operating systems consist of a distributed and multi-layered software stack, typically with a small kernel base and multiple service modules running atop, at user level. With such a structure, control and information is distributed: the privileged but small kernel has direct access to essential parts of the hardware only; all other operating-system services, including device drivers, run as user-level modules, with reduced privileges and limited scope. Multiple notions and granularities of applications and resource principals co-exist, and are catered for and controlled by different operating-system modules or layers.

In such an environment, direct and centralized energy management is unfeasible, as control and accounting information of devices and applications are distributed across the whole system. Three main problems for energy management arise from the scattering and isolation of device and application control:

First, semantic loss arises between modules in terms of energy-management–related information. With an operating system partitioned into multiple modules, resource management becomes partitioned as well, and multiple resource-management subsystems may reside in different modules, each with different semantics and mechanics. As a result, the information relevant for energy management — device power or heat states, application behavior and their effects on power consumption, and so on — must be shared and structured explicitly. Legacy compatibility and virtualization play an important role, as they often limit the design space of protocols and interfaces for semantic sharing between modules.

Second, the communication and interaction required for energy management become increasingly important and challenging in terms of their associated costs. In a modular operating system, the components taking part in the energy management now reside in separate software modules, with boundaries between them for isolation purposes. The isolation comes at the cost of overhead for crossing module boundaries; whereas monolithic kernels can rely on memory procedure calls and memory loads and stores to transfer control and data relevant for energy management, modular operating systems cannot use such direct mechanisms to retrieve the same information, but must resort instead to explicit communication primitives overcoming the protection boundaries. As a result, energy-management communication must be thoroughly designed and engineered in modular operating systems, rather than performed directly and ad-hoc as done in monolithic kernels.

Third, in modular operating systems, dependencies and resource paths from applications down to hardware become more complex. Hardware management is partitioned among multiple drivers, and applications can have different types and granularities; coarse-grain applications (a virtual machine, for example) can consist of multiple sub-applications (applications within a virtual machine, for example). As a result, resource-management dependencies and paths become increasingly complex, and with them their effects on energy consumption and heat generation. Application-specific energy management, however, must deal with the different types and granularities of applications, and with distributed resource-management subsystems.

While numerous research efforts have been conducted on operating-system energy management, so far, none of them has addressed the specific problem of providing system-level energy-management support in modular operating systems. Most of the existing energy management approaches were designed towards a traditional, monolithic operating-system structure; as such, they were safe to disregard operating-system modularization at all. We conjecture that many of those approaches, particularly their policies and algorithms, will be valid and applicable in modular operating systems in principle — provided system-level support for modular energy management is in place.

Substantially fewer approaches to operating-systems energy management have targeted an operating system that was modularized in some kind. Some efforts have investigated how energy pricing and accounting can be facilitated in so-called vertically-structured systems, which multiplex all resources at a low level, and move protocol stacks and most parts of device drivers into user-level libraries; such systems can be termed modular, albeit the modularization occurs in vertical direction only. Vertical structuring, however, implies that devices are shared at a low level only, and renders it hard to achieve energy accounting and budgeting in presence of shared drivers and operating-system services, as they are common in other modular operating systems such as microkernel-based systems. Several other efforts have investigated how energy management can be designed for virtualized operating systems, another, popular form of a modular operating system. While those approaches share many goals, insights, and ideas with our own work, they are typically focused on virtualization systems only, and do not address the problems and specifics of other instances of modular operating systems, such as microkernel-based systems. Finally, some efforts have been conducted to explore energy-management solutions for microkernel-based operating systems. As those approaches typically target on mobile computers, they heavily focus on extended battery lifetime, but less on other energy-management problems such as accounting and budgeting; also, they do not consider the problems arising if a virtualization layer is integrated into the operating-system stack.

## 1.2 Approach

Observing these problems, we present a novel system-level framework for managing energy in modular, multi-layered operating-system environments, as they are becoming common in today's computer systems. Our framework strives to enable energy awareness and energy management if the resource-management subsystem is distributed and scattered among operating-system modules, rather than being centralized and monolithic. We envisage a framework that enables flexibility in energy-management algorithms and objectives; that provides efficient support for modularity and isolation of energy-management subsystems; and that enables customization of the operating system to new application domains and energy-management paradigms, but, at the same time, allows to preserve compatibility to existing and legacy applications.

Our framework makes the following **key contributions**:

**A Model for Modularization-Aware Energy Management** We model the operating system as a set of modules; the energy management becomes a feedback loop involving one or more of such modules. Our model solely relies on the notion of energy as the base abstraction, since energy quantifies power and thermal effects in a partitionable, distributable, and convertible way.

**Exposed and Distributed Energy Accounting** We propose a distributed energy-accounting approach, which accurately tracks back the energy spent in the system to originating applications and other activities. Our approach incorporates both the direct and the side-effect energy consumption spent along the path from different notions of applications down to hardware devices.

**Exposed and Distributed Energy Allocation** We further propose to expose suitable energy-allocation mechanisms from drivers and other resource managers to respective energy-management subsystems. Exposed allocation enables dynamic and remote regulation of energy consumption throughout the modular operating system, across module boundaries and isolation domains.

**Energy-Management Interaction Protocols** We explore two different schemes for carrying out the communication that is required to propagate energy-management–related information between modules. We postulate that the optimal protocol is largely defined by two (opposing) factors: the timeliness requirements of the energy-management policy, and the performance overhead induced by the module isolation. We thus explore two interaction schemes, which differ in their synchronizity with respect to the propagation of energy-management data.

**Virtualization of Energy Effects** Finally, we support energy management not only for physical devices but also for virtual devices. Platform virtualization is a widely used and convenient mechanism to provide legacy compatibility and has found its way into many modern operating systems. Supporting a notion of virtual energy thus provides a convenient development path towards legacy-compatible but energy-aware resource management.

## 1.3 Validation

To demonstrate our approach we have developed a prototype for a microkernel-based modular operating system. We use an instance of the L4 microkernel family as the privileged microkernel. Our system permits the development of custom-built operating-system modules using L4 abstractions and mechanisms; at the same time, it retains legacy compatibility to existing applications by means of a platform virtualization layer. Our environment runs on IA-32 hardware and supports Linux 2.6.9 guest operating systems. Our prototype system forms an instance of a modular operating system that can be practically used in modern computing environments. As such, we consider it a good and realistic candidate for evaluating our energy-management design principles and considerations.

Our prototype employs energy management at several layers and levels of granularity. It currently supports management of two main energy consumers, processor and disk. Processor services are directly provided by the L4 kernel, while the disk is managed by a special user-level driver. Our prototype supports energy management both for physical and for virtual processors and disks. To that end, it features a distributed and recursive energy-management mechanisms for native microkernel modules, for complete virtual machines, and for individual applications running within a virtual machine. Our energy-management infrastructure is extensible, and allows management policies to be adapted to the actual workload or deployment situation without having to exchange the mechanisms themselves.

We have evaluated our approach and prototype implementation on an IA-32 platform equipped with a Pentium D 830 with 3 GHz, an Intel 945G motherboard with 2 GBytes RAM, and a Maxtor Diamond Max Plus 9 IDE hard disk with 160 GBytes in size. We validated our internal accounting mechanisms by means of an external high-performance data acquisition system, which we used to measure the real disk and processor power consumption. For evaluation, we considered two aspects as relevant: first, we were interested in the effectiveness of energy accounting and allocation; second, we were interested in the performance overhead induced by those mechanisms.

To evaluate the effectiveness of our energy-management framework, we pursued several experiments validating the accuracy of our energy accounting and allocation mechanisms with respect to both whole virtual machines and individual virtualized applications. Experiments with a virtualized disk service show that our infrastructure is able to accurately track down all portions of energy consumption from a virtualized application down to the hardware device, and to attribute the portions to the originating software activities. All internal accounting records furthermore correspond with the actual, externally measured processor and disk energy consumption. Experiments with our processor and disk energy allocation mechanisms, which we pursued both at host level and at the level of virtualized applications, demonstrate that our prototype is capable of accurately allocating individual portions of processor and disk energy to different notions of applications. Again, all internal records correspond with real processor and disk power consumption.

To evaluate the performance overheads associated with our prototype framework, we pursued further experiments comparing the performance of our prototype against a version without support for energy management. The results highlight the interplay between accuracy and efficiency of energy management:

overheads for processor energy management may be dramatic, up to factor 10x, for a microkernel IPC benchmark, percolating however to application level at a more moderate level, where overheads range from 0 to 36 per cent for selected application benchmarks, depending on the desired accuracy of scheduling. Overheads for combined processor and disk management are zero with respect to the disk throughput, and limited to an increase in processor utilization. The increase occurs at a low level in absolute terms, but is still significant relatively: utilization rises from 5.1 to at most 7.6 per cent, that is, by at most 49 per cent. The actual overheads again depend on the desired accuracy of scheduling. The evaluation supports our observation that there is a fundamental trade-off between accuracy and performance in energy management. We draw the conclusion it should be up to developers of energy-management policies, which point in the trade-off space to choose; our energy-management infrastructure strives to enable those different decisions to be put in effect.

## 1.4 Organization

The rest of this thesis is structured as follows: Chapter 2 reviews background material and related work in the context of our approach. Chapter 3 explains the design concepts and rationale behind our approach. Chapter 4 illustrates in detail our prototype implementation. Chapter 5 presents the evaluation of our prototype. Chapter 6 summarizes our approach and presents our conclusions.

# Chapter 2

# Background and Related Work

In this chapter, we present background material and related work in the context of our approach. The chapter is organized as follows: In Section 2.1, we present an overview and motivate, why energy management has become a serious challenge for today's computer systems. In Section 2.2, we explain why the operating system plays a crucial and important role in the design of energy-efficient computing. In Section 2.3, we explain why there is an ongoing trend in modern operating-system structures towards modularization and layering. Subsequently, in Section 2.4, we detail why approaches to energy management for traditional operating-system structures fall short and energy management becomes challenging in modularized operating-system structures. Finally, in Section 2.5, we will review and discuss concrete approaches to supporting energy management in both traditional and modularized operating systems, and relate them to our own approach.

## 2.1 Overview

For decades, research and industry have focused on performance as the determining factor for designing computers. Recently, however, energy efficiency has become a serious challenge for computing environments. The desire for energy-efficient computing has two root causes. First, there is an ongoing trend in computing towards mobility, and computer designers are confronted with the challenge to deliver ever-increasing performance under conditions of constrained energy. Saving battery life simply enhances the time span between consecutive battery charge cycles and thus directly benefits the users of mobile devices. Second, computer designers face an increasing hunger for energy in computer systems, especially in server systems, and with it an explosion of associated costs. The energy demands of computer systems are growing rapidly, in absolute terms and in proportion to the operational costs. According to recent studies, the power consumption of a typical server is estimated to have increased by nearly a factor of ten between 1996 and 2006 [Ranganathan et al., 2006]; and the costs for electricity and cooling can represent as much as half of the total costs of the data-center operating budget [Filani et al., 2008, Rasmussen, 2006].

The main reason for the growth in energy demands is the increasing power density of today's computer architectures, which, in turn, stems from high-density chip designs as well as from high-density packaging and integration architectures. Cooling requirements aggravate the overall demand for energy: as increasing power densities lead to a higher probability of thermal fail-overs of the hardware, additional cooling facilities are required to sustain reliability of the computing infrastructure. Cooling facilities contribute a large part to the recurring energy costs: according to recent studies, in about 85 percent of today's data centers, every watt consumed for the actual computer equipment requires another watt for cooling and and other support infrastructure [Malone and Belady, 2006].

Although there have been substantial advances in the design of energy-efficient computing, to date, they have not been able to alleviate much of the simultaneous increase in energy hunger: while power consumption per computational unit has dropped by 88 percent between 2000 and 2006, the at-the-plug consumption has still risen by a factor of 340 percent [Brill, 2007]. Nevertheless, it is now general consensus among researchers and practitioners that energy efficiency and energy management have become a first-class architectural design constraint [Mudge, 2001]. For mobile computing platforms, the most important objective is to extend battery lifetime [Benini et al., 2000, Welch, 1995]; in contrast, the common goal for server systems lies in reducing operational costs and in overcoming the limitations of higher power densities [Bianchini and Rajamony, 2004, Lefurgy et al., 2003]. Hence, "energy-efficient" computing has several different aspects, and we break up the the general goal into four major objectives constituting it:

**Lower Power Consumption** First and foremost, computers should *consume less power*, for several reasons. Naturally, low power consumption helps to increase battery lifetime thus enhances usability and value of mobile devices when not plugged into an outlet. However, low power consumption is also beneficial for economic and ecological reasons: power consumption required for information technology is estimated to account for 8 percent of the overall power demand in the United States [Mudge, 2001]; for many industries, data centers are one of the largest sources of greenhouse gas emissions [Forest, 2008]. Power-efficient computing reduces both the electricity costs for the infrastructure provider and the global impact of the information-technology business on the ecological footprint.

**Lower Operating Temperatures** Second, computing hardware should operate at *lower temperatures*, to reduce the requirements for cooling capacity and to prevent cooling costs from skyrocketing [Kerby, 2007, Skadron et al., 2003a]. Low operating temperatures furthermore reduce the heat density of computer infrastructure and thus allow denser packaging and feature integration without inducing higher risks for thermal rises.

**Fewer Power and Temperature Peaks** Third, computer hardware should avoid *power and temperature peaks* where possible. Although the chance for individual peaks may be low, total power ratings of computer systems are usually computed as the sum of individual worst-case ratings; keeping power consumption steady and avoiding power peaks thus allows electricity budgets and contracts to be based on average rather than on peak power consumption. Similarly, avoiding temperature peaks allows cooling infrastructure and capacities to be designed for average thermal scenarios, rather than to be over-provisioned for cumulated worst-case scenarios [Felter et al., 2005, Ranganathan et al., 2006].

**Better Accounting and Budgeting of Energy-Related Effects** Fourth, and finally, computer systems should involve applications and workloads as the root cause and consumers of computer energy. Accounting and budgeting play a key role in this regard: accountable computing allows energy-management goals to be combined with quality-of-service requirements of applications and the workload [Femal and Freeh, 2005, Ranganathan et al., 2006]. Furthermore, accounting enables fair billing of energy-related costs on the basis of individual applications and customers. Energy accounting and budgeting typically requires monitoring power characteristics at two sources, the computer hardware components and the workload and users executing on the computer [Microsoft Corporation and Intel Corporation, 2009].

## 2.2 Energy Management in Operating Systems

Traditionally, computer designers have striven for energy efficiency mostly at the hardware and facilities layer, for instance, by designing low-power architectures or by improving air conditioners and data center thermodynamics. A substantial amount of effort has been put into low-power circuit design, including research on power-efficient processors and cores, main memory, and storage systems (e.g., [Benini et al., 2000, Mudge, 2001]). Likewise, there exist numerous efforts to improve the energy efficiency of the facilities hosting the computer infrastructure and their integrated cooling equipment (e.g. [Fan et al., 2007, Sullivan, 2000]). Ideally, however, the problem should be addressed holistically at all system levels, including the operating system.

Indeed, for energy-efficient computing, the operating system plays a crucial part in the design methodology. As the software component responsible for both computer devices and the applications running atop, the operating system is the ideal place for controlling, monitoring, and coordinating energy conditions and states on both the hardware and the software side of the computer. Implementing an energy-management subsystem for the operating system thus enables *global and coordinated* management encompassing all devices and activities in the computer; *dynamic and reconfigurable* management, adapting to varying workload conditions and adhering to energy efficiency goals at runtime; and, finally, *accountable* management, tracking back energy consumption, heat generation, and the associated costs to originating activities respectively the customers running them.

In fact, there has been a growing research interest in involving the operating systems in the management of computer energy. Facing both performance and energy efficiency as primary design goals, approaches to energy management at operating-system level typically attempt to integrate both performance and energy considerations into their resource-management process: operating-system energy management then either follows the goal of delivering performance at a given level, but with the lowest power consumption, the lowest heat generation, or the fewest power and heat spikes possible. Or, conversely, it tries to satisfy requested power or heat constraints while providing as much service and as much performance as possible [Isci et al., 2006]. Best-effort operating systems usually adhere to the former goal; however, electricity budgets, utility contracts, or cooling facilities provisioned for worst case scenarios render the latter objectives equally or even more important. In any case, the following two tasks are crucial for operating-system power management:

**Controlling and monitoring hardware devices** Naturally, operating-system power management must both control and monitor the energy conditions of the underlying hardware devices. The control part comprises of manipulating device power and performance states at runtime, in order to reduce the power consumption or temperature. An increasing number of modern computer devices feature such capabilities to control or reduce power and heat dissipation: modern micro-processors, for example, provide different frequency and voltage settings, or other mechanisms such as clock gating or sleep states, enabling the operating system to dynamically change processor power and heat generation. Likewise, modern memory controllers and peripheral network and storage devices often support different performance and standby states with different resulting power consumption and temperature. The monitoring part comprises of observing device power and thermal conditions on a regular base; secondary parameters such as load and general device usage characteristics are often relevant as well. While devices have become fairly mature in terms of *control*, they are often less equipped in terms of *monitoring*. Many modern computers feature thermal sensors, but often only a few of them, sometimes at places in the chassis not necessarily related to the location of devices; also, the sensors are typically thermal diodes, which are slow to read out (in the order of hundreds of milliseconds, a substantial amount of time for an operating system). Specific energy meters for individual hardware components are mostly missing or unavailable to software, rendering hardware-supported energy monitoring on a per-device base unfeasible. Research has therefore responded with estimation schemes for hardware power effects (e.g., [Bellosa et al., 2003, Heath et al., 2006]); the schemes typically rely on device models and on second-order characteristics such as load and usage patterns to approximate per-device power and heat conditions.

**Controlling and monitoring workload** In addition to hardware control, operating-system energy management must also control and monitor the workload and applications running atop, for three reasons: First, dynamic energy-management schemes often attempt to adapt hardware power states to the expected load situation; since applications ultimately create hardware load, monitoring application behavior plays an important role for predicting future load conditions. Second, energy management can actively adapt workload behavior and request patterns in order to impact hardware energy effects. Typical examples for such active mechanisms employed in operating systems are to throttle, batch, or delay individual requests in order to lengthen periods of idleness and underutilization, or to multiplex and balance requests across multiple devices of the same type. Sometimes, direct hardware control even remains unavailable to the operating system; rather, the hardware controller turns the device off or into low-power modes during phases of underutilization automatically (as done in memory controllers [Fan et al., 2001]). In those cases, software control is the only means to influence the energy consumption of a device. Third, and finally, workload control and monitoring is the prerequisite for accounting and budgeting of energy-related effects. Only the operating system can attribute energy consumption and heat generation to originating applications, and only the operating system has sufficient control over schedules to guarantee that quality-of-service constraints and service-level agreements can be negotiated on the bases of individual customers or applications.

To summarize, coordinated and dynamic energy management at operating-system level requires monitoring power, heat, or other status information of both computer devices and of applications. It further requires mechanisms to directly control device power states as well as the applications generating device load. Based on such information and control, operating-system energy management can strive to adhere to energy-management objectives while still adhering to quality-of-service constraints or service level agreements; or, conversely, energy management can attempt to maximize performance while adhering to energy-management constraints. Energy management can finally support accountable computing, where energy consumption and heat generation are attributed and billed to originating customers and their applications.

## 2.3 Modularization in Operating Systems

At the same time power and thermal management have become a challenge for modern computing and operating systems, there has been a a growing trend in the recent past, in research and practice, towards modular operating-system design with a small kernel base.

Traditional multi-programming operating systems separate the software stack into two layers, user level and kernel level. Applications run in unprivileged user mode, executing arbitrary code (applications, applets, plug-ins) coming from arbitrary sources (vendors, software distributors, the Internet). To enable applications to run concurrently but isolated from each other, and to provide safe and coordinated access to hardware devices, the operating system runs in privileged processor mode and protected from the applications. The operating system is activated whenever applications demand privileged service or access to hardware. The operating system may also become active independently from applications, for instance to handle device requests or to interrupt the current application and execute another one.

Early operating-system kernels were rather small in size and functionality, mainly because computers were small in size and their capabilities as well. With increasing capabilities and capacities of processors, memory, and peripheral devices, however, kernel sizes have increased dramatically. A modern UNIX or Windows kernel for a recent computer system is built from millions of lines of source code, containing a huge conglomerate of complex and entangled functions, with modules for various operating-system tasks such a process management, device handling, file systems, user-interface management, networking, process communication, and so on. Along with the monolithic design and the growth of kernel size, however, have come severe difficulties: since the operating system is packed into a single kernel image, its growing size and functionality has lead to a complex and entangled piece of code, which is suffering from fundamental problems in flexibility and reliability:

**Limited extensibility** Monolithic operating systems are complex and entangled, and the lack of properly defined interfaces and module boundaries negatively affects their extensibility and maintainability. Once an abstraction or its implementation has been established in a particular kernel, it becomes tedious and expensive to change or remove it [Engler et al., 1995]. The result is that monolithic operating systems generally advance very slowly, since enhancements and optimizations are hard to integrate; for example, many of the core kernel primitives of UNIX operating systems, notably processes, virtual memory, or inter-process communication, have existed for decades, and have not changed significantly throughout particular UNIX revisions and flavors.

**Limited Reliability** All code of a monolithic operating systems runs in privileged processor mode; its sheer size and complexity adversely impact system reliability and dependability; any mistake in any subcomponent can bring down the whole operating system [Swift et al., 2003,Tanenbaum et al., 2006]. As device drivers constitute a large fraction of the operating system code base and show error rates up to seven times higher than other parts of the software infrastructure [Chou et al., 2001], the problem of poor fault isolation and containment is particularly problematic at operating-system level.

The negative implications of lacking extensibility and reliability has been aggravating with the growing diversity in fields of application for standard operating systems: modern operating systems such as Linux or Windows are typically deployed in multiple application domains simultaneously, such as the server, desktop, embedded space.

To alleviate the problems of monolithic kernel design, researchers have proposed and explored different solutions, including approaches to add enhanced reliability subsystems to the operating system [Ng and Chen, 1999,Swift et al., 2003] or to use type-safe languages and compilers [Bershad et al., 1995, Seltzer et al., 1996]. In this work, we focus on a different subset of approaches, which we refer to as the paradigm of *modular operating systems*. In short, a modular operating system performs all its tasks in independent modules, which interact via dedicated and well-designed interfaces only. Modules are confined in isolated domains with different privileges, permitting flexible system design and containment of faults (Figure 2.1).

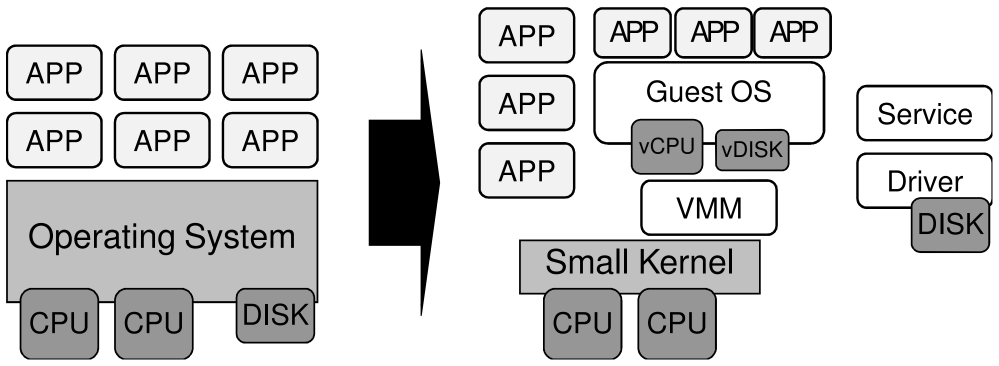

Figure 2.1: The trend to modular operating system structures, with a small privileged kernel and isolated operating-system modules and drivers atop. To stay compatible to existing legacy applications, modular operating systems often employ some form of virtualization.

Concept-wise, modular operating systems are a long-standing research idea, with their roots in early multiprogramming computer systems [Hansen, 1970, Wichmann, 1968]. However, while an old idea from a standpoint of theory and research, only recently modularization has been successfully employed in modern real-world operating systems with their vast functionality and their varying fields of application. In the area of embedded systems, where reliability and trustworthiness are major concerns, microkernel-based modular systems have been of widespread use for years — traditionally, however, as special-purpose real-time operating systems [Hildebrand, 1992, Green Hills Software, 2009]. With the growing capabilities of modern embedded hardware and consumer electronics, however, modern operating-system concepts and virtualization technology have become important factors for embedded operating systems as well [Heiser et al., 2007, Heiser, 2009]. This trend has recently lead to commercial, microkernel-based hypervisor systems for the embedded space, such as OKL4 [Open Kernel Labs, 2009] and VMware MVP [VMware Inc., 2009b], which are successfully deployed on millions of mobile phones. In the area of server systems, operating-system modularization has also become mainstream technology in the last few years, mainly in form of hypervisor-based virtualization environments. Such environments have been deployed successfully on millions of enterprise computing systems worldwide over the last few years, with different products such as VMware ESX [VMware Inc., 2009a], Citrix XenServer [Citrix Systems Corporation, 2009], or Microsoft Hyper-V [Microsoft Corporation, 2009] offered on the market.

In general, the main characteristics of such modern and practical modular operating systems are:

**Small Kernel Base** A widely accepted principle to improve flexibility and reliability of operating systems is to keep the amount of code running in the privileged kernel to the possible minimum [Accetta et al., 1986, Engler et al., 1995,Liedtke, 1995]. Small kernels enforce the remaining operating-system functionality to be constructed as user services, like other applications; also, they contain fewer bugs and are therefore easier understood, fixed, and validated [Heiser, 2005, Klein et al., 2009, Tanenbaum et al., 2006]. While a minimal kernel — or microkernel — and its interface do not *mandate* a modular or reliable system structure on top [Härtig et al., 1997], they clearly suggest it; indeed, many modular operating systems of practical relevance run on top of some sort of microkernel [Heiser, 2005, VMware Inc., 2009a]. Minimizing the kernel part thus permits a modular and flexible system structure, and enables a higher level of assurance and reliability.

**Confinement and Isolation** Another important design principle is that a modular operating system should allow executing its modules and components in isolation. Confinement and isolation can be done in different ways; typically, the kernel provides fundamental isolation abstractions, either by means of hardware protection domains or by means of a software-protected runtime system. Widely used hardware mechanisms are memory address spaces and processor privilege modes [Herder et al., 2006, Liedtke, 1995]; typical representatives of software protection domains are safe languages [Hunt and Larus, 2007] and virtual machines [Adams and Agesen, 2006]. Depending on the particular design of the modular operating system, the isolation results in different system structures: multi-server systems such as Sawmill [Gefflaut et al., 2000], K42 [Krieger et al., 2006], or Minix [Herder et al., 2006] arrange the system as a set of individual user servers that interact with clients and with each other to provide the operating-system functionality. In contrast, vertically structured systems such as ExoKernel [Engler et al., 1995], Nemesis [Leslie et al., 1996], or, similarly design-wise, virtual- machine environments [Barham et al., 2003] arrange the system in multiple layers and move the functionality into protocol stacks and user-level libraries.

**Legacy Compatibility** Although much effort has been invested in their design and development, modular operating systems were not generally accepted as a practicable methodology for general-purpose system construction for a long time. Besides the lack of performance of early versions, which could be overcome over time [Härtig et al., 1997] but still contributed to negative perception, modular operating systems suffered of the fundamental compatibility problem any new operating-system endeavor suffers: applications, libraries, and other system services must be ported to the new infrastructure. Compatibility, however, is a fixed requirement for an operating system to become prevalent [Krieger et al., 2006]. Over time, the sluggish advancements of traditional operating systems and the boot-strap dilemma of advanced operating systems have provoked the emerge of platform virtualization as a solution. Originally designed to allow time sharing and logical partitioning for mainframe computers [Goldberg, 1972], virtualization conveniently solves the compatibility problem by providing one or more identical copies of the underlying hardware and enabling arbitrary applications (written for a particular hardware platform) to be executed without or with very few modifications. Virtualization layers have become very popular and the de-facto standard to solve legacy-compatibility problems; they have been integrated into commodity operating systems [Kivity et al., 2007, Sugerman et al., 2001] and into modular operating systems [Barham et al., 2003, Härtig et al., 2005, Heiser, 2009].

To summarize, the trend in operating-system design is moving away from monolithic, heavy-weight towards modular structures; modern operating systems often come as a small kernel and a set of service modules running in deprivileged user mode. They have integrated virtualization capabilities to remain compatible to existing applications, but simultaneously provide custom abstractions to make the system extensible to new application scenarios.

## 2.4 Energy Management and Modular Operating Systems

Although there has been a considerable research interest in operating-system energy management (Section 2.5 will review concrete approaches in detail), so far, most such schemes have been targeting standard, monolithic operating-system structures. Monolithic kernels pack all their functionality and data in a single kernel image; they have full — and exclusive — control over both hardware devices and applications. As a fundamental consequence, their energy-management subsystem becomes centralized and monolithic as well. We postulate that direct and centralized energy management, as done in monolithic operating systems, is an unfeasible approach for modular operating systems. This paragraph is devoted to a discussion on that matter. We first describe the main characteristics of centralized energy management; we then describe why such centralism becomes unfeasible in modular operating-system structures.

### Management Centralism in Traditional Operating Systems

Energy management can be seen as special case of resource management. Resource management is concerned with multiplexing hardware resources to applications as the resource principals; energy management is concerned with doing so in a *energy-aware* way. A central aspect of monolithic operating systems is that resource management — and thus energy management — occur in *direct* and *centralized* fashion, both with respect to applications and with respect to hardware devices.

Since the kernel instance executes in privileged operating mode, it has direct control over hardware devices and their modes of operation. Co-located with the kernel, the resource-management subsystem can directly inspect power, performance, or temperature conditions of the devices, and it can directly control their power and performance states. The kernel also has direct control over the application stack, and is capable of performing energy management on the basis of individual applications. To that end, it can monitor and control individual applications at runtime in order to meet energy management and performance goals.

A monolithic kernel is not only privileged, it is also centralized in resource management: the kernel is *the only* entity responsible for *all* applications and *all* devices. Moreover, typically the monolithic operating system shows a fairly straight-forward application structure, where the kernel provides some unified abstraction — a process, for instance — for both isolation and resource management [Banga et al., 1999]. The kernel controls all applications and processes, carrying out their requests on the available hardware devices. Its resource-management subsystem is centralized and comprehensive, capable of controlling all hardware and all applications in unison.

### Distributed Resource Management in Modular Operating Systems

Obviously, direct and centralized resource management are unfeasible in modular operating systems. A modular operating system goes far beyond the simple application–kernel world in its structure, consisting of a distributed and multilayered software stack, with a small kernel base, multiple operating-system service modules at user level, and possibly multiple notions and granularities of applications and resource principals. At the lowest-level of the environment, the privileged but small kernel has direct control over those parts of the hardware that are essential to providing safe and secure isolation and resource management; usually, the kernel manages, at the very least, access to processor, physical memory, and parts of the peripheral hardware, by providing basic abstractions such as virtual processors, virtual-memory segments, and safely controllable device registers and hardware interrupts. All other operating-system services, however, run as user-level service modules, providing different services and abstractions of higher value and complexity: user-level drivers control hardware not directly managed by the kernel; user-level service modules are responsible for providing higher-order abstractions and resource-management functionality; examples are the POSIX layer in K42 [Krieger et al., 2006]), which provides an application programming interface akin to traditional UNIX systems; or the user-level virtualization layer in L4Linux [LeVasseur et al., 2004], which provides a software abstraction that looks like physical hardware. The resulting structure inherently implies a distribution and compartmentalization of operating-system resource management, where parts of the resource management reside within the kernel, and other parts reside within the different operating-system modules atop.

### Energy Management in Modular Operating Systems

Distributed resource management implies distributed energy management: there is no direct and centralized control over devices and their power states; likewise, there neither exists a single type of application exported to users, nor are all applications controlled by a single kernel instance. The privileged kernel only has access to those devices essential for secure isolation; user-level drivers only have access to their own dedicated set of devices; No component has comprehensive control over all devices and their power states. Likewise, application control is scattered across operating-system modules. The kernel and drivers have coarse-grained information on how low-level activities use the physical hardware. However, they do not possess any knowledge of the energy consumption of higher-level notions or types of applications, since this information is stored in higher-level resource-management subsystems. No component has comprehensive control of all individual applications and their effects on power and heat.

The scattering and isolation of device and application control bears substantial challenge for energy management in operating systems; we have identified three root factors causing the challenge: i) a semantic loss between modules; ii) an increased communication overhead; and iii) an increasing importance of inter-dependencies between different resource principals and devices.

**Semantic Loss between Modules** Monolithic operating systems are usually compiled from a single source-code repository; agreement about data structures and procedure signatures occurs through the common programming language. Modular operating systems, in contrast, do not have such shared build source; modules are compiled separately, possibly from source code in different programming languages, with different compilers and build environments, and from different vendors; each module is, in principle, a black box to the other modules. As a consequence, multiple energy-management subsystems may reside in different modules, and each subsystem cannot silently assume mutual agreement on the semantics and structure of energy-management–related data and control flow; rather, the bits of information relevant for energy management — accounting data, device power, performance, and heat states, information on applications, and so on — must be shared and structured explicitly. Legacy compatibility and virtualization play an important role, as they often limit the design space of protocols and interfaces for semantic sharing between modules.

**Increased Communication Overhead** In a modular operating system, the components taking part in the energy management reside in separate software modules; for reasons of confinement of privileges and failures, module domains are isolated from each other, by means of appropriate hardware or software protection mechanisms. The protection comes at the cost of increased overhead for crossing module boundaries; whereas monolithic kernels can rely on procedure calls and memory loads and stores to transfer control and data relevant for energy management, modular operating systems cannot use such direct mechanisms to retrieve the same information, but must resort instead to explicit communication primitives overcoming the protection boundaries. Crossing such boundaries, however, is expensive — to give an example, costs for direct procedure calls amount to a few dozen processor cycles at most on x86 processors; in contrast, simply switching the address-space base registers amounts to a few hundred cycles at least, without even considering indirect costs for cache misses. Direct sharing of memory is not always an option, as it must be carried out carefully in order to preserve the desired modularization and isolation [Gefflaut et al., 2000]. Even with well-designed and implemented communication and sharing primitives (e.g., [Aron et al., 2001, Haeberlen et al., 2000, Liedtke, 1993]), a residual communication overhead remains; consequentially, energy-management communication must be thoroughly designed and engineered in modular operating systems, rather than performed directly and ad-hoc as done in monolithic kernels.

**Importance of Interdependencies** Monolithic operating systems with their kernel-and-user-only separation scheme show a fairly simple dependency path in terms of resource and energy management: the kernel always executes on behalf of the application having invoked it; device accesses and subsequent energy effects can be directly tracked back to that application. Device sharing occurs, but only between applications and devices, and only between a single notion of applications. In modular operating systems, dependencies and resource paths become more complex. Multiple resource-allocation components and device-driver modules may exist concurrently, at different position in the hierarchy and with different scopes. Applications can have different types and granularities; coarse-grain applications (e.g., virtual machines) can consist of multiple sub-applications (e.g., applications within virtual machines). Sharing not only occurs between applications, but also between operating-system modules (e.g., a virtual file server at user-level, serving disk requests from an associated storage system). As a result, resource dependencies and paths become increasingly complex, and with them their effects on energy consumption and heat generation. Application-specific energy management cannot neglect the interdependencies and paths, however; rather it must cope with the different types and granularities of applications and with the sharing patterns of both hardware devices and software modules.

To summarize, a monolithic kernel has full control over all hardware devices and their modes of operation; it can directly regulate device activity or energy consumption to meet thermal or energy constraints. A monolithic kernel also controls the whole execution flow in the system. It can easily track the energy consumption at the level of individual applications and leverage its application-specific knowledge during device allocation to achieve dynamic and comprehensive energy management. Modern modular operating-system environments, in contrast, consist of a distributed and multi-layered software stack including a small kernel, multiple operating-systems components, device-driver modules, guest virtual machines, and other service infrastructure. In such an environment, direct and centralized energy management is unfeasible, as device control and accounting information are distributed across the whole system.

As a result, modular operating systems require well-designed, system-level support for energy management, in order to overcome the semantic loss between operating-system modules; to mitigate the effects of increased communication overheads across modules; and to deal with the increasing importance of interdependencies between different resource principals, system services, and hardware devices.

## 2.5 Related Approaches

So far, we have discussed existing approaches to energy management in operating systems only in a non-specific way. In this chapter, we will substantiate our general observations by discussing concrete approaches and relating them to our own work. We begin by reviewing existing approaches for traditional, monolithic operating systems. We subsequently review approaches for advanced operating systems, which already show some form of modularization. The section is divided into four subsections:

**Energy Management for Traditional Operating Systems** Most of the existing approaches to operating-system energy management target standard, monolithic operating systems, and do not address the problems that arise if the operating systems consists of several layers and is distributed across multiple components. We will review them in Section 2.5.1.

**Energy Management for Vertically Structured Systems** Vertically structured operating systems are modularized in the sense that they divide the operating system into multiple layers. There have been some approaches addressing energy-accountability issues within vertically structured systems. We will review similarities and differences in Section 2.5.2.

**Energy Management for Virtualized Systems** Several other efforts have investigated how energy management can be designed for virtualized operating systems, another, popular form of modular operating system. We will review similarities and differences in Section 2.5.3.

**Energy Management for Microkernel-Based Systems** Some previous work has been done investigating how energy management can be integrated into microkernel-based operating systems. We will review similarities and differences in Section 2.5.4.

### 2.5.1 Energy Management for Traditional Operating Systems

There exists a plethora of operating-system energy management approaches, most of which target standard operating systems. We review those approaches in the following, classifying them by four characterizing aspects: i) their application domains ii) their targeted hardware components; iii) their goals; and iv) their policies. Obviously, our classification scheme is not mutually exclusive, and approaches may fall in several categories: we therefore list prominent representatives for each aspect. We afterwards discuss the shortcomings of those approaches with regard to operating-system modularization.

### Application Domains

Operating-system energy management has two main application domains: mobile platforms and, more recently, the server space. Originally, operating-system energy management targeted mobile platforms, and typically focused on extending battery lifetime [Benini et al., 2000, Welch, 1995]. More recently, operating-system energy management has emerged to also target stationary and server systems, typically with the goal to reduce their operational costs and to overcome their limitations of higher integration densities [Bianchini and Rajamony, 2004, Lefurgy et al., 2003].

### Targeted Hardware

Pioneering approaches focused on reducing energy consumption of individual devices found in mobile systems, such as the processor [Weiser et al., 1994], memory [Huang et al., 2003], hard drives [Li et al., 1994], and network devices [Stemm and Katz, 1997]. Holistic schemes such as EcoSystem have investigated how the operating system can manage energy consumption of all laptop devices in a uniform way [Zeng et al., 2005, Zeng et al., 2002]. Servers, in turn, are distinct from mobile systems in that they comprise larger and more hardware: modern servers feature multiple processors with large caches, large amounts of possibly hierarchical memory, and high-speed network interfaces; they can have locally attached storage as well as external, network-attached storage. Multiple servers can be connected as clusters, blade servers, or in similar organization, forming computing systems of size of data centers. With such an architecture, energy management of individual devices or computers becomes too narrow. Research has responded by proposing mechanisms encompassing multiple devices [Bianchini and Rajamony, 2004, Felter et al., 2005, Isci et al., 2006, Merkel and Bellosa, 2006] and multiple computer nodes, as found in clusters [Elnozahy et al., 2003, Elnozahy et al., 2002] and data centers [Raghavendra et al., 2008, Ranganathan et al., 2006].

### Goals

With the growing complexity and emerging application domains of computer, energy management has also shifted its goals, from initially focusing on saving energy to reduce battery life to more complex objectives. As already explained (Section 2.1), besides lower power consumption, the most important objectives are lower operating temperatures, fewer power and temperature peaks, and better accounting and budgeting.

An important sub-goal in reaching those objectives is *modeling* power and thermal effects in the computer, and there exist numerous attempts to incorporate such models into the operating system, for individual devices such as processors [Bellosa, 2000, Snowdon et al., 2007], memory [Delaluz et al., 2001, Lebeck et al., 2000], or hard disks [Allalouf et al., 2009, Zedlewski et al., 2003], and for whole computers [Economou et al., 2006, Govidan et al., 2009] and computing centers [Chase et al., 2001, Heath et al., 2006]. Beyond modelling hardware energy effects, some studies have also investigated how applications can be modeled and analyzed for better energy management [Verma et al., 2009, Weissel and Bellosa, 2004].

Related to power modelling are the tasks of energy accounting and budgeting of individual applications. Accounting has been identified as essential prerequisite for operating systems that strive to provide energy awareness and quality-of-service guarantees at the same time [Neugebauer and McAuley, 2001]; and several attempts have been made to provide accounting and budgeting capabilities for mobile as well as server systems [Bellosa, 2000, Chase et al., 2001, Femal and Freeh, 2005, Flinn and Satyanarayanan, 1999, Ranganathan et al., 2006].

### Policies

Finally, a wide design space exists — and has been explored — in terms of the policies for operating-system energy management. Pioneering approaches for laptops and mobile computing were mostly concerned with turning individual devices into low-power modes during phases of idleness and underutilization [Lu and Micheli, 2001]. A vast number of approaches concentrated on processor energy management since the seminal work in [Weiser et al., 1994], which has led to a wide range of complex and flexible policies that perform fine-grain selection of processor power states — sleep states as well as as well as dynamic voltage and frequency settings —, while striving to mitigate the possible performance losses (e.g., [Flautner and Mudge, 2002, Govil et al., 1995, Snowdon et al., 2009, Weissel and Bellosa, 2002]). Similar investigations have been made for memory controllers, where intelligent data placement and scheduling can help to increase idle times of individual memory banks [Delaluz et al., 2001, Delaluz et al., 2002, Lebeck et al., 2000], and for peripheral devices such as hard disks, where, in addition to ordinary spin-down and dynamic rotation policies [Li et al., 1994, Lu et al., 2000, Gurumurthi et al., 2003], energy-aware caching and pre-fetching can help to increase disk idle times [Papathanasiou and Scott, 2004, Zhu et al., 2004] even further. In light of the interdependency between energy effects and application behavior, many other approaches have investigated, how applications and the operating system can interplay in a coordinated way to forecast periods of idleness and underutilization [Anand et al., 2004, Flautner and Mudge, 2002, Heath et al., 2002, Sachs et al., 2004, Weissel et al., 2002].

Again, with growing complexity and increasing importance of the server space, the policies of energy management changed. While the traditional opinion has been that idle periods in server workloads are too short for spin-down policies to become applicable [Carrera et al., 2003, Zhu et al., 2005], more recent studies on productive servers show that server workloads do show diurnal pattern variations that are sufficiently high to allow typical mobile disk power-management schemes to be applied [Narayanan et al., 2008]. Similarly, the traditional opinion has been server workloads are typically structured as multiple-application programs, thus server power management should place its focus on system-level rather than application-centric policies [Lefurgy et al., 2003]. More recent studies again show, that taking into account application behavior can indeed enable better energy management for servers and data centers [Kansal and Zhao, 2008, Mandagere et al., 2007, Raghavendra et al., 2008]. Finally, in presence of multiple devices and computers with different characteristics, many approaches have investigated how applications and request traffic can be allocated and balanced across devices and computers, thereby reducing power and power peaks [Colarelli and Grunwald, 2002, Felter et al., 2005, Kumar et al., 2003, Zhu et al., 2005] as well as heat and heat peaks [Gomaa et al., 2004, Merkel and Bellosa, 2006, Moore et al., 2005].

## Discussion

All the discussed approaches for traditional operating systems have in common, that they heavily focus on energy-management algorithms and policies, but take a centralized operating-system structure for granted. As such, they are not precluded from being applicable to modular operating systems *per se*; however, they do not address the specific problems that arise if the operating system consists of several layers and is distributed across multiple components, as modular operating systems do. We conjecture that much of the existing body of research can be applied to modular operating systems as well – provided properly designed mechanisms overcoming the problems of modularization are in place. It is our key goal to provide such a surrounding framework enabling different energy-management mechanisms in modular operating systems. We evaluate our framework in a realistic modular operating system and for different physical devices and different energy-management policies (see the following chapters). Naturally, given the wide range and technical complexity of existing energy management schemes for modern operating systems, it was impossible for us to substantiate that our approach is generic and flexible enough to fit every imaginable requirement. We still see our work as a first important step towards the development of energy-management schemes for modular operating systems.

### 2.5.2 Energy Management for Vertically Structured Systems

Vertically structured operating systems are one representative of modular operating systems, albeit the modularization occurs in a single "direction" only. The idea of vertically structuring an operating system originally stems from research in the area of microkernels [Engler et al., 1995] and multi-media operating systems [Leslie et al., 1996]. The fundamental approach is to layer the system into a small kernel at the bottom, applications at the top, and protocol stacks, user-level drivers, and other system libraries in between, with the advantage that such a structure allows multiplexing all resources at a low level, and moving kernel services into user-level libraries. Unlike microkernel-based systems, vertically structured systems abandon shared operating-system components or servers, and require the functionality typically performed by the kernel to be executed within each application itself. As a result, most resource and energy consumption can be accounted to individual applications, and there is no significant anonymous consumption anymore. Such a structure can enable accurate and easy accounting and pricing of energy, as demonstrated in a research effort for the Nemesis operating system [Neugebauer and McAuley, 2001]. A big limitation of vertical structuring, however, is that it trades off freedom in the design space of modularization against better facilitation of accounting. In contrast, it was one of the key goals of our work to explicitly account the energy spent in more complex shared service and driver components, since such components are common in other forms of modular operating systems.

### 2.5.3 Energy Management for Virtualized Systems

In terms of resource management, virtualization environments resemble the structure of vertical operating systems: a hypervisor or host operating system also multiplexes system resources at a low level, and lets each virtual machine use its own protocol stack and services. Scheduling occurs in layered, hierarchical fashion, with the hypervisor, guest kernels, and applications being stacked on top of each other. In addition, host-level resource management occurs independent from guest operating systems, permitting virtual components to recursively implement their own strategy.

However, with the ongoing trend in virtualized environments to restrict the hypervisor support to a minimal set of hardware and to perform most of the device control in unprivileged driver domains [Fraser et al., 2004,LeVasseur et al., 2004], virtualized environments have abandoned the pure principles of vertically structuring, in favor of a multi-server like structure, where shared service components run atop the actual hypervisor. Hosted virtualization solutions such as VirtualBox [Sun Microsystems Corporation, 2009] or VMware workstation [VMware

Inc., 2009c], a popular solution for end-user and workstation systems, show a similarly loose vertical structuring: a small host kernel module is responsible for world switches, but all other virtualization functionality is contained in a host application relying on the capabilities of the fully-fledged host kernel. Either way, virtual machine monitors layer the operating system stack, and, incidentally, modularize it. As such, they can be regarded as a modular operating system, albeit with strict interfaces based on hardware semantics.

In light of the ongoing trend to virtualization, research has already investigated whether and how energy-management issues change. Many of the existing efforts have focused on the ability of virtualization to encapsulate a whole application stack and how it can be leveraged for server energy management: that is, virtual machines can be moved around easily and are an ideal container for dynamic, energy-aware workload placement [VMware Inc., 2007, Nathuji and Schwan, 2008, Nathuji et al., 2008, Raghavendra et al., 2008, Tolia et al., 2009, Verma et al., 2008, Verma et al., 2009].

Fewer studies have been performed on how the modularization of operating system caused by virtualization changes the mechanics of operating-system energy management. Recently, Wang and colleagues proposed a cluster-level architecture that coordinates power and performance management of virtual machines within and across physical computers [Wang and Wang, 2009]. The most prominent work in this area, however, has been done by Nathuji and colleagues. In his PhD thesis and publications, Nathuji investigated mechanisms for coordinated power management between diverse management components that exist across system layers, again within and across physical computers [Nathuji, 2008]. Particularly with regard to virtual-machine environments, his work explores how processor–power-management policies in the virtualization layer can coordinate with guest application policies via virtualized processor power states [Nathuji and Schwan, 2007]; how token-based processor scheduling schemes help power budgeting across virtualized applications and the computer hardware [Nathuji and Schwan, 2008]; and how platform power budgets in virtualized systems can incorporate quality-of-service hints from guest operating systems [Nathuji et al., 2009].

Since our work explicitly encompasses virtualization, much of the existing findings are applicable to our goal of providing energy-management capabilities for modular operating system as well. Our work is different in several aspects: First, while Wang, Nathuji, and colleagues are focused on virtualization solutions only, our work also investigates and evaluates aspects of modular operating systems not directly related to virtualization. For example, their energy-management mechanisms and policies proposed are solely based on virtualized devices such as virtual processors; our work also investigates the case where a microkernel provides more fine-grained abstractions such as kernel threads and light-weight communication. Second, the works by Wang, Nathuji, and colleagues were heavily centered around processor power management; in contrast, our work also explores management of peripheral devices such as hard disks. Third, many of the mechanisms proposed in their work were explicitly designed to leverage existing hypervisor mechanisms for energy management: Wang and colleagues used the credit scheduler in the Xen hypervisor [Xen.org, 2009] for carrying out their processor power management decisions. Nathuji and colleagues also used existing mechanisms where possible: mapping virtual power states to physical power settings and scheduling parameters in [Nathuji and Schwan, 2007] was mostly done using existing Xen mechanisms; the quality-of-service-aware power budgeting mechanism in [Nathuji et al., 2009] was based on the capabilities of Hyper-V to cap processor utilization of individual virtual processors. Nathuji and colleagues deliberately made the decision to use existing mechanisms, in order to permit their findings to be applied the wide range of virtualization solutions currently deployed or under development [Nathuji and Schwan, 2007]. In contrast to all those approaches, it was our explicit goal to explore how resource and energy-management primitives can be designed to work across module boundaries, without restricting ourselves to existing ones.

### 2.5.4 Energy Management for Microkernel-Based Systems

Finally, some previous approaches have investigated how energy management can be integrated into microkernel-based modular operating systems. In the following, we will discuss them and relate them to our own work.

Two prominent and long standing microkernel implementations for embedded systems are Symbian OS and QNX. The Symbian Operating System, a microkernel-based system for mobile phones, has a feature-rich power-management subsystem designed to enable controlling the hardware power-management capabilities, extending the battery life time, and managing the user's perception of the phone operational state. Its user interface comprises of both system calls and up-calls: system calls enable applications to initiate device standby, to manage wakeup events, or to register notifications; up-calls, in turn, allow the kernel to notify applications on power-related events such as system standby. The core power-management functionality is implemented within the kernel, both for processor and peripheral devices; device-specific kernel extensions, called power handlers, encapsulate low-level device management code and communicate with the rest of the kernel through well-defined interfaces. Interfaces and extensions are implemented using C++ language constructs [Sales, 2005].

Similarly, the power-management subsystem of QNX, a real-time microkernel operating system for embedded devices and mission-critical applications, defines a set of mechanisms that enables developers to control and manage power in their devices. Akin to the goals of our work, and in contrast to Symbian OS, the QNX framework avoids imposing a power policy on applications. Instead, it lets developers create the policies based on their own application-specific needs. To that end, the framework supports a user-space power-manager application called power server, which coordinates the power management across client applications, the kernel, and device drivers (which are not part of the kernel in QNX). The framework also provides libraries to establish well-defined power-management interfaces between the involved components [QNX Corporation, 2009, Ethier, 2004].

Both the Symbian and QNX power-management subsystems share many of the goals and ideas underlying our own work; still, there are some differences: both frameworks heavily focus on low-power states of the hardware such as device standby and sleep states; to our best knowledge, they do not pay attention to the problem of energy accounting and budgeting in modular operating systems. Also, neither of the two approaches was explicitly designed with virtualization in mind — despite the growing importance of virtualization solutions even in the embedded area [Heiser, 2008, Heiser, 2009].

Finally, the study by Lawitzky and colleagues [Lawitzky et al., 2008] focuses on integrating a power-management subsystem into an embedded real-time operating system based on the OKL4 microkernel [Open Kernel Labs, 2009]; their approach combines dynamic frequency and voltage setting with deadline-based real-time scheduling. Our work differs from theirs in two aspects: first, their approach focuses more on the algorithms and policies of processor power management, but leaves the actual implementation in the kernel. In contrast, our work investigates how energy management and can be exposed from the kernel and device drivers in order to provide more flexibility in terms of energy-management policies. Second, their approach focuses on processor power management, while our work also explores power management of peripheral devices.

### 2.5.5 Summary

To summarize, approaches relevant and related to our own work fall into four categories: approaches for traditional operating systems, approaches for vertically structured system; approaches for virtualized systems, and approaches for microkernel-based systems.

Approaches to power management in traditional operating systems do not address the specific problems of operating-system modularization. We conjecture many of those approaches, particularly their policies and algorithms, are valid and applicable in modular operating systems in principle, provided a system-level support for modular power management is in place; it is our key goal to provide such a surrounding and enabling framework.

Approaches for vertically structured systems, at least in their pure form, render it hard to achieve resource and energy accountability for shared drivers and operating-system services. In contrast, it was one of the key goals of our work to explicitly track and account the energy spent in service or driver components.

Approaches for virtualized systems share many goals, insights, and ideas of our own work; however, those approaches are typically focused on virtualization systems only, and do not address the problems and specifics of other instances of modular operating systems, such as microkernel-based systems. Also, many of the proposed schemes mechanisms were explicitly designed to leverage existing system-level resource-management primitives where possible; in contrast, it was our explicit goal to explore how resource-management primitives can be changed or transformed to allow exposed energy management across module boundaries.

Finally, approaches for microkernel-based systems are typically geared towards embedded computers such as mobile phones; as such, they heavily focus on extended battery lifetime, by exploiting low-power states of the hardware; in contrast, our work also concentrates on the problem of distributed energy accounting and budgeting in modular operating systems, and on the problems arising if a virtualization layer is integrated into the operating-system stack.

# Chapter 3

# Energy-aware Modular Operating Systems

In this section, we present the design of a novel framework for managing energy in distributed, multi-layered operating-system environments, as they are becoming common in today's computer systems. Our framework strives to enable energy awareness and energy management if the resource-management subsystem is distributed and scattered among operating system modules rather than being centralized and monolithic. The presented approach targets multi-component and multi-layered operating systems; concept-wise, we do not limit ourselves to specific scenarios such as the server or mobile domain; we believe our findings to be valid and applicable to any modular operating system, whether it may run on a mobile device or on a server. The chapter is organized as follows: Section 3.1 presents an overview, which defines the problem we intend to solve and the design goals we strive to obey, and introduces the key concepts of our approach. Section 3.2 presents our general model for energy management in modular operating systems. Section 3.3 presents our exposed and modularized approach to energy accounting. Section 3.4 presents our exposed and modularized approach to energy allocation. Section 3.5 explains the design of two interaction protocols that coordinate how energy-management data and decisions are propagated between different modules. Finally, Section 3.6 details how we deal with virtualized resources and devices, respectively the energy aspects thereof.

## 3.1 Overview

Our approach strives to enable energy awareness and energy management if resource-management subsystems are distributed and scattered among operating-system modules rather than being centralized and monolithic; to that end, it provides the mechanisms needed for communicating energy-management–related information between modules; for overcoming the semantic loss of energy-management information and structure induced by module isolation; and for dealing with the increasing complexity and interdependency of multiple notions and hierarchies of energy-consuming activities on the one hand, and the varying types, sharing levels, and paths to hardware devices on the other hand.

### 3.1.1 Goals

We begin by stating a set of primary design goals our approach should reflect:

**Flexibility in energy-management algorithms and objectives** Our approach must be flexible enough to support diversity in energy-management schemes. Specifically, the approach should be flexible in terms of management *objectives*, and in management *algorithms*. Primary objectives of computer energy management are reducing power consumption and heat, avoiding power and temperature peaks, and enabling energy budgeting and accounting. There exists a variety of algorithms to achieve those energy-management objectives; a valid solution must be flexible and extensible enough to suit the diversity of both objectives and algorithms.

**Efficient support for modularity and isolation** Modern operating systems consist of multiple components isolated from each other. Any approach to managing energy in a modular operating system must provide interfaces that allow overcoming the semantic loss between modules; it must perform efficiently across isolation boundaries and protection domains; and it must comprise all the different types of resources, activities, and their interdependencies. However, the approach must also retain the desired isolation properties of modular operating systems, and enable preserving isolation guarantees for energy and thermal characteristics across activities and the computer users running them.

**Preserve compatibility and enable customizability** Finally, modern operating-system environments often employ some form of virtualization to preserve compatibility to legacy code; at the same time, special interfaces and abstractions such as a native programming interfaces and abstractions allow the system to remain open to customization. Hence, distributed energy management must cater for both appropriate virtualization and compatibility of energy effects, and for managing energy in presence of custom operating-system abstractions and mechanisms.

### 3.1.2 Approach

We present a novel framework for managing energy in distributed, multi-layered operating systems. The framework provides the following key contributions:

**A Model for Modularization-Aware Energy Management** As first contribution, we model the operating system as a set of modules comprising resources, activities, and energy-policy managers; the energy management then becomes a feedback loop involving one or more of such operating-system modules. We furthermore propose to solely rely on the notion of energy as the base abstraction. Energy quantifies the physical effects of power consumption in a distributable way and can be partitioned and translated from a global, system-wide notion into a local, component or user-specific one.

**Exposed and Distributed Energy Accounting** As second contribution, we propose a distributed energy-accounting approach, which accurately tracks back the energy spent in the system to originating activities. In particular, the presented approach incorporates both the direct and the side-effect energy consumption spent along the path from an application down to the hardware; such a path may involve transitioning through system-service modules, virtualization layers, or subsequent driver components, and thus result in additional energy consumption on other devices.

**Exposed and Distributed Energy Allocation** As third contribution, we propose to expose suitable resource and energy allocation mechanisms from drivers and other resource managers to the respective energy-management subsystems. Exposed allocation enables dynamic and remote regulation of energy consumption in a policy-neutral way, allowing policies and algorithms to be contained in separate modules, where they can be tuned or exchanged individually.

**Energy-Management Interaction Protocol** As the fourth contribution, we explore two suitable interaction protocols for communicating energy-management information in a modular operating system. Exposing mechanisms for energy accounting and allocation requires propagating energy-accounting state and resource-allocation decisions between involved modules. Different communication protocols are thinkable; we postulate that the optimal protocol is largely defined by two opposing factors, the timeliness requirements of the energy policies, and the performance overhead induced by module isolation. We explore two different protocols, a synchronous and an asynchronous variant.

**Virtualization of Energy Effects** As fifth and final contribution, our framework supports accounting and allocation of energy not only for physical devices but also for virtual devices. Platform virtualization is a convenient mechanism for enabling legacy support while advancing the actual operating-system structure underneath; supporting a notion of virtual energy provides the additional benefit of a development path towards fine-grain energy-aware resource management for virtualized applications. Energy-awareness in a guest operating system is not only beneficial for hosting legacy application workload; with the ongoing trend to perform the device control in unprivileged driver domains [Fraser et al., 2004, LeVasseur et al., 2004], it also allows drivers and service modules running in virtual machines to leverage the energy-virtualization capabilities for managing their own services.

### 3.1.3 Scope of the Approach

The presented approach targets multi-component, multi-layered, and multi-server operating systems, including microkernel-based and hypervisor-based operating systems, and with or without platform virtualization layers. We evaluate our approach on a single-node, x86-based, stationary computer system (See Section 5). Conceptually, however, we are not limiting ourselves to specific scenarios such as the server or mobile domain, and believe our findings to be valid and applicable to any modular operating system, whether it may run on an embedded device or on a large server system. Conversely, we are not focusing on energy-management problems that are particular to specific hardware scenarios such as preserving battery life on mobile systems or balancing temperatures across multi-processor or multi-node systems. Since few research approaches have focused on the area of energy management for modular operating systems so far, our approach focuses on system-level support for energy management, that is, on mechanisms and infrastructure rather than on policy and strategies. We hope that our findings may serve as a starting point, or piece in the foundation of modular operating-system energy management.

## 3.2 A Model for Modular Energy Management

Current approaches to operating-system energy management are tailored to single building-block operating-system designs, where one kernel instance manages all software and hardware resources. Our first step towards energy management for modularized operating systems is to break up this centralized system-design paradigm, and to instead model the operating system as a set of modules and subsystems. Each of the operating system modules has a specific task; some are responsible for controlling a hardware device; others export a service, a set of library functions, or other software resources for use by activities. As the operating-system tasks are distributed, so are the energy-management subsystems: Control and monitoring of hardware power and temperature states reside within the driver modules controlling the actual devices, while control and monitoring of applications are managed by the modules exporting and managing the notions of activity in the system.

### Energy-Management Components

We propose a unified energy-management model, which strives to reflect the different notions of activities and energy consuming resources, as well as their isolation and interdependencies. The model is illustrated by Figure 3.1; it consists of *resources and resource drivers, client activities, and policy managers*:

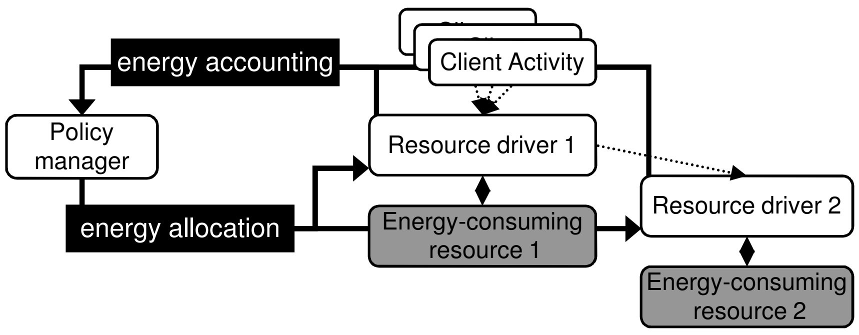

Figure 3.1: A model for modular energy management. Resource drivers are responsible for managing hardware or software resources respectively; the energy consumption of resources is accounted back to the original clients. Energy management is modeled as feedback loop performing resource allocation based on incoming accounting and monitoring data; policy managers represent the policies and strategies used to implement the feedback loop; all modules may (but need not) reside in different components or protection domains.

**Resources** represent energy-consuming devices or software abstractions. Examples are hardware devices such as processors, main memory, or disk devices, and software abstractions such as communication channels, or virtual devices. Resources are controlled and catered for by a resource provider, or resource driver; the primary task of the driver is control the hardware or software resource and to multiplex client activity among it. A main characteristic of resources and providers is that they can be stacked: that is, low-level resource providers export rudimentary hardware abstractions (e.g., a virtual network or disk device); higher-level resource providers refine and export them as more elaborate abstractions (e.g., as socket, or file).

**Client Activities** represent consumers of resources and thus clients of resource providers. We treat energy consumption and heat dissipation as the result of such resource consumption. Examples of client activities are processes or threads, as defined by UNIX or Windows, or more complex containers such as a whole process hierarchy belonging to an application or even a complete guest operating system. As with resources and providers, client activities can be stacked upon each other, and an activity may consume low-level resources to provide higher-level resources. As an obvious result, client activities can act as resource provider themselves as well.

**Policy managers** represent the subsystems responsible for deciding in which performance or power state the hardware should be put into, and how allocation of resources to activities should take place. In centralized operating systems, the policy-management subsystem is entangled with the resource-provider subsystems in the kernel. In contrast, modular operating systems may feature multiple policy-manager components, at different position in the hierarchy and with different scopes, each responsible for a set of subordinate resources and their energy consumption.

### 3.2.1 Unified Notion of Energy

Objectives, policies, and mechanisms of operating-system energy management are diverse; the underlying management infrastructure must be unified and general enough to encompass that diversity. Energy accounting, budgeting, and other application-centric aspects of energy management furthermore require that the management infrastructure allows the effects of energy consumption to be attributed to different applications or users.

We therefore propose to use the notion of *energy* as the base abstraction in our system. Our idea of using energy as base unit is inspired by the *currentcy* budgeting model in ECOSystem, although ECOSystem is tailored towards mono-

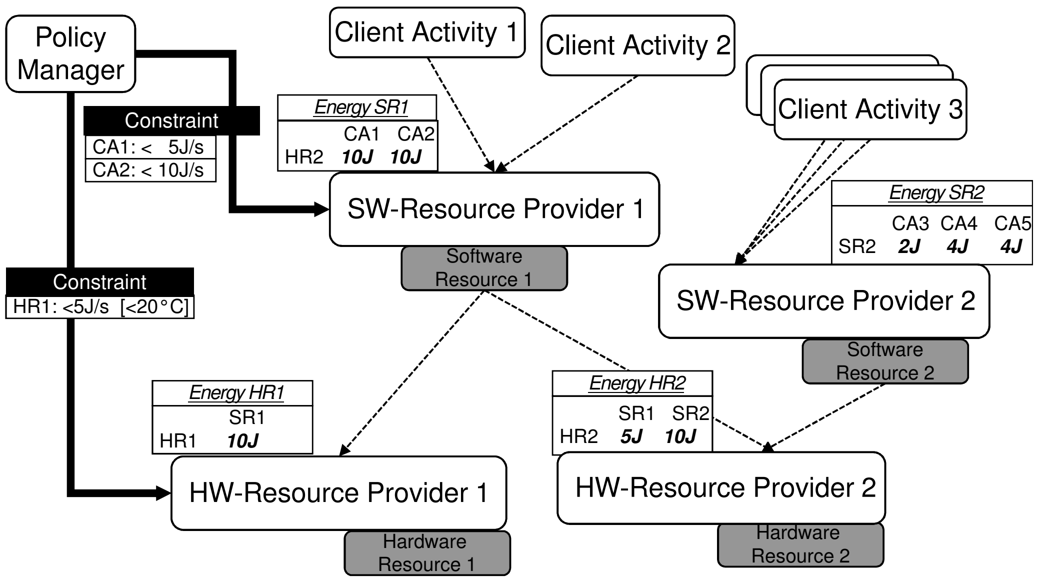

Figure 3.2: Using energy as basic notion for distributed energy management. Energy acts as a quantifiable and partitionable base unit, for different types of software and hardware resources and activities. Resource providers perform accounting and balances on the base of energy. Policy managers express energy-management goals only via energy constraints. Temperature effects and goals must be translated using thermal device models.

lithic operating systems, as detailed in Section 2.5 [Zeng et al., 2003]. Two reasons speak in favor of such an approach: First, energy allows the infrastructure to quantify and manage power consumption in a uniform way, and serves as a coherent base metric to unify and integrate management schemes for different hardware devices. For example, using energy as the base unit enables accumulating the (energy) consumption of a client activity among multiple devices to a single, but *meaningful* value. Second, energy quantifies the energy effects in a way that they are partitionable among multiple clients and users. Again, to give an example, resource providers can use energy as a base unit to divide their own energy allotments among the client activities they serve (see Figure 3.2).

However, using energy as sole unit also requires that energy-management policies and goals are expressed in form of energy constraints. In the following subsection, we will explain how this can be done for the other commonly observed and explored energy effect in computers: the temperature.

### 3.2.2 Translating Temperature and Thermal Models

With its unifying and partitioning properties, energy is in contrast to thermal conditions of the computer; device temperatures are neither unifying nor partitionable. Operating temperatures of different hardware devices such as processors, main memory, and disks, are highly dependent on device-internal physical characteristics such as thermal conductivity and absorption of the materials used. As such, they are not comparable, let alone accumulatable, for different types of devices. Moreover, while temperature variations go along with variations in energy consumption, their time scales are very different: energy consumption reacts and changes quickly depending on the activity and workload on a device, while temperature changes take significantly more time [Merkel and Bellosa, 2006]. As a consequence, attributing the temperature of a device to individual activities using it is much harder, if not impossible, than to attribute its energy consumption.

Fortunately we can avoid using temperature as a base unit while still encompassing thermal effects in our system. Thermal effects can be translated and expressed as energy constraints, by means of a thermal model. Research has proposed several approaches to modeling thermal behaviors of computer systems and devices: Skadron and colleagues model thermal behavior in architecture-level power and performance simulators [Skadron et al., 2003b]. Bellosa and colleagues derive, based on a thermal model, the temperature of a processor from its energy consumption at runtime [Bellosa et al., 2003]. Merkel and colleagues use a similar model to distinguish hot and cold processes [Merkel and Bellosa, 2006]. Heath and colleagues have built a complete software suite that emulates temperatures based on simple layout, hardware, and component utilization data [Heath et al., 2006].

Our framework requires that thermal conditions and constraints are expressed as energy conditions and constraints; doing this allows us to attribute thermal behavior such as device temperatures to individual activities, and also allows us to translate global thermal constraints into energy budgets specific to individual activities and to individual devices.

### 3.2.3 Energy-Management Feedback Loop

As described in Section 2.2, the main tasks of operating-system energy management are to monitor and control energy-related aspects of both hardware devices and the applications running atop. We formulate the procedure of energy management as a feedback loop consisting of three steps:

**Step 1** Monitoring energy consumption and accounting it to client activities.

**Step 2** Analyzing the accounting data, and making an allocation decision based on an energy policy.

**Step 3** Implementing the decision by modifying hardware power states or by allocating or reallocating energy-consuming resources to activities.

In the first step, the system must determine the current energy state of each resource and account it to the originating activities using the resource. This step must be performed by the resource drivers responsible for managing the particular hardware or software resources. In the second step, the system must propagate the energy-accounting information to the associated policy managers; their task is to analyze the accounting data and use it as input for the particular policy. If needed, the policy managers make a decision to change hardware power or performance states, or to (re-)allocate resources. In the final step, the policy managers carry out their decisions, by propagating them to the respective resource drivers, which eventually put them through he on the resources.

### Modular Policy Management

The basic rationale behind our model is the rule of separation of policy from mechanism, a long-standing [Levin et al., 1975] and common [Raymond, 2004] guideline for operating-system design. Our basic assumption is that energy-management policies change more rapidly than the underlying mechanisms they require, since they depend on the objectives and on the deployment environment of the operating system. Separating policy from mechanism thus allows to preserve flexibility and customizability of energy management. We therefore model the policy as separate energy-policy manager, whereas the two other steps, energy accounting and allocation, are mechanisms, bound to the respective resource providers.

Since the system is distributed, energy-policy managers cannot assume direct control or access over resources; instead, they require remote mechanisms to account and allocate the energy. Hence, by separating policy from mechanism, we translate our general goal of distributed energy management into the two specific aspects of **exposed and distributed energy accounting** and **exposed and distributed energy allocation**; these are the subject of the following sections.

## 3.3 Exposed and Distributed Energy Accounting

Energy accounting refers to the systematic monitoring of data relevant for the energy-management process. It has been identified as a crucial task for operating-system energy management for quite some time [Neugebauer and McAuley, 2001]. Our model uses energy as basic unit, thus we narrow the accounting process down to the problem of providing accurate data on the *energy* consumption *per client activity and resource*.

Ultimately, the purpose of energy accounting is to answer questions like “How much power is a given resource consuming at the moment?” or “How much of the power consumption on a given resource can be attributed to a given activity?”. Indirectly, energy accounting should also answer questions like ”How much heat is generated on a given device?” However, we leave such subsequent questions up to the policy managers, which we thus treat a black box in terms of accounting. The energy-accounting process is illustrated by Figure 3.3; it consists of three substeps : i) determining or estimating device-energy consumption; ii) attributing it to individual activities; and iii) exposing the accounting records to policy managers.

### 3.3.1 Determining Device-Energy Consumption

Many modern computers and devices have limited or no hardware support for determining energy consumption at runtime. While they often provide feature-rich facilities to measure performance-related statistics (such as performance counters in processors [Levine and Roth, 1997, Intel, 2010]), they typically lack comparable capabilities for power or energy consumption. Computers and devices do often feature some form of thermal sensors [Naveh et al., 2006]; however, raw thermal information is of little use for accounting purposes, as it cannot be partitioned among client activities.

As a result, energy-estimation schemes have become a crucial task for energy management. Research has proposed several estimation models for computer devices, including models for processors [Bellosa et al., 2003, Joseph and Martonosi, 2001], and for peripheral devices such as disks [Allalouf et al., 2009] or network interface cards [Zeng et al., 2003]. Those models typically leverage hardware effects that are not directly related to energy but measurable in software. Examples of such effects are processor performance states, measurable via hardware counters, and device usage, measurable by monitoring client activity in software. From these effects, the estimation models derive the energy consumption, which often includes calibration of some sort. As device types — sometimes even devices of the same type —, often vary in terms of measurable effects and their correlation with power consumption, individual estimation schemes are necessary for individual types of devices.

From these observations we draw two important conclusions for our management model: first, determining or estimating the energy of a physical device is a low-level, hardware-specific task that requires detailed knowledge of the particular device. Second, the actual estimation model and subsequent measurement tasks may vary among device types, sometimes even between individual devices of the same type. Our framework therefore requires each driver of an energy-consuming device to account its own energy consumption to client activities; we leave it open to the implementation, how the driver achieves this requirement. We will present two implementations for processor and disk devices when presenting our prototype in Chapter 4.

## 3.3.2 Attributing Device-Energy Consumption

Once current device-energy conditions are known, the next step consists of attributing them to client activities. Device drivers usually deal with low-level notions of client activities such as threads (for processors), address spaces (for main memory), or device requests (for disks, network interfaces, or other peripheral devices). Ultimately, however, goal is to attribute energy to logical entities such as applications, users, user groups, or other principals and domains, which are vessels, or containers, encompassing the low-level notions [Banga et al., 1999].

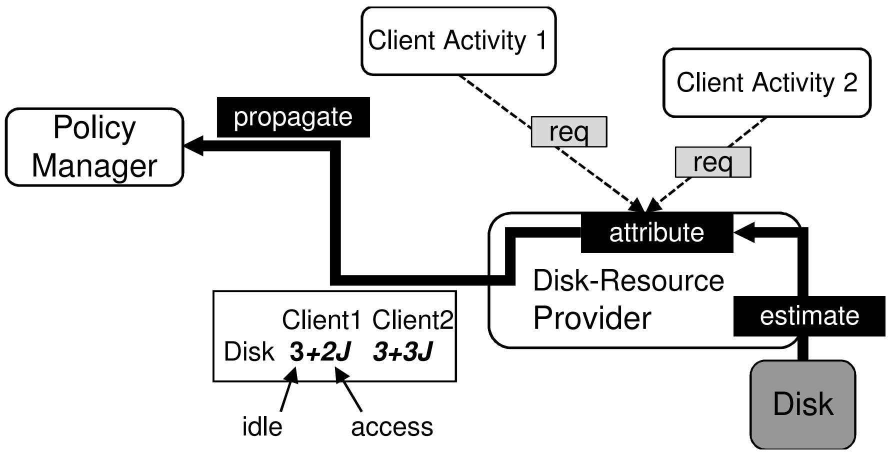

Figure 3.3: Exposed energy-accounting process. Device-energy consumption is estimated by means of an estimation model. The energy consumption is attributed to activities, for example when requests are issued or finished. Requests are mapped to activities by means of a resource domain identifier, and separated into an idle and an access energy portion. Energy-accounting records are finally exposed and propagated to energy-policy managers.

To match diversity in logical principals, we introduce the construct of a *scheduling domain*. For resource drivers performing the energy accounting, the domain is merely an attribute of their low-level client activities, emitted to policy managers for identification. A management interface enables the scheduler to choose different (or common) identifiers for different client activities. A policy manager can denote the same identifier for several address-spaces belonging to a single application, or for several virtual processors belonging to a single virtual machine. The result of such grouping is that resource drivers will charge energy consumption to the same scheduling domain. This approach can be regarded as a basic adoption of the resource-container concept in [Banga et al., 1999], although with much fewer semantics.

When attributing energy consumption to client activities, we break down the energy consumption into *access* and *idle consumption*. Access consumption consists of the energy spent when using the device. This portion of the energy consumption can be reduced by controlling client activity (e.g, the number of client requests issued over time). Idle consumption, in turn, is the minimum energy consumption of the device, which it needs even when it does not serve requests. Energy management cannot control this portion; however, it still needs to be accounted and attributed properly to the originating activities.

Many modern hardware devices support multiple active power states: modern processors, for instance, support different frequency and voltage settings [Naveh et al., 2006]. Similarly, although not yet available by default, multi-speed disks allow lowering the spinning speed during phases of low disk utilization [Gurumurthi et al., 2003]. Also similarly, multi-technology disk systems integrating magneto-electronic and solid-state hardware, enable multiple device states with respect to performance and energy consumption [Narayanan et al., 2009]. Obviously, using multi-activity devices have implications on both the accounting and the allocation step of our energy-management model. To retain fairness, we propose to decouple the power state of a multi-speed device from the accounting of its idle costs. Clients that do not use the device are charged for the lowest active power state. Higher idle consumptions are only charged to the clients are actively using the device.

### 3.3.3 Exposing Device-Energy Consumption

Having determined and attributed energy consumption, the resource driver propagates the accounting records to the corresponding policy managers. The policy managers leverage the accounting information to retrieve a global view on the energy and thermal conditions for subordinate devices and activities.

An important question with regard to the exposure of energy consumption is *when* a resource driver should communicate those internal records to external policy managers. The internal events when accounting data accrues are ultimately dictated by the way the driver-internal resource allocation takes place, which is implementation-specific. Examples of such events are the generation or termination of resource requests, for instance, when processor threads start up or terminate, or when network interface requests are issued or completed. However, the propagation of accounting data does not necessarily have to occur at those internal events; rather, the propagation should take place whenever the energy-policy managers require the data. We therefore propose different management protocols, which are unified for both energy accounting and allocation. We will discuss them later, in Section 3.5.

## 3.3.4 Energy Accounting of Software Resources

Our energy-management model allows hierarchical arrangements of resources and activities: resource providers may become activities on other resources themselves; however, as a result, activities of interest for accounting may operate on software resources rather than directly on raw hardware.

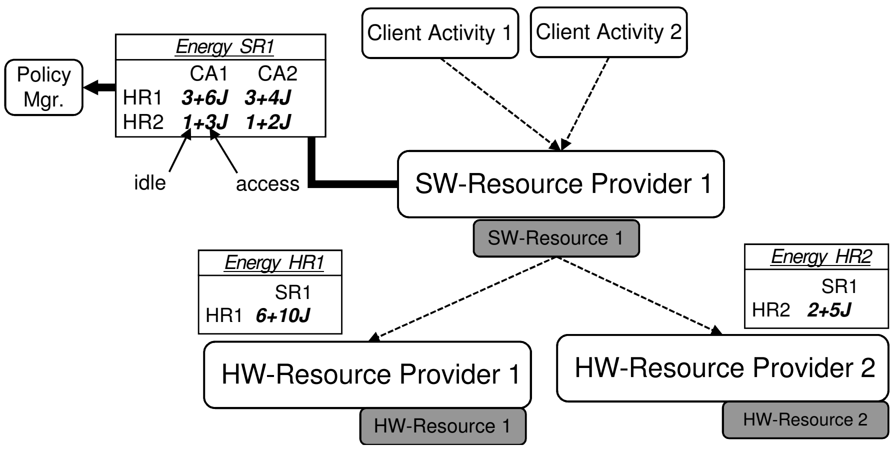

Figure 3.4: Recursive accounting of software energy consumption; for each client, the resource driver reports idle and active energy to the energy manager. The driver is assumed to consume resources itself, whose energy consumption is apportioned recursively to the two clients.

To account the energy consumption of a software resource, our approach must incorporate the energy spent by a given activity *in all services*, and *by all devices* along the path from the software resource down to hardware devices. For that purpose, all resource drivers perform energy accounting in our infrastructure, those responsible for managing a raw hardware device, and those responsible for software resources. Since provisioning software resources may involve interacting with several different other resources, we perform *recursive* accounting of the energy spent in the system: at the lowest level, each driver of a raw physical device determines the energy spent for fulfilling a given request and passes the cost information back to its client. If the driver requires other devices to fulfill a request, it charges the additional energy to its clients as well. As a result, recursive accounting yields a distributed matrix of software-to-hardware transactions, consisting of the idle and the active energy consumption of each physical device required to provide a given software resource (Figure 3.4). Each device driver is responsible for reporting its own vector of the physical device energy it consumes to provide its software abstraction. Since idle consumption of a device cannot be attributed directly to requests, each driver additionally provides an "electricity meter" per clients. The meter indicates the client share in the total energy consumption of the device, including the cost already charged with the requests. A client can query the meter each time it determines the energy consumption of its respective clients. Note that, with infrequently used devices, such a scheme may induce a fairness problem in that the first client using a device after a long period of idleness may get accounted all the idle energy waster previously. At present, however, we leave this problem open and use the same idle accounting scheme independent of the utilization patterns of individual devices.

Finally, we also note that in monolithic operating systems, resource consumption paths may involve recursive consumption of other resources as well: think of a UNIX network socket as an example; provisioning the socket not only requires energy for networking interface card but also processing energy for queuing requests, checksumming, et cetera. However, monolithic operating systems are typically vertically structured, with the kernel at the bottom always executing on behalf of the application atop having invoked it. In modular operating systems such simple layering is impossible. A software resource may be serviced by a separate resource provider layered alongside the clients in a separate protection domain rather than below, in the kernel.

## 3.4 Exposed and Distributed Energy Allocation

To enable flexible and extensible definition of energy policies, our framework requires each driver to expose suitable resource-allocation mechanisms to external policy managers. Policy managers leverage the allocation mechanisms to ensure that energy consumption matches the desired constraints.

Resource-allocation mechanisms relevant for energy management can be distinguished into hardware and software mechanisms. Hardware mechanisms typically provide a means to change power consumption of a device, offering several modes of operation with different energy and performance coefficients. The main reason to offer hardware mechanisms is to enable the operating system to dynamically adapt the hardware performance to the actual load situation, avoiding energy waste. Software-based mechanisms, in turn, rely on the assumption that energy consumption depends on the level of utilization, which is ultimately dictated by the number of device requests. Adapting the request rate in software then helps to control the energy consumption of hardware.

Again, our model uses energy as basic unit, and we narrow the task of ex-

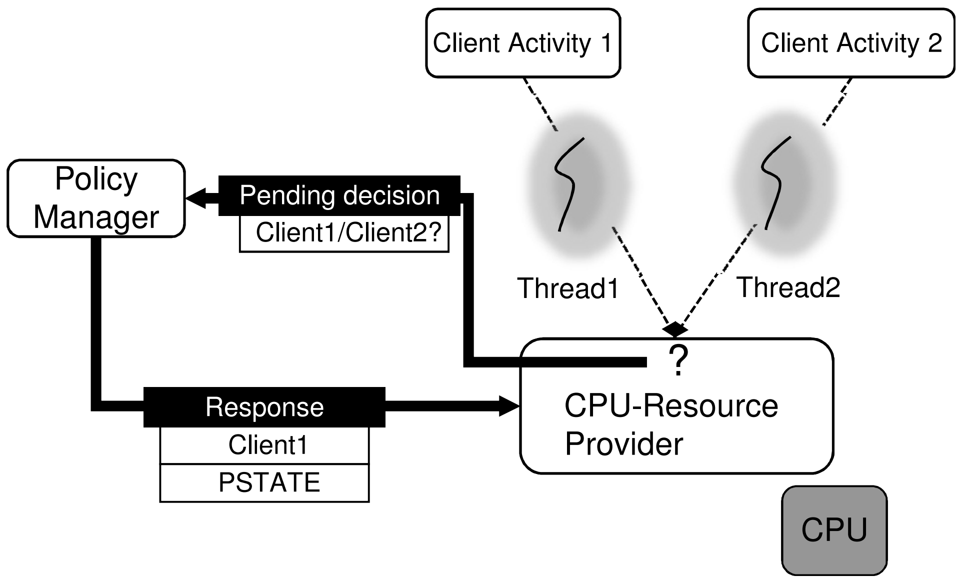

Figure 3.5: Exposed energy-allocation process, illustrated using the example of processor scheduling. Events where processor-allocation decisions are pending are propagated from the processor-resource provider to associated policy managers. Allocation decisions are performed by the policy managers based on their energy policies. The decisions are re-injected into the resource provider and then carried out by its internal dispatching logic. Decisions contain updated scheduling state as well as updated processor power or other hardware state.

posed energy allocation down to the problem of providing accurate means for controlling *energy* consumption *per activity and resource*. We leave subsequent questions (e.g., thermal control or balancing) up to the policy managers. The energy-allocation process is illustrated by Figure 3.5; it consists of the following sub-problems: i) exposing pending resource-allocation decisions from resource providers to policy managers ii) re-injecting the responses back from policy managers to resource providers.

## 3.4.1 Exposing Resource-Allocation Decisions

To regulate energy spent on a device or resource, each driver must vector pending allocation decisions to the policy managers. Typical allocation decisions pose questions such as "Given two runnable activities and a resource, which should be scheduled?" or "Given a set of runnable activities and a resource, in which hardware state should the resource be put into?".

Having received a pending resource-allocation decision, the policy manager first answers the decision based on the gathered energy-accounting state and its own policy. It then re-injects updated resource-allocation decisions, thereby ensuring that resource and energy consumption matches the desired constraints. Resource-allocation responses typically contain updated information about how the driver should schedule requests to its resource(s), until the next decision arises.

Note that our framework provides allocation mechanisms for both hardware devices and, where required, for software resources, which corresponds to our recursive approach for energy accounting of software resources. Regulating the client activity on a software resource implicitly regulates the energy consumption of all subordinate hardware devices required to provide that software resource.

### 3.4.2 Interface-Design Considerations

Designing the actual interface for propagating allocation decisions, responses, and power state changes is implementation-specific; we present implementations for both processor and disk devices when presenting our prototype, in Chapter 4. To identify activities, we continue to use the *scheduling domain* construct. When exposing resource-allocation decisions, the driver uses the domain identifier to denote the resource consumers involved in the particular decision.

To give an example of such an interaction, a processor driver exposes preemptions and blockings of its subordinate virtual processors to a scheduling-policy manager. Thread preemption or blocking may occur, for example, if a thread invokes a blocking system call, or if the processor allotment of the current thread ends and a new one must be selected for execution. Having received notice about the pending decision, the scheduling policy manager inspects its own per-thread energy-accounting records and selects an appropriate new thread for execution, according to its scheduling policy. It propagates the selection to the resource driver, which, in turn, realizes the policy by scheduling the thread accordingly.

In the example above, pending decisions are exposed at the time of their occurrence, that is, the instance a thread blocks or is preempted. Likewise, the response follows immediately upon the call as well. Such a protocol may become costly, if resource driver and policy manager reside in different isolated components. We therefore propose to use different interaction protocols, with varying degree of sychronizity, depending on the requirements of the particular energy management policy. We will discuss them in detail in the following paragraph.

## 3.5 Energy-Management Interaction Protocol

Separating policy from mechanism in energy accounting and allocation requires us to propagate information about energy accounting and allocation between resource drivers and policy managers. Multiple communication schemes are thinkable to coordinate the interaction between resource drivers and policy managers; we refer to those schemes as energy-management interaction protocols.

The accrual of accounting data and the occurrence of allocation decisions are defined by resource and driver-internal characteristics. The propagation of those events to policy managers, however, does not necessarily need to occur at the same time. Policy managers may reside in a different protection domain, and crossing boundaries can incur significant processing overhead; a protocol that immediately pushes out accounting or allocation events whenever they arise also has the highest performance overhead.

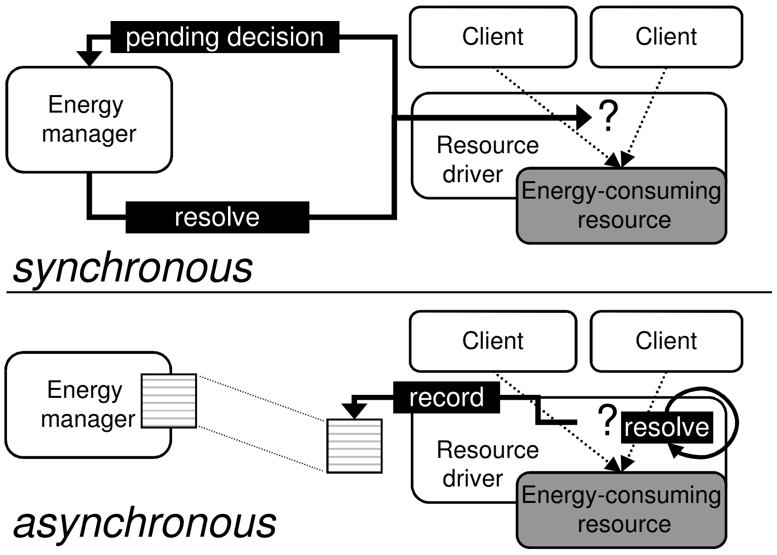

Figure 3.6: Adaptive, exposed resource-management communication. Energy managers can trade timeliness for performance by choosing between low-overhead, asynchronous, and higher-overhead, synchronous communication.

We postulate that the actual accuracy and granularity requirements ultimately depend on the energy-allocation policy. We assume that the time span between accounting and allocation events in the resource provider, and the need to process them in the policy manager is variable, and propose to trade timeliness for performance whenever the particular policy permits it. In our framework, we propose two different energy-management interaction protocols, a synchronous and an asynchronous variant, each with different characteristics in terms of accuracy and performance. The protocols are illustrated by Figure 3.6. We leave it up to policy managers to decide which protocol to use for which particular resource. We will present implementations and an evaluation for both protocols and for both processor and disk devices when presenting our prototype in Chapter 4. In the following, we explain the mechanics and characteristics of the two protocols.

### 3.5.1 Synchronous Interaction Protocol

The synchronous protocol is based on a rendezvous between the resource provider and the policy manager at the time a particular allocation event occurs. The protocol consists of a *callback* from the resource provider to the policy manager and a *callback response* from the manager to the provider. The callback contains accounting and scheduling state normally internal to the resource provider; the callback response, in turn, resolves the pending scheduling decision(s) and potentially changes hardware power states. Since protection domains must be crossed for that purpose, the protocol may become costly. On the other hand, it permits resolving pending resource decisions under direct participation of the policy manager.

### 3.5.2 Asynchronous Interaction Protocol

The asynchronous protocol is based on the idea of tracing events without propagating their occurrence anywhere. The protocol design and mechanics are inspired by previous work on using event logging for multi-processor scheduling [Stoess and Uhlig, 2006, Uhlig, 2005], albeit that work focused on multi-processor systems rather than on energy management. Asynchronous tracing records the occurrence of a particular scheduling event within a data structure shared between resource provider and policy managers. Shared memory does not require transition between protection domains, thus the mechanism allows low-overhead, bulk data transport of energy-accounting and other resource state. The state can later on be evaluated and inspected by the policy manager. Since the policy manager does not directly participate in the communication then, pending resource decisions cannot be resolved instantly.

We therefore additionally declare default, *shortcut* policies that can be used by the resource providers together with the asynchronous communication. If the policy manager selects the asynchronous protocol for a given resource, it implicitly agrees to the shortcut policy that allows the resource provider to resolve the pending decision internally. In other words, the policy manager must map its own energy-management goals to the local shortcut policy in the resource driver.

### 3.5.3 Discussion

Our adaptive communication scheme is based on the insight that there is a fundamental trade-off between exposed and timeliness resource management. Our scheme integrates that trade-off into the resource-management process by offering it to the policy manager as a “knob”. To give an example, our system allows the policy manager responsible for processor scheduling to let the kernel (i.e., the processor resource provider) either synchronously vector out any thread scheduling decision and its resolution — a scheme widely referred to as scheduler activations [Anderson et al., 1991] — or to let the kernel apply an internal default policy such as a proportional-share scheduling scheme [Waldspurger and Weihl, 1994], but, at the same time record the occurrence of scheduling decisions together with other accounting state to a shared buffer. The former scheme allows strict isolation with respect to processor energy (or time, the policy itself is exchangeable) for the price of an expected performance decrease, while the latter scheme expectedly increases performance but gives up fine-grain energy isolation.

Note that, at present, we leave the question of how and when the particular protocol should be chosen open; future work must investigate how the selection of protocol can be facilitated in a flexible way, for example, by offering a dynamically configurable system, or by allowing different scheduling domains to use different protocols at runtime. As of now, however, our approach is limited, both in design and evaluation, to an exploration of the two protocols.

## 3.6 Energy Virtualization

As stated in Section 3.1, a widely used vehicle to provide compatibility in operating systems is platform virtualization, which provides one or more identical copies of the underlying hardware and enables arbitrary applications and operating systems to be executed atop without or with very few modifications. Virtualization layers have become very popular, and can be found in many traditional and modular operating systems. Virtual environments are a striking example where distributed, multi-layered energy management can be beneficial: on the one hand, only guest operating systems can pursue fine-grain energy management and keep up guest-intrinsic application or user specific service demands. On the other hand, only the hypervisor and host-level resource-management subsystems can control global, machine-wide energy requirements and conditions. Host-level resource management also remains the resort that enforces given energy requirements for malicious, defect, or simply energy-unaware guests.

A key constraint of virtualization is that the interface between physical and virtual world is defined through the hardware-like specification of each virtual device. This design results in a strict separation that allows guest-internal software to execute without considering the fact that it does not run on real hardware. While this is advantageous in terms of transparency and separation of concerns, it also leads to semantic loss between the different levels of virtualization environments: information does not easily pass the strict interfaces separating physical and virtual worlds. The result is that resource management is separated into pieces as well: At host-level, hypervisor and driver modules have direct control over hardware devices and their energy consumption, and can obtain coarse-grained per virtual machine information on how energy consumption is spent on the hardware. However, the host level does not possess any knowledge of the energy consumption of individual virtualized applications. The guest operating systems, in turn, have intrinsic knowledge of their own applications, but operate on deprivileged virtualized devices, without direct access to physical hardware.

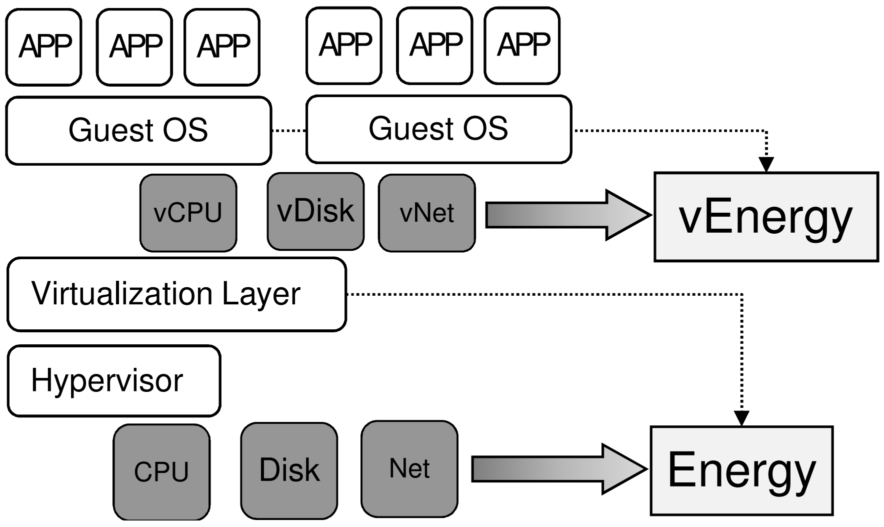

Figure 3.7: Host-level energy-management capabilities enable coarse-grain, but enforceable energy policies; guest-level energy-management capabilities enable fine-grain, application-specific schemes. Virtualization of device-energy effects is key to enable energy management for virtual devices.

To overcome the semantic loss, our framework supports energy accounting and allocation not only for physical devices but also virtual devices. That is, our framework enables guest operating systems using a virtual device to determine their own energy share on that device. For virtualization environments, the advantage of having support for virtualized energy is actually twofold: first, it enables guest-level resource-management subsystems to leverage their application-specific knowledge for performing fine-grain energy management. Second, it enables drivers and other operating-service modules also running in a virtual machine to recursively determine the energy for their own services.

The main difference between a virtual device and other software services and abstractions lies in its interface: a virtual device closely resembles its physical counterpart. For virtualization of energy effects, however, this constraint is somewhat artificial, as most current hardware devices offer no direct way to query energy consumption anyway, and require estimation schemes based on other device characteristics. For those cases, we require that the virtualization layer emulates the relevant behavior for the virtual devices, supporting energy estimation in the guest without major modifications to the guest's energy accounting. Note that we are not concerned with providing exact legacy-compatible hardware power semantics for virtual devices such as the virtualized power states proposed by Nathuji and colleagues [Nathuji and Schwan, 2007]; rather, our ultimate goal is to enable the guest to use the same driver for virtual and for real hardware.

# Chapter 4

# Application to an L4-Based Operating System

Based on our design principles, we have developed a distributed, multi-level energy-management framework for a microkernel-based component operating system. We use an instance of the L4 microkernel family as the privileged microkernel. Our system permits the development of custom operating-system modules, but simultaneously retains legacy compatibility to existing applications by means of a platform-virtualization layer. Our environment runs on IA-32 hardware and supports Linux 2.6.9 guest operating systems.

Our prototype currently supports management of two main energy consumers, processor and disk. Processors are directly managed by the microkernel, while the disk is managed by a special user-level driver. Our prototype supports processor and disk-energy management both for physical and for virtual devices. To that end, it features distributed and recursive mechanisms for accounting and allocating energy both to complete virtual machines and to individual virtualized applications. Our prototype system forms an instance of a modular operating system that can be practically used in modern computing environments. As such, we consider it a good and realistic candidate for evaluating our energy-management design principles and considerations.

The chapter is organized as follows: Section 4.1 presents our prototypical architecture. Section 4.2 presents the device models we used for processor and disk-energy estimation. Section 4.3 presents our implementation of exposed and distributed energy accounting. Section 4.4 presents our implementation of exposed energy allocation. Section 4.5 presents how we preserve compatibility to legacy resource management, while enabling support for energy-aware improvements. Section 4.6 finally explains how we implemented exemplary policies for processor and disk-energy management, both at the host-level and at the guest-level of our operating system.

## 4.1 Prototype Environment

Our application and evaluation prototype is a microkernel-based component operating system. The system permits the development of individual, custom-built operating-system modules and applications at user level; simultaneously, it retains legacy compatibility to applications written for existing operating systems by means of a platform-virtualization layer (Figure 4.1). Legacy compatibility currently extends to applications built for the Linux kernel.

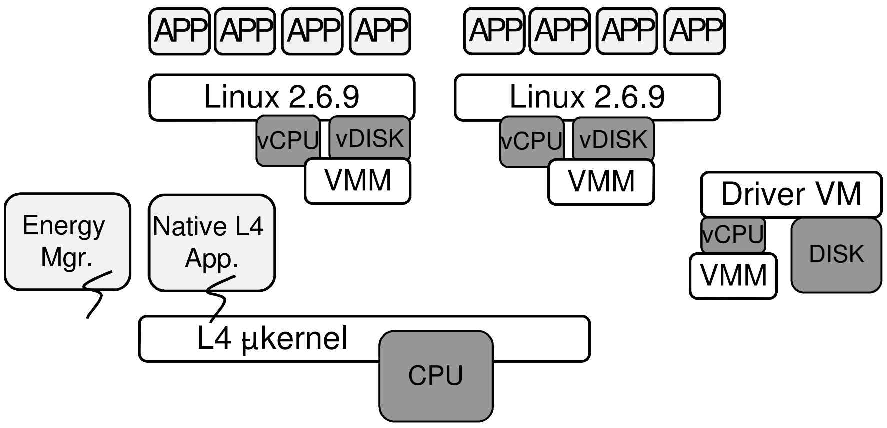

Figure 4.1: Prototype architecture. Our energy-management framework is integrated into an L4-based component operating system. Virtualization is a key part of the operating system, catering for both application compatibility and device driver reuse.

We use an instance of the L4 microkernel family as the privileged microkernel. L4 provides core abstractions for isolation and resource management, but leaves more complex operating-system functionality up to user-level components. Likewise, L4 only provides basic abstractions for device isolation and interrupt management, and leaves all other coordination of peripheral devices up to external user-level drivers. Our platform-virtualization layer allows modified Linux guest operating systems to run on top of L4 instead of on bare hardware. For managing guest–operating-system instances, the prototype includes a user-level virtual-machine monitor, which maps the virtualization logic onto L4's core abstractions. To provide user-level disk-driver services, our framework also uses virtualization, and employs special device driver virtual machines, each of which reuses standard Linux disk-driver logic for hardware control.

With respect to energy management, our prototype currently supports two main energy consumers: processor and disk. To that end, our prototype implements exposed mechanisms for distributed energy accounting and allocation. For policy management, our prototype features a host-level policy manager responsible for controlling the energy consumption of physical processors and disks. It is complemented by an optional energy-aware guest operating system redistributing its host-level power allotment among its own applications. Since the energy-aware guest operating system requires virtualization of the energy effects of processor and disk, our prototype also implements a recursive accounting and allocation scheme for virtualized devices.

The remainder of this section presents our basic prototype architecture. We first describe the principles of the L4 microkernel as they are required to understand our energy-management prototype. We afterwards describe the core design of our L4 based virtualization layer. We finally present the architecture of our energy-management prototype which we have built on top of L4 and the and L4 virtualization environment.

### 4.1.1 The L4 Microkernel

L4 is the privileged kernel used in our prototypical component operating system. Since our system features a platform virtualization layer, the microkernel also acts as hypervisor; we will use the term microkernel to denote both. L4 was initially developed by Jochen Liedtke at GMD, Germany, at IBM Watson Research Center, NY, USA, and at the University of Karlsruhe, Germany. Liedtke implemented early kernel versions in assembly code for x86 processors. Several follow-up versions, with diverging goals, application programming interfaces, and corresponding implementations, were developed at Dresden University of Technology, Germany, at the University of New South Wales, Sydney, Australia, at NICTA in Sydney, Australia, and at the University of Karlsruhe, Germany. At present, L4 is available for a large number of architectures and platforms, including IA-32, AMD-64, ARM, PowerPC, Alpha, and MIPS.

#### L4Ka::Pistachio

In our prototype, we use a recent IA-32 implementation of the L4 microkernel, code-named *L4Ka::Pistachio*. L4Ka::Pistachio is an ongoing research effort of the System Architecture Group at the University of Karlsruhe [L4 Development Team, 2009a]. L4Ka::Pistachio implements the L4 specification, Version X.2 Rev. 6, which is co-authored by the members of the System Architecture Group [L4 Development Team, 2009b]. We will hence use the term L4 for both the abstract kernel and our concrete implementation.

### L4 Mechanisms and Abstractions

L4 is a state-of-the-art microkernel, which provides a minimal set of abstractions designed to build extensible systems on top [Liedtke, 1995]. L4 provides two core abstractions for user-level resource management: threads and address spaces; it further provides two mechanisms for inter-component coordination and resource management: synchronous communication (IPC) and recursive rights delegation (mapping); finally, it provides basic abstractions for device and interrupt handling:

**Threads** provide the context for three different operating-system concepts: execution, communication, and scheduling. In their first role, L4 threads serve as the basic abstraction of user-level control flow, by referring to execution state such as instruction and stack pointer, or general-purpose registers. In their second role, threads serve as endpoints for the kernel IPC primitive. L4 therefore associates IPC state with each thread, to keep track of attempted or ongoing IPC operations and their particular form and content. In their last role, L4 associates scheduling information such as time-slice lengths, time quanta, or priorities with each thread, and employs an internal (currently priority-based round-robin) scheduler that dispatches the threads using a default policy.

**Address Spaces** are the abstraction for memory protection and isolation [Liedtke, 1995]. On x86 platforms, the address space abstraction also extends to the I/O space connecting peripheral devices [Stoess, 2002]. Thus L4 enables isolation for memory and devices using the same core abstraction. Address spaces are passive in that that they are not directly accessible from user level, but only via the threads living within.

**IPC** is a rendezvous-based, synchronous communication mechanism in L4. Both a send and a receive operation exist. While L4 permits sending to unique destinations only, it allows the use of wild-cards as receive destination. As system calls are expensive, and for reasons of atomicity, L4 also offers coalesced send and receive operations. L4 supports message contents of different complexity, with registers, strings, and virtual-memory mappings as transferable objects. Finally, L4 supports the specification of timeouts for blocking IPCs.

**Mapping** is the main primitive for user-level resource management in L4. A mapping provides a notion of a transferable resource permission. By means of mapping, L4 allows address spaces to transfer their right to access a given resource — a memory page or an I/O port on x86 hardware — to other address spaces. The resource right can either be copied into the destination (meaning that it remains in the source address space), or moved (meaning that it is removed from the source address space). Furthermore, the resource right can be restricted underway. As the destination address space is free to re-map its own rights to other, subordinate address spaces, resource mapping occurs in a a recursive manner. In L4, the mapping primitive leverages the IPC mechanism presented beforehand; that is, an IPC can contain special message items denoting a resource mapping from the source to the destination. For revocation of rights, however, L4 provides separate, asynchronous kernel primitive that does not require explicit consent from any of the existing right receivers.

**Device and Interrupt handling** With respect to management of peripheral devices, L4 only provides basic abstractions for device-memory isolation and interrupt delivery, and leaves all other management up to external user-level drivers. Isolation of I/O device memory is handled via recursive right delegation, as already explained. Interrupts and exceptions are handled by means of IPC. For each external interrupt line, L4 creates an in-kernel interrupt thread, which will send a special interrupt request message to an attached handler thread whenever the interrupt occurs. Similarly, whenever a synchronous processor exception occurs, L4 synthesizes an exception message on behalf of the faulting user level thread to a designated per-thread exception handler.

### 4.1.2 L4-Based Virtualization

Providing compatibility to existing and widely-used operating systems (such as Linux or Windows) can be termed a “classic” issue for microkernel-based systems. Being able to run existing applications on a microkernel is not only desirable for the practical purpose of enhancing its general usability and prevalence; it is also often a prerequisite for scientific research, since evaluation of any operating system typically requires running standard benchmarks or near-realistic scenarios. Consequently, approaches to shoehorning some monolithic operating system on top of a microkernel for compatibility purposes have a comparatively long history and tradition in the history of microkernels [des Places et al., 1996, Härtig et al., 1997, Krieger et al., 2006]. Our prototype makes use of L4-based virtualization; in the following, we therefore first briefly sketch the history and origins of such approaches, and then detail our the L4Ka virtualization environment we have used for implementation.

### L4Linux and Derivatives

The first “virtualization” endeavor for the L4 microkernel was L4Linux, a port of the Linux operating system on top of the L4 microkernel. L4Linux is fully binary compatible to Linux on x86 platforms — however, the authors needed to change a significant portion of the architecture-dependent parts of the Linux kernel code base. Originally L4Linux was not developed with virtualization in mind; rather, the effort was undertaken to evaluate the performance of microkernel-based systems [Härtig et al., 1997]. However, as virtual-machine monitors regained interest afterwards, particularly those for commodity hardware platforms [Rosenblum and Garfinkel, 2005], some debate was spent on the question whether L4Linux could be termed a virtualization solution, and whether microkernels and hypervisors share common goals and techniques [Hand et al., 2005, Heiser et al., 2006]. Research has found that there is a substantial similarity between para-virtualization systems and microkernel systems alike L4Linux: para-virtualization, as in in the Xen virtual-machine monitor [Barham et al., 2003] or VMware’s VMI approach [Amsden et al., 2006], means porting an existing guest operating system to a hypervisor; L4Linux means porting Linux to a microkernel. Moreover, there even exist some approaches to providing true virtualization on L4 using x86 processor extensions [Biemueller and Dannowski, 2007].

As a general result, virtually all approaches to commodity operating-system compatibility for L4 are nowadays advocated as virtualization environments [Härtig et al., 2005, Heiser, 2009, Uhlig et al., 2004, LeVasseur et al., 2004]. Still, however, most of those approaches — including Karlsruhe’s virtualization environment, which serves as the basis for our energy management framework — can be seen as refinement of the original L4Linux approach.

### L4Ka Virtualization Environment

Our energy-management prototype leverages the L4Ka para-virtualization environment. The environment is developed and maintained by the L4 team at the University of Karlsruhe; it is a derivative of the L4Linux approach, allowing modified Linux guest operating systems to run on top of L4 (Figure 4.2). At present, the environment supports Linux a para-virtualized version of the Linux 2.6.9 kernel, which we also use in our prototype. In the environment, the L4 kernel itself acts as the privileged hypervisor, responsible for partitioning processor(s), physical memory, device driver memory, and interrupts. For managing guest–operating-system instances, the environment includes a user-level virtual-machine monitor running as a normal L4 program. The virtual-machine monitor provides the virtualization service based on L4’s core abstractions; it can be described as an interface layer that translates virtualization API invocations — sensitive instructions, in other words — into microkernel API invocations of the underlying L4 architecture. For performance reasons, a large fraction of the user-level translation code executes in-place, within the address space of guest operating systems. If necessary, for instance for reasons of security, the in-place part calls into an external monitor module, which runs in a separate address space and has extended privileges. In the following, we briefly explain the mechanics of the L4Ka virtualization environment for processor and memory, as they are required to understand our implementation of our energy-management prototype; more details can be found in [LeVasseur, 2009, LeVasseur et al., 2004, LeVasseur et al., 2008, Uhlig et al., 2004].

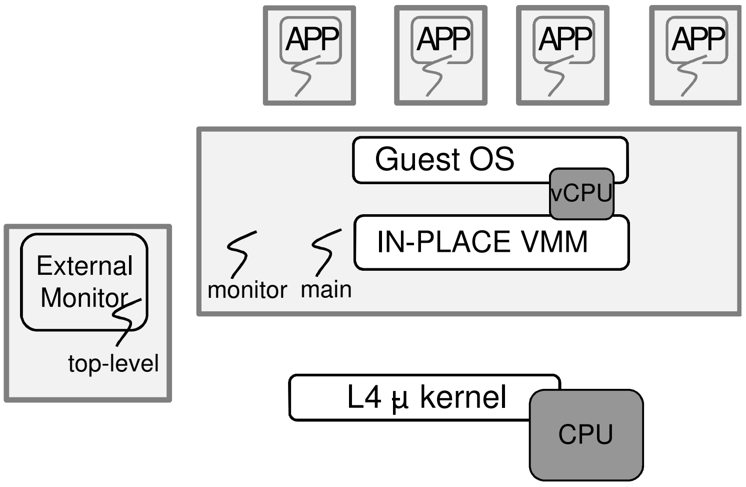

Figure 4.2: L4Ka virtualization architecture. An in-place virtual-machine monitor part virtualizes guest–operating-system functionality within the guest address space, using two L4 threads, monitor and main. If necessary, for instance for reasons of security, the in-place part calls into an external module, which runs in a separate address space and has extended privileges. For each guest address space and virtual processor, the virtual-machine monitor spawns an additional L4 thread, to host guest user code. Each user thread on a virtual processor is dispatched by the main thread on that processor.

### Processor and Memory Virtualization

The L4Ka virtual-machine monitor virtualizes physical processors by mapping kernel and application contexts of guest Linux instances to corresponding L4 threads. Likewise, the virtual-machine monitor virtualizes physical memory by mapping guest kernel and application memory onto different L4 address spaces. The resulting architecture constitutes a three-level hierarchy: At the highest level, a per-processor top-level virtual-machine dispatcher thread runs in the address space of the privileged external monitor and serves as dispatcher for virtual processors. At the second level, each virtual processor is represented by an address space and two threads representing the guest kernel. One guest kernel thread, named *main*, serves as the execution context for the virtualized guest–operating-system code; the other thread, named *monitor*, acts as the in-place resource manager and scheduler of main. Finally, at the third and lowest level, the virtualization environment spawns an L4 address space per user-level application, and, within it, an L4 thread per virtual processor, to execute guest user code. Each user thread on a virtual processor is scheduled by the main thread on that processor.

To give two examples how processor and memory virtualization works, let us consider the functionality for returning from the guest kernel to a guest application. and the workings of memory virtualization. For returning from the guest kernel to a guest application, the virtual-machine monitor logic translates the `iret` instruction, issued by the guest kernel on the main thread to transfer control to a user-level program, into an IPC reply message to the waiting user-level thread representing that program. The reply message transfers control from the guest kernel's execution context to the application that was originally selected by the Linux kernel for activation. For memory virtualization, the virtual-machine monitor translates the modification of a guest page table by the Linux kernel into an invocation of the associated virtualization routine running in-place, in the context of the main thread. Should the in-place module afterwards require a mapping of physical page frame, it requests, via L4 IPC, the external monitor to grant it access to that frame. The external monitor in turn responds to that request using again L4 IPC and a piggybacked page mapping.

### 4.1.3 User-Level Device Drivers

L4 leaves most of the management of peripheral devices to user-level driver components, and only provides basic abstractions for I/O-memory isolation and for interrupt delivery. As a result, in an L4-based system, peripheral devices are managed mainly by a corresponding user-level device driver. The driver must be granted access to device registers by means of L4 mappings; also, it must register itself as interrupt handler for device-specific interrupts. The driver is then responsible for managing the low-level device functions such as hardware states, control registers, and interrupt delivery and acknowledgement; it is also responsible for exporting some sort of device abstraction and for mediating shared access to the device from multiple clients. Research has explored multiple techniques to facilitate the development of user-level device drivers for microkernels [Liedtke et al., 1991, Goel and Duchamp, 1996, Forin et al., 1991]. In general, however, writing and maintaining drivers from scratch is a fairly tedious task, requiring a lot of development effort both for low-level, device-specific code and for user-specific functionality.

#### L4 Driver Reuse

An alternative approach to provide user-level device drivers is to *reuse* driver code, for instance from operating systems with an existing driver code base such as Linux or Windows. Many research approaches on driver reuse required a special interface or glue code [Maren, 1999, Appavoo et al., 2002b] transplanting the driver logic from its original environment into the microkernel world. To overcome the problems associated with this glue code, we leverage a different approach for driver reuse, which is — again — based on virtualization. Details can be found in [LeVasseur et al., 2004]; shortly described, our approach runs the unmodified device driver, together with its original operating system as a glue layer, in a virtual machine. Porting the driver then becomes a matter of developing a fairly simple translation layer mediating requests between clients and the guest operating system the driver is embedded in.

### 4.1.4 Energy-Management Framework

Our energy-management prototype currently supports two main energy consumers: processor and disk. In our prototype, processors are directly managed by L4, while the disk is managed by a special device driver virtual machine implementing the driver-reuse approach. To implement our design concepts, we have extended the microkernel and the driver virtual machine with appropriate functionality that provides and exports accounting and allocation of processor respectively disk energy to external policy managers.

For processor-energy accounting, we leverage an existing estimation scheme based on hardware performance counters. For disk-energy accounting, we use a time-based approach that attributes the disk power consumption linearly to different device states. To provide exposed processor-energy allocation, we have implemented a modified version of the L4 kernel, which exports an interface to direct kernel processor scheduling from user-level policy managers. Similarly, the disk-driver virtual machine exposes an interface that allows disk requests from individual clients to be regulated remotely from the host-level policy manager.

#### Host-Level Energy Management

As a control center for host-level energy management, our prototype features a policy manager application (hence called *energy manager*), responsible for controlling the energy consumption of physical processors and disks. Using our dis-

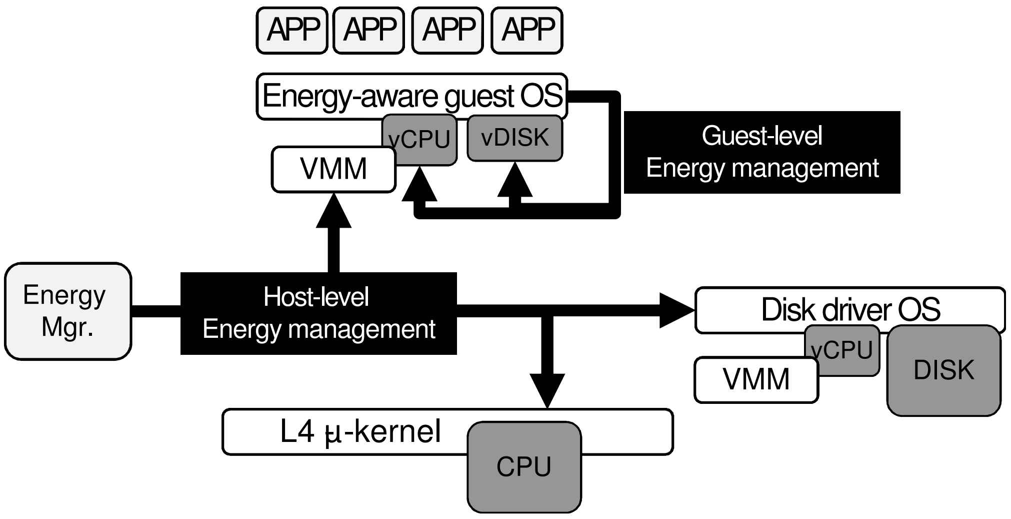

Figure 4.3: A host-level policy manager controls system-wide energy constraints and enforces them among guest virtual machines and native L4 applications. An energy-aware guest operating system is capable of performing its own, application-specific energy management.

tributed accounting scheme, the energy manager periodically obtains, per client virtual machine, the processor and disk-energy consumption from the microkernel and disk-driver virtual machine, and matches them against a given power limit. To bring both in line, it responds by invoking the exposed allocation mechanisms for the processor and disk devices.

### Guest-Level Energy Management

The energy manager is responsible for managing complete virtual machines and native L4 applications. To let guest virtual machines pursue more fine-grained, application-level energy management, we have complemented the host-level part with an (optional) energy-aware guest operating system, which redistributes its virtual machine wide power allotment among its own, subordinate applications. In analogy to the host-level, where physical devices are allocated to virtual machines and L4 threads, the guest operating system regulates the allocation of virtual devices to ensure that its applications do not spend more energy than their alloted budget.

Since the energy-aware guest operating system requires virtualization of the energy effects of processor and disk, our prototype implements the recursive accounting and allocation scheme for virtualized devices. That is, the virtual-machine monitor creates, for each client virtual machine using a processor or disk device, a local view on the current energy consumption on its virtual processor or disk devices; it thereby enables the guest to pursue its own energy-aware resource management. The virtual energy-accounting records are based on the physical energy records from the microkernel and disk-driver virtual machine.

Note, that our energy-aware guest operating system is an optional part of the prototype: it provides the benefit of fine-grained energy management for Linux-compatible applications. For all energy-unaware guests, our prototype resorts to the coarser-grained host-level management, which achieves the constraints regardless whether the guest-level subsystem is present or not.

## 4.2 Device-Energy Models

In the following section, we present the device-energy models that we use to base for processor and disk accounting on. We generally break down the energy consumption into *access* and *idle consumption*. Access consumption consists of the energy spent when using the device. Idle consumption, in turn, is the minimum power consumption of the device, which it needs even when it does not serve requests. Many current microprocessors support multiple active power modes, and similar technology exists in multi-speed disks. There is no conceptual limit to the number of power states, and by design, we decouple the power states of multi-speed devices from the accounting of its idle costs. Implementation-wise, however, we are currently limited to two device states only: access and idle.

### 4.2.1 Processor-Energy Model

In the following section, we will detail how we estimate the processor energy at runtime, using an estimation model based on processor performance counters. Current IA-32 processors do not have any support for directly determining their energy consumption at runtime. Instead of direct measurement, our prototype must therefore resort to an estimation scheme to monitor processor-energy consumption.

#### Non-Linear Correlation of Processor Time and Energy

A fairly trivial approach to estimate processor energy is to assume that the processor-energy consumption of an activity correlates linearly with its time spent on the processor. Research has shown that such a method tends to yield accurate and stable results for early IA-32 architectures such as Pentium II [Rohou and Smith, 1999]. However, experiments also have shown that the approach is not accurate enough for more modern IA-32 microprocessors. Even for compute-intensive applications, generating a hundred percent of processor load, the variance in power consumption can be as much as about 20 W on a Pentium IV processor [Bellosa et al., 2003]. This is a considerable amount, given that the idle-power consumption of that processor is about 12 W. For our framework, we therefore resort to an energy estimation scheme that is more accurate and provides a higher resolution than the simple processor-time based approach.

### Performance-Counter–based Energy Estimation

Our framework uses a different approach and bases processor-energy estimation on the rich set of performance counters featured by modern IA-32 microprocessors. Originally designed for evaluating performance characteristics of processors or the workload running atop, performance counters give detailed information on the occurrence of processor-internal events, like past clock cycles, cache misses, or pipeline stalls. As such, however, performance counters can also be (ab-)used for estimating, at runtime, who has used how much processor energy on the processor.

| **Event** | **Pentium IV 2 GHz [nJ]** | **Pentium D 3 Ghz [nJ]** |
|---:|---:|---:|
| time-stamp counter | 6.17 | 14.2 |
| unhalted cycles | 7.12 | 12.9 |
| $\mu$op queue writes | 4.75 | 0.8 |
| retired branches | 0.56 | 8.4 |
| mis-predicted branches | 340.46 | 234.2 |
| memory retired | 1.73 | 5.98 |
| MOB load replay | 29.96 | 43.2 |
| ld miss 1L retired | 13.55 | 8.81 |

Table 4.4: Set of performance-counter events and their energy contributions on two Pentium microprocessors.

Performance-counter–based processor-energy estimation has been explored and documented in [Bellosa et al., 2003, Joseph and Martonosi, 2001, Kellner, 2003]; for the sake of clarity, and since we use a custom calibration scheme, we will shortly describe the approach here. With the performance-counter approach, each performance-counter event is assigned a weight representing its contribution to the overall processor energy. The weights are retrieved by means of of a calibration procedure, which counts events during test runs of applications with constant and known power consumptions. For $m$ sample applications and $n$ different events, the calibration results in an energy vector $\vec{e}$:

$$\vec{e} = (e_1, e_2, ..., e_m)$$

respectively a matrix $P_{mn}$ denoting $n$ performance-counter–event occurrences for each of the $m$ sample applications:

$$P_{mn} = [p_{i,j}] \quad (1 \leq i \leq m, 1 \leq j \leq n)$$

The problem of estimating the energy from performance counters can be described as an optimization problem: find $n$ unknowns solving $m$ equations. For the sake of simplicity, we assume linear correlations between performance-counter events and energy, thus the problem constitutes an (over-determined) system of $n$ unknowns and $m$ linear equations. A solution for such a problem can be approximated using standard mathematical algorithms such as the Least-Squares method or the Simplex algorithm for linear-programming problems (see, e.g., [Rao, 1996]). We used the Least-Squares method in our implementation, which tries to find $\vec{x^T} = (x_1, x_2, ..., x_n)$ minimizing the sum of the squared differences between the data values and their corresponding modeled values:

$$\min_{\vec{x}} ||P_{mn}\vec{x} - \vec{e}||_2$$

To simplify estimation, and since we assume each performance-counter event to contribute *positively* to the overall energy consumption, we further add the following constraint:

$$x_i \geq 0 \quad (1 \leq i \leq n)$$

### Performance-Counter Calibration for Pentium Processors

Table 4.4 lists the weights for a Pentium IV processor and for the Pentium D processor that we used for evaluation. The weights for the Pentium IV processor were taken from the study in [Kellner, 2003], which uses about a dozen different sample applications stressing different parts of the processor. To determine the weights for the Pentium D processor used in our evaluation, we ran similar calibration experiments using the same sample applications as in the study, but with a different solver: while the original study used `dqed`, a `netlib` Fortran subroutine implementing the least squares method [Hanson and Krogh, 1995], we used the `llsp` subroutine from OpenOpt, a universal numerical optimization package that interfaces with many different algorithms for solving optimization problems [National Academy of Sciences of Ukraine, Cybernetics Institute, 2009].

The particular selection of performance counters is a matter of experimenting empirically during calibration. Quite promising correlations were found with counters for ALU integer operations, load/store operations and cache references [Bellosa et al., 2003]. The sample applications consist of three groups: the first group stresses the processor ALU, for instance by operating entirely on registers using instructions such as `bswap` or `xor`; the second group operates on registers and memory (including caches); the third and final group consists of other programs performing standard computer algorithms such as a checksum algorithms, or the cryptographic algorithms SHA-1 and RIPEMD-160.

Note that, while both the Pentium IV and the Pentium D offer a fairly large amount of eight performance-counter registers that can be read out concurrently, many modern processors such as Intel Atom or Core 2 processors only two such registers. Recent research has shown, however, that a more elaborate and general calibration procedure than the one described above can yield viable energy models even with the limited number of registers available in contemporary microprocessors [Snowdon, 2009]. We chose the current model for pragmatic reasons, since an implementation was already available to us; we do not see, however any conceptual limit to extend our current formulation with more sophisticated models or calibration schemes.

#### Energy Estimation at Runtime

Once the weights are determined, estimating the energy at runtime becomes a rather simple task (See Figure 4.5): to obtain the processor-energy consumption during a certain period of time, for instance, during execution of a virtual machine, the framework sums up the number of events that occurred during that period, multiplied with their corresponding weights. Distinguishing idle from access energy is also straightforward: the time-stamp counter, which counts clock cycles regardless whether the processor is halted or not, yields an accurate estimation of the processor's idle consumption.

### 4.2.2 Disk-Energy Model

Disk drives are a major contributor to the overall power consumption of a computer system. Although a typical modern disk drive shows a power consumption in the range of 5 W-15 W, which may seem relatively low if taken individually, the storage subsystem of a server or data centers typically consists of a fairly large amounts of such disks, often organized in an array or a multi-tier disk hierarchy. As a result, the overall power consumption of the storage subsystem can be substantial, particularly in server systems and data centers, and may very well exceed the power consumption of the processors [Carrera et al., 2003, Schulz, 2007].

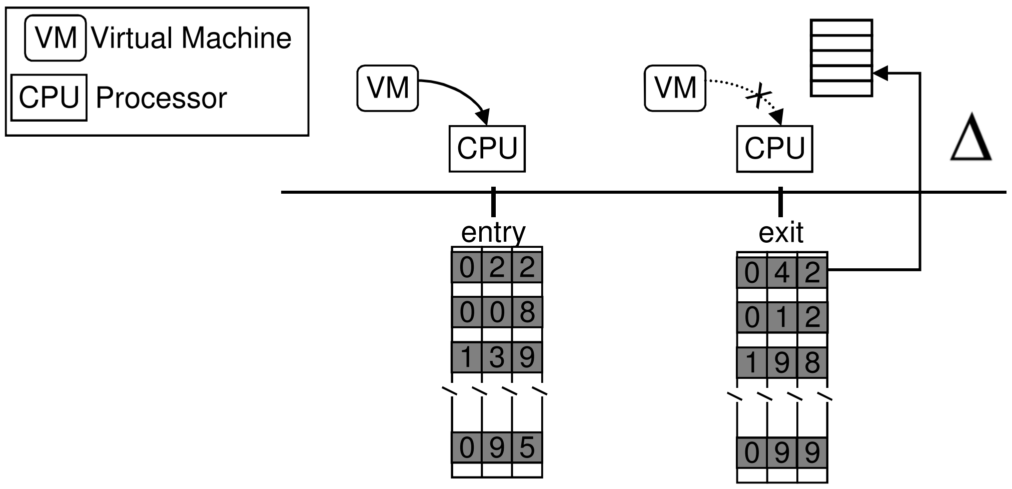

Figure 4.5: Runtime energy estimation of a virtual machine via performance counters. The advances of performance-counter values between context switches reflect the energy consumption of each individual virtual machine.

As with processors, existing disk systems typically do not provide support (e.g., sensors) for measuring their energy consumption at runtime. We therefore again resort to an estimation scheme to monitor disk-energy consumption. We currently do not consider mechanisms for suspending a disk in our implementation; for lack of availability, we also do not consider multi-speed disks as well. We thus distinguish two different power states only: access and idle. Note again that, as with the processor-energy estimation, we did not focus on developing or exploring accurate disk-energy modeling schemes in our work; rather, we chose an estimation model easy to implement but sufficiently accurate. We again do not see any conceptual limit, however, to extend or replace the current formulation with more sophisticated schemes, such as the one recently proposed in [Allalouf et al., 2009].

### Constant and Fixed Portions of Disk-Energy Consumption

Storage components consume energy when idle, which fits our model that breaks device-energy consumption into an idle and an access portion. The idle part is required to drive the spindle and electronic control components; since we currently do not support spinning down disks in our implementation, we assume that the idle part does not vary with changing workloads at runtime. The access portion, in turn, is induced by disk seek resulting from disk activity, and must be attributed to those applications causing it.

**Request-Based Disk-Energy Estimation**

Our disk-energy estimation model differs from the processor model in that it uses a time-based approach rather than event sampling. Instead of attributing energy consumption to events, we attribute energy consumption to different device states, and calculate the time the device requires to transfer requests of a given size.

To determine the transfer time of a request — which is equal to the time the device must remain in active state to handle it —, we divide the size of the request by the disk's transfer rate in bytes per second. Although we ignore several parameters that affect the energy consumption of requests, (e.g., seek time or the rotational delays), our evaluation shows that our simple approach is quite accurate. Our observation is generally substantiated by the study in [Zedlewski et al., 2003], which indicates that a 2-parameter model (distinguishing between idle and access energy) is inaccurate only because of sleep-modes, which we disregard for our approach anyway.

## 4.3 Distributed Energy Accounting

Our framework requires each driver of a resource to determine its energy consumption and to account the consumption to clients. Resources can be physical devices, like a disk or network controller, or refined software abstractions, like socket or a virtualized hardware device. Clients may be guest applications, native L4 applications, and complete virtual machines.

For accounting energy consumption of physical devices, our infrastructure uses the device-energy models presented above: Access consumption is charged directly to each request, after the request has been fulfilled. The idle consumption, in turn, cannot be attributed to specific requests; rather, it is alloted to all clients in proportion to their respective utilization. For use by the energy manager and others, the driver grants access to its accounting records via shared memory and updates the records regularly.

In this section, we explain how we implemented runtime energy accounting and allocation for processor and disk devices. We detail how these mechanisms enable both virtual-machine wise and native-application wise energy accounting for hardware devices, and recursive energy accounting for virtualized software devices.

### 4.3.1 Processor-Energy Accounting

Our processor-energy model relies on collecting performance-counter values at runtime. To accurately attribute the processor energy to virtual machines or other user contexts, we record the performance counter at times of preemption respectively dispatching of L4 threads. The recording mechanism instruments context switches between L4 threads within the microkernel; at each switch, the current values of the performance counters must be recorded and associated with the thread preempted at the switch. Since our policy management resides outside the kernel, the records must be made accessible to user level, for use by policy-manager modules. According to the design principle of adaptive energy-management interaction protocols (Section 3.5), we provide two different mechanisms for energy accounting, a synchronous and an asynchronous variant.

### Synchronous Propagation of Performance Counters

The synchronous variant exports performance counters at the time of occurrence, that is, on context switches. The variant consequently requires each L4 context switch to be propagated to user level. However, a context switch constitutes both an accountingx and an allocation event — an old thread is preempted and a new thread must be selected — and is therefore important both for exposed energy accounting and allocation. We therefore use a unified notification scheme for both accounting and allocation, called *preemption IPC*. We present the scheme in detail in Section 4.4.1; shortly described, a preemption IPC is a message that L4 to a designated user-level scheduler thread, which serves as the user-level policy manager of the preempted thread (Figure 4.6). Being notified about the context switch, the policy manager then takes a snapshot of the performance-counter values relevant for energy management and records the snapshot into a shared memory buffer for analysis.

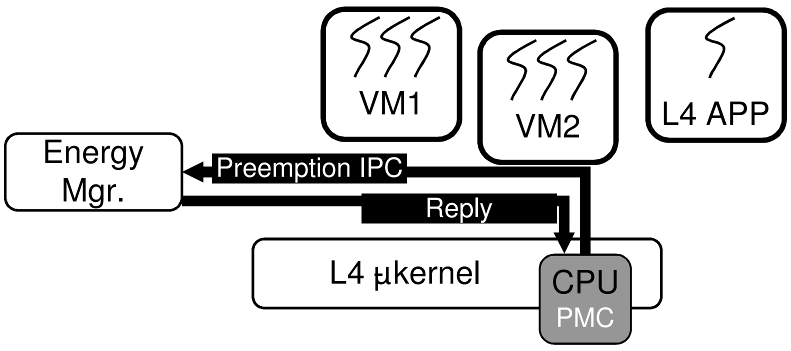

Figure 4.6: Synchronous processor-energy accounting: L4 vectors thread preemption events to user-level policy managers. The policy manager records a performance-counter sample, estimates the energy consumption of the past interval and re-injects pending scheduling decisions with a reply.

### Asynchronous Propagation of Performance Counters

Since control transfers between kernel and user space are costly, our framework alternatively allows the policy manager to select an asynchronous accounting mechanism, which separates the occurrence of performance-counter accumulation (at the time of context switches) from their analysis and the derivation of the energy consumption. The price for retrieving records asynchronously is a potential time delay since the policy manager is not notified instantly.

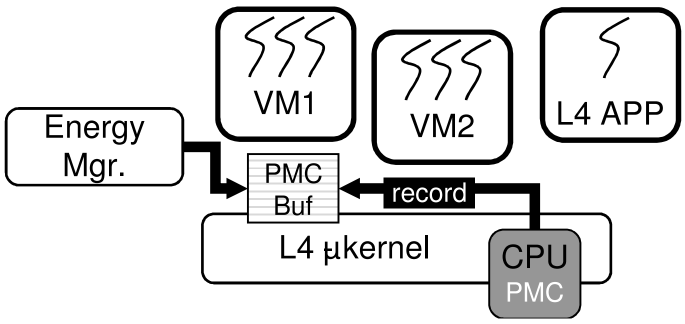

Figure 4.7: Asynchronous processor-energy accounting: L4 collects the performance-counter traces at thread preemption events and records them in a memory buffer, which is also accessible to the policy manager. Individual L4 threads can be pooled based on their scheduling domain identifier; L4 only records transitions between domains.

The asynchronous accounting variant instruments context switches between L4 threads; at each context switch, the instrumentation code records the current values of the performance counters into a buffer in memory, which is shared between L4 and the user-level policy manager (Figure 4.7). The instrumentation code also records the scheduling-domain number of each thread, which can be freely set by the policy manager. We assume that the identities of individual threads within a scheduling domain are not relevant to the user-space policy manager; for reasons of performance, L4 therefore ignores context switches between such threads and only records those transitions taking place in between domains.

### Processor-Energy Estimation and Accounting

With either of the mechanism, the user-level energy manager is provided with a record of performance-counter snapshots at the time of thread (or scheduling-domain) context switches. To estimate and account processor-energy consumption, the energy manager must analyze the record and derive from it the consumption of each client virtual machine or application. To attribute a portion of the estimated energy consumption to an application or virtual machine, the manager sums up the advances of performance counters during scheduling times of all threads belonging to the corresponding application or virtual machine. The advances are weighted with energy weights according to our processor-energy model.

Rather than charging the complete energy consumption to the active virtual machine or application, the energy manager subtracts the idle cost and splits it between all virtual machines and applications running on that processor. The time-stamp counter, which is included in the recorded performance counters, provides an accurate estimation of the processor's idle cost:

```
/*
 * apportion idle energy (pmc[0] = TSC)  to all clients
 */
for (id = 0; id < max_clients; id++)
   client[id].pidle +=
        weight[0] * pmc[0] / max_clients;

/*
 * calculate access energy (pmc[1] ... pmc[8])
 * charge to current clients
 */
for (p=1; p < 8; p++)
  client[cur_id].paccess += weight[p] * pmc[p];
```

## 4.3.2 Disk-Energy Accounting

To provide disk service, our framework reuses legacy Linux disk-driver code by executing it inside a virtual machine, an approach originally proposed in [LeVasseur et al., 2004]. The driver functionality is exported via a translation module that mediates requests between the device driver and external client virtual machines (Figure 4.8). The translation module runs within the same address space as the device driver and handles all requests sent to and from the driver by means of an extra server thread. It receives disk requests from client virtual machines, translates them to basic Linux block I/O requests, and passes them to the original device driver. When the device driver has finalized the request, the module again translates the result and returns it to the client virtual machine.

### Request-Based Disk-Energy Accounting

The translation module has access to all information relevant for accounting the energy dissipated by the associated disk device. We implemented accounting completely in this translation module, without changing the original device driver. The module estimates the energy consumption of the disk using the energy model

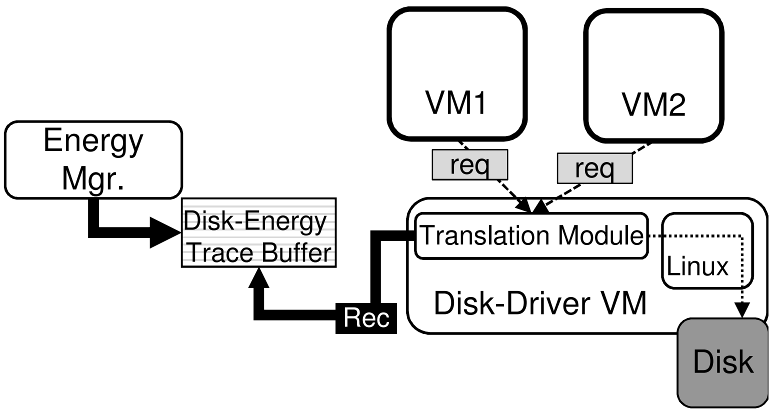

Figure 4.8: Disk-energy accounting. The translation module in the driver estimates the disk energy, apportions it to clients, and stores the energy-accounting data in a memory buffer shared between the driver and the energy manager.

presented above. When the device driver has completed a request, the translation module estimates the energy consumption of the request, depending on the number of transferred bytes:

```
/* estimate transfer cost for size bytes */
client[current].daccess += (size / transfer_rate) *
(active_disk_power - idle_disk_power);
```

Because the idle consumption is independent of the requests, it does not have to be calculated for each request. However, the driver must recalculate it periodically, to provide the energy manager with up-to-date energy-accounting records of the disk. For that purpose, the driver invokes the following procedure periodically every 50 ms:

```
/*
 * estimate idle energy since last time
 */
idle_disk_energy = idle_disk_power * (now - last)
                    / max_clients;
for (i = 0; i < max_clients; i++)
   client[i].didle += idle_disk_energy;
```

### Propagation of Disk-Energy Records

Similar to propagation of processor-energy–accounting records, our prototype also provides two different protocols for propagating disk-energy–accounting records, a synchronous and an asynchronous variant. Again, we use a single mechanism for disk-energy accounting and allocation, and present the details of the mechanism later when detailing exposed disk-energy allocation (Section 4.4.2). In short, both the synchronous and asynchronous variant use a shared-memory segment to access energy accounting records. Whenever a request has been finished, the translation module records the access costs of the request into the segment, together with an identifier of the disk client. The idle-accounting procedure in the listing above also uses the segment to share idle-energy records with the energy manager. The difference between synchronous and asynchronous variants is that the synchronous one additionally lets the translation module send a notification message to the energy manager whenever a disk client issues requests to the client.

### 4.3.3 Recursive Energy Accounting

In the design chapter, we have argued that modular operating-system environments require energy management not only for raw hardware devices, but also for refined software resources (Section 3.3). Particularly in virtualization environments, we must enable both host-level and guest-level energy management, and cater for managing energy consumption of both physical hardware devices and of virtual software devices. Our framework therefore supports energy accounting not only for physical devices but also for *virtual devices*.

For that purpose, the framework is capable of attributing the energy consumption of virtual devices to originating guest applications running within a virtual machine. Conceptually, such energy accounting works similar to physical devices, where energy consumption is attributed to native L4 applications or to complete virtual machines. However, an important difference remains: while a physical device request consumes energy on the particular device only, fulfilling a virtual device request issued by a client may involve interacting with *several different* physical devices. With respect to energy accounting of virtual devices, it is therefore not sufficient to focus on single physical devices; rather, accounting must incorporate the energy spent recursively in the virtualization software layer or subsequent service layers and modules used to provide software resources.

#### Recursive, Request-Based Disk-Energy Accounting

We therefore perform a recursive, request-based accounting of the energy spent in the system. Since our framework currently supports accounting of processor and disk energy, the only case where recursive accounting is required occurs in the virtual disk driver located in the driver virtual machine. The cost for the virtualized disk consists of the energy consumed by the disk and the energy consumed by the processor while processing the requests. Hence, our disk driver also determines the processing energy for each request in addition to the disk energy as presented above. Like with accounting of physical disks, we again instrumented the translation module in the disk-driver virtual machine to determine the active and idle energy of the processor per client virtual machine.

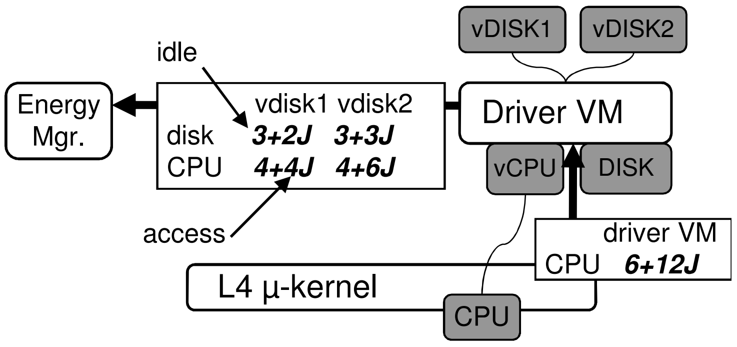

Figure 4.9: Recursive accounting of disk-energy consumption; for each client virtual machine and physical device, the driver reports idle and active energy to the energy manager. The disk driver is assumed to consume 8 J processor idle power, which is apportioned equally to the two clients.

The Linux disk driver combines requests to get better performance and delays part of the processing in work-queues and tasklets. When determining the active processor energy, it would be infeasible to track the processor-energy consumption of each individual request. Instead, we retrieve the processor-energy consumption at times and apportion it between the requests. Since the driver runs in a virtual machine, it relies on the energy-virtualization capabilities of our framework to retrieve a local view on its own processor-energy consumption (we present details on our implementation of energy virtualization in Section 4.5.2).

The Linux kernel constantly consumes a certain amount of energy, even if it does not handle disk requests. According to our energy model, we do not charge idle consumption with the request. To be able to distinguish the idle driver consumption from the access consumption, we approximate the idle consumption of the Linux kernel when no client virtual machine uses the disk.

To account active processor consumption, we assume constant values per request, and predict the energy consumption of future requests based on the past. Every 50th request, we retrieve the driver's processor-energy consumption and adjust the expected cost for the next 50 requests. The following code illustrates how we calculate the cost per request. In the code fragment, the static variable `unacc_cpu_energy` keeps track of the deviation between the consumed energy and the energy consumption already charged to the clients. The function `get_cpu_energy()` returns the energy (idle and active) consumed by the virtual processor since the last query:

```
/*
 * subtract idle processor consumption
 *  of driver virtual machine
 */
unacc_cpu_energy -= drv_idle_cpu_power * (now - last);

/*
 * calculate cost per request
 */
num_req = 50;
unacc_cpu_energy += get_cpu_energy();
unacc_cpu_energy -= cpu_req_energy * num_req;
cpu_req_energy = unacc_cpu_energy / num_req;
```

## 4.4 Exposed, Energy-Aware Resource Allocation

Like with accounting, our framework requires each driver to expose its resource allocation mechanisms to policy-manager subsystems possibly residing in different modules or protection domains. Policy managers leverage the allocation mechanisms to ensure that device-energy consumption matches the desired energy policies. In this section, we first explain how we implemented exposed processor-energy allocation by means of a user-controlled processor scheduling facility for L4; we then detail our device driver enhancements for exposed disk-energy allocation.

### 4.4.1 User-Controlled Processor Scheduling for L4

The time a particular activity spends on a processor directly correlates directly with the energy consumption and heat dissipation it causes there (although the correlation does not need to be linear, as we have explained in Section 4.2.1). Controlling processor scheduling is therefore paramount for any processor-energy policy management. In the following section, we will describe the *user-controlled* processor scheduling facility we have designed and implemented for the L4 microkernel. We will begin with describing the limitations of existing kernel-based schedulers and then motivate, how we enhanced flexibility and extensibility of processor allocation through user-controlled processor scheduling. Again, our prototype provides two different scheduling protocols, a synchronous and an asynchronous variant, which we will describe in given order. Note, that our approach focuses on the *software* side of the scheduling problem, since hardware aspects such as setting different frequency or voltage states can be easily exposed by giving access to the respective hardware registers, or by encapsulating the functionality in a simple interface. Once user-level policy managers have sufficient knowledge and control to define and dispatch different processor-scheduling domains — which our software approach, in turn, strives to provide —, they can freely use such hardware mechanisms in addition to the software control.

### Limitations of Kernel-Based Processor Scheduling

Traditionally, processor scheduling has been performed at kernel level, mostly for reasons of performance and complexity. Processor scheduling is widely regarded as too entangled with other operating system concepts — control flow, communication, accounting, interrupt handling, to name a few — to be easily removed without degrading efficiency. For that reason, virtually all microkernels that are of practical relevance employ a kernel scheduler, with the rationale that a microkernel must be efficient to be usable at all.

However, kernel-managed scheduling requires a kernel policy that allocates threads to processors. As long as user-level resource management conforms to that kernel policy, kernel scheduling is convenient, and enables development of concurrent programs or overlapping computations. However, as soon as user-level management claims freedom in scheduling, the kernel policy bars the way to the flexibility the microkernel ultimately strives to provide.

### Flexibility through User-Directed Processor Scheduling

In our framework, all energy-policy management resides outside the kernel. As a result, kernel-level scheduling with a specific standard scheduling policy is particularly burdensome. We therefore explore the design of a microkernel architecture that strives to expose *all* scheduling from the kernel to user level. The key idea of our approach is simple: whenever the microkernel encounters a situation that is ambiguous with respect to scheduling, we hand over the decision to user-level. For that purpose, we enhance all kernel operations with an additional interface that allows the kernel to resolve the ambiguity based on the user's decision. While the general idea of exporting kernel scheduling to the user is not new [Appavoo et al., 2002a, Tucker and Gupta, 1989, Anderson et al., 1991, Marsh et al., 1991, Leslie et al., 1996, Ford and Susarla, 1996], existing approaches are limited to single protection domains. In contrast, our approach relies on a generic scheme that is neutral to address spaces or other protection mechanisms.

User-directed scheduling exports the control over processor resource management to applications, allowing them to develop domain-specific policies (Figure 4.10). The advantage of user-level scheduling is obvious: It permits flexible definition of the systems' scheduling behavior. The potential drawback is evident as well: It increases kernel–user interaction and thus potentially reduces efficiency. Since both flexibility and performance are desirable properties for our framework, we explore two different points in the trade-off space, a synchronous and an asynchronous scheduling protocol. The synchronous, pure user-level scheduling variant, exports all scheduling decisions from the kernel to user-level schedulers. Such a solution provides the flexibility of adapting the scheduling policy to the target scenario without changing the microkernel; it allows, for instance, to exchange an energy-driven scheduler with a more throughput-oriented one for those environments where processor-energy management is not required. Also, it allows individual policies to be applied in different scheduling domains, enabling, for instance, guest operating systems to accurately pursue and direct their own internal, resource management. As the synchronous protocol induces substantial performance overhead, we additionally offer an asynchronous scheduling protocol, which retains a kernel-level scheduling policy for performance reasons, but still keeps user-level schedulers informed about the occurrence and history of kernel scheduling events via an asynchronous tracing mechanism.

To summarize, we are aware of potential performance problems of user-directed scheduling; nevertheless, we expect our work to be an insightful step towards the development of extensible and flexible processor-energy allocation schemes. In the following paragraphs, we first describe the scheduling-relevant aspects of the original L4 version. We afterwards present the changes we have made in order to provide synchronous and asynchronous processor scheduling.

### Original L4 Scheduling Behavior

In it original implementation, threads are scheduled by L4, according to an internal round-robin strategy, without user-level components taking notice. L4's kernel scheduler resembles traditional process schedulers of monolithic operating systems. All runnable threads are enqueued into a processor-local ready queue. Timer interrupts, blocking kernel operations, and user-initiated thread switches trigger the invocation of the L4 kernel scheduler, which chooses the next running thread based on the kernel policy. L4 provides an interface that allows user-level programs to adjust scheduling parameters. For that purpose, L4 arranges threads into a hierarchy, by associating a scheduler thread with each thread.

While user-level schedulers can tweak the kernel scheduling policy, core scheduling behavior such as the policy itself cannot be changed. As a result, a scheduler that requires a policy different to the default kernel policy needs to pursue complicated steps to implement it. The root cause is that there is no visible difference between the thread currently running and other runnable threads. For a user-level scheduler, all its subordinate threads appear to be running at the same time. To implement its own strategy, a scheduler must thus prevent L4 from selecting

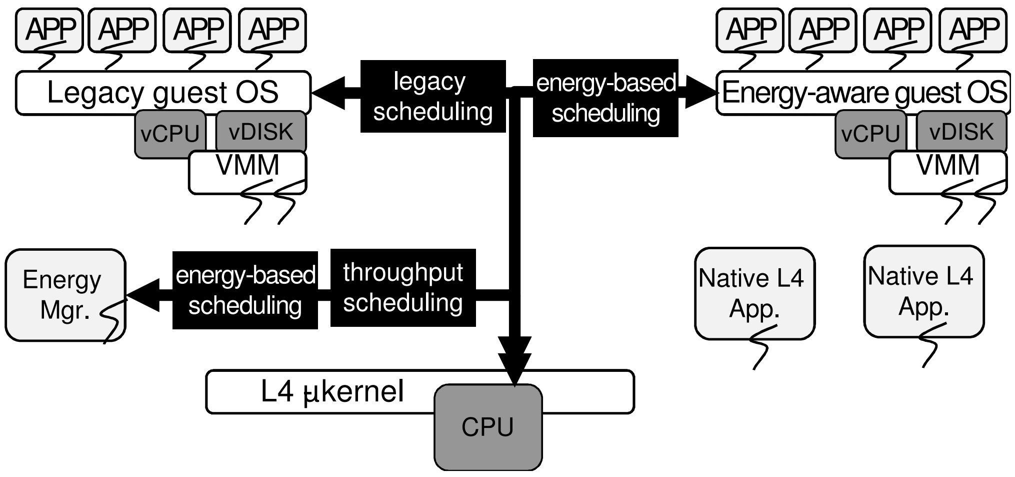

Figure 4.10: Flexibility through user-level scheduling. User-controlled scheduling allows different policies to be implemented in different sub-domains. At the host-level, the energy manager can implement an energy-aware or a more throughput-oriented scheme; at the same time, guest-level schedulers can implement both legacy scheduling and energy-aware scheduling, depending on their own capabilities; L4 does not implement a policy, but merely carries out user-specified scheduling decisions onto subordinate virtual machines respectively virtualized applications.

the "wrong" thread, by making sure that all threads except the one it itself designates to run are not runnable. Assuring that is awkward and inefficient; threads may change their own state when invoking kernel operations, and the scheduler requires a complex state machine to ensure correct states while on-going operations are handled gracefully.

### Synchronous User-Level Scheduling for L4

With the synchronous scheduling protocol variant, the basic idea of our new architecture is a very simple one: L4 does not perform any kernel scheduling anymore. It jettisons all scheduling-related context information such as time-slices, priorities, or run queues, and reduces thread semantics to the notion of execution (and communication) state. The timer interrupt becomes a "normal" interrupt again, treated like all other external interrupts; other timing semantics vanish from the kernel as well.

Lacking in-kernel scheduling, our new microkernel will execute the currently running thread (in its notion of execution context) unconditionally, until a blocking kernel event or operation (hence called preemption) occurs. To give user-level schedulers full control over the dispatching, L4 vectors out any thread preemption to an associated scheduler thread, by means of a preemption IPC. A preemption IPC is a coalesced send and receive operation to the scheduler, where the preempted sender automatically blocks waiting for a reply message. To re-grant processor time, the scheduler thread responds by sending back an IPC reply message, which again results in the destination thread running until the next preemption occurs (see Figure4.11). Note, that the notion of preemption IPC already exists in the original L4 kernel, but with different semantics related to time-quanta [L4 Development Team, 2009a]. To designate schedulers, the kernel keeps the scheduler hierarchy concept as it is in the original L4 version, and associates a user-configurable scheduler thread with each thread.

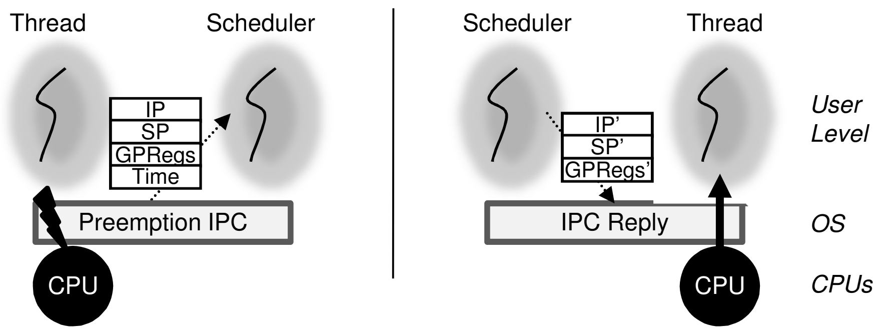

Figure 4.11: Synchronous scheduling control via preemption IPCs. A preemption IPC is a coalesced send and receive operation to the scheduler, where the preempted sender automatically blocks waiting for a reply message. To re-grant processor time, the scheduler responds by sending back a reply message, which again results in the destination thread running until the next preemption occurs.

For reasons of convenience, a preemption IPC contains the current thread's execution state such as instruction and stack pointer, and general-purpose registers. It also contains information of all other threads involved in the particular scheduling event. A reply may contain updated state, which is then transparently installed into the thread's user frame. Transferring the execution state allows the schedulers to inspect thread execution context; updating the state with the reply eases typical scheduler operations such as sending signals or switching to a different user-level thread. Note that, to facilitate processor-energy accounting, the message does not have to contain any performance counter records; being notified about preemptions instantly, the user-level schedulers and energy managers can obtain the performance-counter snapshots themselves.

### Synchronous Scheduling on Kernel Operations

In the original L4 version, kernel operations involving threads often lead to scheduling events that are processed and handled by the in-kernel scheduler. Our synchronous version changes the semantics of all those events, as we will detail in the following. Fortunately the number of such operations is very limited; besides IPC, there are only three other kernel operations. For the sake of completeness and importance, we also present the new semantics of the exception and interrupt handling, although all changes to their handling are a direct consequence of the changes in IPC semantics.

**IPC** Specifically, the transfer of an IPC always requires a scheduling decision, no matter if it hands-off the processor from the sender to the destination, or if it leaves more threads than it blocks. The original version resorts to a simple time-slice donation shortcut policy in case of a hand-off, circumventing even the kernel-level scheduler [Elphinstone et al., 2007]. We instead treat IPC according to our explicit model: the kernel *always interposes* the IPC path, sending a preemption IPC to the associated scheduler. A special case then arises with preemption IPC messages themselves, which are sent as a result of a preemption or activation: for preemption traffic, the kernel directly transfers control from the sender to the receiver. This behavior is safe, since a preemption IPC always blocks the sender waiting for re-activation by the scheduler, and since no-one else except the receiving scheduler is runnable.

**Switch to Idle** As most other kernels, L4 originally features an in-kernel, per-processor idle thread that is invoked when no other thread is runnable, and runs until the next external interrupt occurs. Our modified L4 version does not contain any notion of idle thread anymore (with the minor exception of some remaining idle functionality for compatibility at boot time). To allow switching to low-power modes, L4 permits the top-level scheduler to execute special idle instructions such as `hlt` on IA-32 based processors.

**Thread Switch** L4 originally offers a `ThreadSwitch` system call that donates the current time slice from the caller to the callee; if no destination is specified, the in-kernel scheduler selects the next thread to run. For our new architecture, `ThreadSwitch` is a bogus operation, since only the current thread is runnable; for reasons of compatibility `ThreadSwitch` is still available, but now sends a preemption IPC to the invoker's scheduler.

**Exchange Registers** The `ExchangeRegisters` system call allows a thread to read or modify parts of the execution and communication state of another thread, provided both threads are executing within the same address space. To that end, it also allows the invoker to suspend or resume other threads. In analogy to IPC, our new kernel therefore preempts the invoker, in case multiple threads are involved in the operation.

**Interrupts and Exceptions** The implications of user-level scheduling on interrupts all arise from the novel semantics of hierarchical scheduling dealt with in the IPC path. On exceptions, L4 delivers a blocking IPC to the associated exception handler, with the same preemption logic being involved as in regular IPCs. Likewise, whenever a thread is preempted by an external interrupt, L4 sends a preemption message to the designated scheduler, which is, in this case, the lowest common scheduler of the current thread and the interrupt handler thread. Again, for reasons of performance, L4 piggybacks the interrupt message to the preemption message, resulting in just a single IPC being sent on each interrupt.

### Synchronous Hierarchical Scheduling

Our re-designed synchronous scheduling behavior results in a hierarchical architecture that gives full control over the dispatching of each thread to parent scheduler threads. The hierarchy also defines how situations are resolved when multiple threads become activated or preempted at the same time. In such cases, the kernel actually sends a single preemption IPC only, but chooses as destination the *lowest common* top-level scheduler of all involved threads (Figure 4.12). The kernel further allows top-level schedulers to respond to such forwarded preemption IPCs *on behalf* of the actual, lower-level schedulers; in other words, top-level schedulers may send re-activation messages that *pretend* to be coming from the low-level schedulers. The kernel therefore modifies the sender identifier appropriately when executing the IPC path.

This scheme may seem complex at a first glance. In practice, however, it actually enables a straight-forward scheduling process at user level, which resembles stack-based exception handling of processors, with the extension that exceptions are routed in hierarchical fashion: that is, schedulers recursively donate processor time to a destination thread, waiting for a preemption IPC from any thread within the hierarchy subordinate to that destination. Consider, as illustration, the example depicted by Figure 4.12, where a thread sends an IPC message to a second thread, which leaves both send and receive partner runnable. In that case the kernel determines the top-level scheduler of both partners, and sends a preemption IPC containing notice of both activations. The top-level scheduler may then decide to either activate one of the two partners, using an re-activation IPC deceiving to be coming from their direct scheduler. Alternatively, it may decide to leave both partners preempted and schedule a completely unrelated thread instead (e.g., because that thread has higher priority).

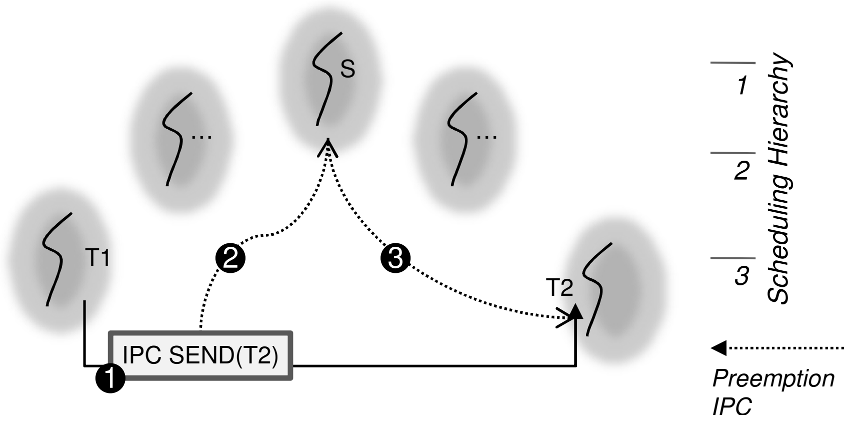

Figure 4.12: Recursive Scheduler Hierarchy. While sending an IPC to T2 (1), T1 is preempted; the kernel sends a preemption IPC on behalf of T1, to the lowest common scheduler S of T1 and T2 (2). S is free to choose the next running thread; in the example, it puts through the IPC message by directly activating T2 (3).

Hierarchical scheduling models are a well-known approach to provide scheduler extensibility [Ford and Susarla, 1996,Goyal et al., 1996,Jones et al., 1997]. In our architecture, the hierarchy serves similar purposes: it transfers local scheduling decisions to local agents; and it transferring decisions involving multiple domains to a top-level agent. Hierarchical redirection schemes have also been proposed as a means to permit access-control in microkernel-based system [Jaeger et al., 1999, Liedtke, 1992]. Our scheme leverages ideas from those approaches, but for the different purpose of controlling processor scheduling rather than communication security.

## Asynchronous Scheduling Control via Event Tracing

The synchronous, preemption IPC based protocol propagates *all* scheduling decisions to the user-level; it thus permits fine-grain isolation of processor time and energy between threads. However, as the preemption message transfer requires additional transitioning between the kernel and user-level, synchronous scheduling incurs a performance overhead. Specifically, since some scheduling events occur during the kernel IPC path — the sender blocks waiting for the unblocked, activated receiver —, synchronous scheduling inherently penalizes threads that frequently communicate with each other.

According to our design credo of adaptive energy-management interaction protocols, our kernel offers a second, more relaxed protocol that uses event tracing and policy shortcuts to optimize scheduling control. Whenever a processor-scheduling policy does *not* require strict and timely notification of kernel scheduling events, L4 can employ this alternative protocol to resolve scheduling decisions; it comprises of two parts: and the recording of the scheduling event into an in-memory buffer shared with the user-level scheduler, and the application of a pre-defined shortcut policy. In our current architecture, we use a shortcut policy that implements a time-based lottery scheduling policy [Waldspurger and Weihl, 1994, Uhlig et al., 2004].

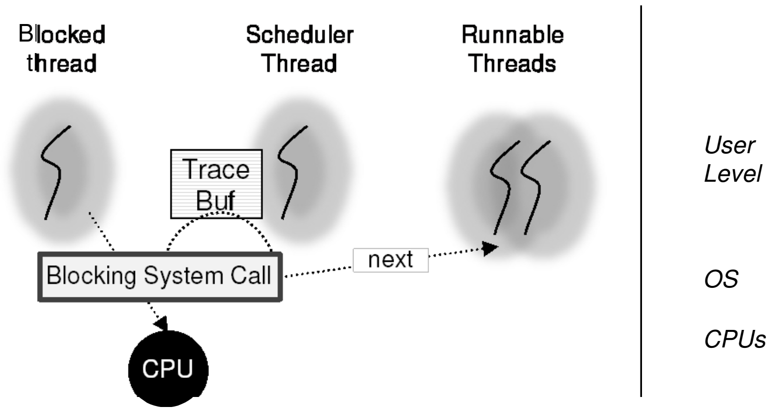

Figure 4.13: Relaxed, Asynchronous Scheduling Control via Event Tracing. L4 records the occurrence of thread context switches in a scheduler-accessible memory buffer, and resolves the pending decision internally, using a shortcut scheduling policy.

In the asynchronous model, effectively all cases that block the current thread and activate one destination — including IPC — become a shortcut scheduling decision. The alloted strides determine which thread to grant the processor next. In contrast to the synchronous protocol, the asynchronous protocol does require the kernel to take performance-counter snapshots for energy accounting. The kernel therefore piggybacks performance-counter snapshots onto the traces generated on thread scheduling events. The energy manager can later on (e.g., at a timer interrupt) analyze the buffer and reproduce the complete energy trace of its subordinate threads, and use that to readjust processor allocation. We will present details on energy policy management in Section 4.6.

The goal behind using scheduling shortcuts is to improve performance in presence of frequent scheduling events. While this resembles the original L4 scheduling behavior with an in-kernel round-robin policy, there are still two important differences: First, to still keep associated user-level schedulers informed about occurrence of scheduling events, the kernel also records the occurrence of the event into a shared-memory buffer, which is mapped into the address space of the scheduler. Second, to allow user-level schedulers to perform processor allocation on thread groups such as virtual machines, the kernel implements the shortcut policy in hierarchical fashion: that is, each domain forms a basic processor-scheduling group comprising all threads with that domain identifier. Strides can be set for thread containers as well as for individual threads within the container; the kernel scheduler first selects the particular thread group to be scheduled based on the top-level tickets; in a second step, it then selects the thread within that group based on the second-level tickets.

Note that, at present, our asynchronous protocol offers a single, global scheduling policy; a natural improvement to that scheme would be to offer different replaceable kernel scheduling policies, potentially even with individual policies for different subsystems in the hierarchy. While we believe that such an approach may be promising in general, we are also aware of its potential problems: to implement such policy replacement in practice, on can either offer a fixed set of scheduling policies to user-level schedulers (as done in the K42 operating system, for example [Appavoo et al., 2002a]); or, alternatively, one can provide means for user-defined specialization of the kernel scheduler (as done in the Exokernel, for example, using a safe language [Bershad et al., 1995]). The former approach limits extensibility, however, and still requires user-level schedulers to agree with the kernel on a pre-defined policy; the latter approach typically requires complex and sophisticated safety-checks [Druschel et al., 1997].

### 4.4.2 Exposed Energy-Aware Disk Control

In this section we detail how we implemented an exposed allocation mechanism regulating disk energy of individual clients. With respect to the problem of allocation, disk management is far less complex than the processor scheduling: Processor work units usually come as "threads", which have non-uniform and non-fixed demands that may be preempted before completion. Also, threads can interact with each other through communication mechanisms. In contrast, disk devices are request based, with each given request entailing a work unit whose amount of work (or size) is known in advance and does not change. Furthermore, disk requests are independent of each other on the software side (although individual requests and their sequence of dispatching may influence the disk performance or energy consumption), and individual requests are not preemptable once they have been scheduled to the hardware device. Substantial differences also exist in the timeliness requirements by hardware power management features. Transitioning between different processor power states has become feasible even on the base of individual threads [Flautner and Mudge, 2002, Weissel and Bellosa, 2002]. In contrast, transitioning a disk between different active and idle states or between different rotational speeds takes a significant amount of time [Gurumurthi et al., 2003] and has a significant impact on disk reliability [Zhu et al., 2005].

For those reasons, our approach again focuses on the *software* side of the disk-energy allocation. Since transitioning between states is expensive, the delay times between transitions are rather high for standard disk-energy–management policies (up to of a few seconds); as a result, hardware mechanisms such as setting sleep states can be exposed in rather straight-forward manner, without any performance implications. We find software-based mechanisms to be more suitable as vehicle for exploring energy-management–protocol issues and efficiency problems than hardware disk states.

Our current prototype implements an energy allocation scheme based on the idea of *power capping*: the scheme allows external disk-energy–policy managers to cap the active disk power consumption of individual client virtual machines to a policy-specific budget. Although a rather simple scheme, power capping has proven to be a powerful tool, which can serve two different purposes: First setting a global, computer-wide energy budget can supplement hardware cooling facilities, as they avoid power and thermal peaks that otherwise result in hardware failures; power capping thus allows hardware and facility designers to use cheaper hardware and cooling if the operating system can ensure a give power and thermal limit can not be surpassed [Raghavendra et al., 2008, Govidan et al., 2009, Rohou and Smith, 1999, Ainsworth et al., 2008, Hewlett-Packard Development Company, 2008]. Second, setting application-specific energy budgets allows energy isolation between individual application domains. With our budgeting algorithm, policy managers can stipulate individual disk power budgets, and the management infrastructure enforces enforces them among all guest operating systems, native applications, and service components.

Note that, in our infrastructure, reducing the disk request rate not only reduce the direct (access-energy) consumption of the disk; it also reduces the recursive processor-energy consumption the disk driver requires to process, recompute, and issue requests. Our implementation of the capping algorithm is simple: the disk driver processes a client virtual machine's disk requests only to a specific request budget, and delays all pending requests. Again, our prototype offers a synchronous and an asynchronous mechanism for disk-energy management. Depending on the mechanism used, the capping logic resides in the energy manager or in the driver itself, as we will describe in the following.

### Synchronous Propagation of Disk-Allocation Decisions

The synchronous disk-energy management variant notifies the energy manager whenever a disk client issues a request to the disk driver; since the disk driver implements request batching, the actual notification may comprise more than a single request. The driver waits for a response from the energy manager before proceeding (note that, at present, our driver is serialized at this point; however, we could easily add multiple threads in the driver to provide concurrency among multiple clients). Being notified about the pending requests, the energy manager then retrieves accounting records and other request data from the memory segment shared with the driver. To delay pending requests, the energy manager modifies the request data accordingly; eventually, it dispatches a response to the waiting driver. To implement notifications, data sharing, and agreement among data types data structures, we leverage a high-level interface definition language (IDL) that translates object interfaces into low-level L4 IPCs [Haeberlen et al., 2000]. The following code snippet illustrates the synchronous scheme:

```
/*
 * IO processing in the disk driver
 */
void process_io(client_t *client)
{
   ring = &client->ring;

   /* Notify manager and wait */
   IDL_disk_request_client_stub(manager_tid, client);

   for (i=0; i < client->requests; i++)
   {
      desc = &client->desc[ ring->start ];
      ring->start = (ring->start+1) % ring->cnt;
      initiate_io(conn, desc, ring);
   }
}

/*
 * Notification stub in the energy manager
 */
void IDL_disk_request_server_stub(client_t *client)
{
   /*
    * Client request data is shared between
    * manager and driver
    */
    client->requests = apply_disk_energy_capping(client);

   /* IDL stub will respond with IPC */
}
```

### Asynchronous Propagation of Disk-Allocation Decisions

With the asynchronous variant, disk drivers and policy managers communicate disk-energy–management events via shared memory only. As a result, the capping policy itself must reside in the driver, as a shortcut policy. The mechanics of asynchronous disk throttling stay in analogy to asynchronous processor scheduling, where a shortcut processor policy resides in the kernel as well. To put disk-energy caps in effect, the energy manager changes corresponding client throttle factors asynchronously, by modifying the values in the shared memory segment appropriately; the following code snipped illustrates the asynchronous algorithm:

```
/*
 * IO processing in the disk driver
 */
void process_io(client_t *client)
{
   ring = &client->ring;

   for (i=0; i < client->budget; i++)
   {
      desc = &client->desc[ ring->start ];
      ring->start = (ring->start+1) % ring->cnt;
      initiate_io(conn, desc, ring);
   }
}

/*
 * Update budget in the disk driver
 */
void update_bio_budget()
{
    for( idx = 0; idx < MAX_CLIENTS; idx++ )
    {
        /*
         * Retrieve budgets from shared data structure
         */
        client[idx]->budget =
               get_throttle(client->client[idx]);
        process_client_io( &client );
    }
    update_bio_budget_timer.expires =
               jiffies + UPDATE_PERIOD;
    add_timer(&update_bio_budget_timer);
}
```

## 4.5 Legacy Compatibility

To enable running applications written for existing operating systems, our framework allows the execution of a para-virtualized Linux operating system on top of L4. Virtualized operating systems pursue their own internal, resource management, and we consider it to be a key criterion to fully preserve their semantics as far as possible. On the other hand, our framework also allows for incremental improvements of such legacy operating systems. The following paragraphs are devoted to a discussion on that matter. We will first discuss how our framework enables faithful emulation of user-directed processor and disk-resource–management decisions. We will then detail the enhancements we have added to allow exchanging or adapting guest-internal policies — which are performance oriented by tradition — with more energy-efficient policies.

### 4.5.1 Legacy Resource Management

A key goal of our approach is to preserve enough compatibility that the overall system can run existing applications. Like their native counterparts, virtualized operating systems pursue their own, internal resource management: that is, the guest operating systems manage and multiplex access to their virtual devices from their subordinate applications. It is a primary goal of faithful virtualization to fully preserve guest-internal resource-management semantics as far as possible. It is also worth noting, however, that the microkernel-based virtualization approach, which we use in our prototype, is far more at risk to break guest semantics than the virtualization-only approach, as done in Xen [Barham et al., 2003] or VMware [VMware Inc., 2009a]. The main reason for the challenge lies in the microkernel API: microkernel systems usually offer basic abstractions such as threads, address spaces, or IPC [Liedtke, 1995, Herder et al., 2006], and leave it up to user-level programs to provide a virtual platform. In contrast, virtualization-only approaches typically provide virtual devices as first class abstractions (e.g., virtual processors, virtual physical memory, virtual network interfaces). Naturally, it is easier to emulate a processor or a memory-management unit faithfully, if the virtual processor or virtual memory-management unit is a *first-class* abstraction provided by the kernel.

We envisage a system that preserves the advantageous system structure of microkernel-based systems, allowing, for example, the development of a virtual-machine monitor or other services as a de-privileged user-level component; at the same time, however, the system should provide the same level of faithful virtualization as known from pure virtualization systems; it was therefore a first goal to accurately emulate guest-level resource management in our prototype, in the following, we will describe how we attain legacy compatibility both with disk and with processor resource management.

### Legacy Disk Scheduling

Preserving legacy disk semantics is a simple and straight-forward task, and does not require any additional means beyond establishing a virtual disk architecture. The simplicity stems from two reasons: first, our current prototype supports access to storage at the (coarse) granularity of virtual disks subsystem only. Second, disk requests are typically independent from each other, with no interaction occuring in between them once they have been issued to the virtual disk driver.

### Legacy Processor Scheduling

With regards to legacy processor semantics, our faithful scheduling architecture plays a key role, as it enables effective, hierarchical, and accurate user-directed scheduling. Our approach enables scheduling that is fully directed by the user-level energy manager and the kernels of guest operating systems. A three-level scheduling hierarchy (see Figure 4.14) allows the user-level virtual-machine monitor to schedule its subordinate virtual machines according to a host-level energy policy (by means, e.g., of an energy-proportional scheduler). Scheduling at the guest level, however, is completely up to the guest operating system: our recursive approach allows the directives of guest schedulers to be faithfully emulated and mapped to the L4 kernel thread interface. As a result, our scheme allows local components to faithfully emulate their own scheduling policies; at the host-level, the energy manager can choose to implement different policies for processor-energy management, with varying thermal or power-centric goals; guest-level schedulers, however, can still rely on the infrastructure to ensure that their own application-specific policies are carried out on their own subordinate applications as well.

## 4.5.2 Support for Energy-Aware Improvements

Beyond preserving original legacy semantics at guest level, our framework also allows existing guest–operating-system code to be “improved” for better support for energy management. Our approach is driven by the idea of para-virtualization, where guest operating systems are slightly modified to run on virtual hardware that is not fully compatible to existing physical hardware. Similarly, we also perform small modifications of existing resource-management guest code in order to enable the guests to perform application-specific management of processor and disk energy . Our framework therefore supports accounting and control not only for physical but also of virtual devices.

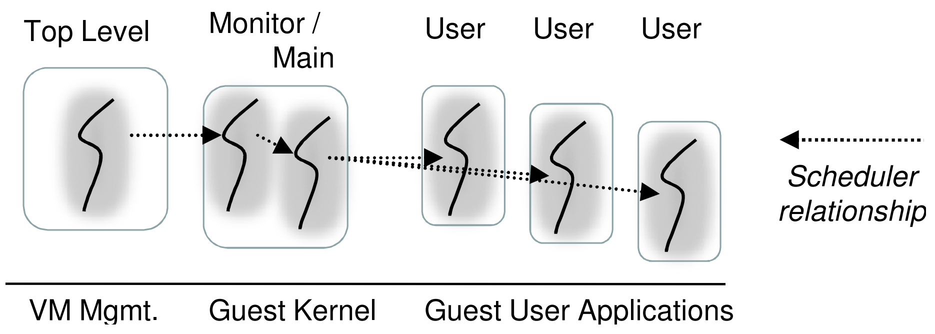

Figure 4.14: 3-Tier scheduling architecture supporting legacy guest–operating-system scheduling. At the lowest level, a root-scheduler thread located in an the external resource monitor allocates processor time to complete virtual machines. At the second level, two threads running within the guest address space redistribute their time allotment among the guest kernel and subordinate applications.

As already explained, the main difference between a virtual device and other software services and abstractions lies in its hardware-like interface: a virtual device closely resembles its physical counterpart. However, most current devices offer no hardware interface to query energy or power consumption, and the most common approach to determine the energy consumption is to estimate it based on certain device characteristics, which are assumed to correlate with the energy consumption of the device. By emulating the according behavior for the virtual devices, we support energy estimation in the guest without major modifications to the guest's energy accounting. Our ultimate goal is to enable the guest to use the same driver for virtual and for real hardware. In the remainder of this section, we describe how we support energy accounting of virtual processors and disks.

### Virtual Processor-Energy Accounting

For virtualization of physical energy effects of the processor, we provide, to each virtual machine, a set of virtual performance-counter registers, which give guest operating systems a private view of their current energy consumption. The virtual model relies on the recording of performance counters at times of thread preemptions, which we presented in Section 4.3.1.

Like their physical counterparts, each virtual processor has a set of virtual performance counters, which factor out the events of other, simultaneously running virtual machines. If a guest operating system determines the current value of a virtual performance counter, an emulation routine in the in-place monitor obtains the current hardware performance counter and subtracts all advances of the performance counters that occurred when other virtual machines were run-

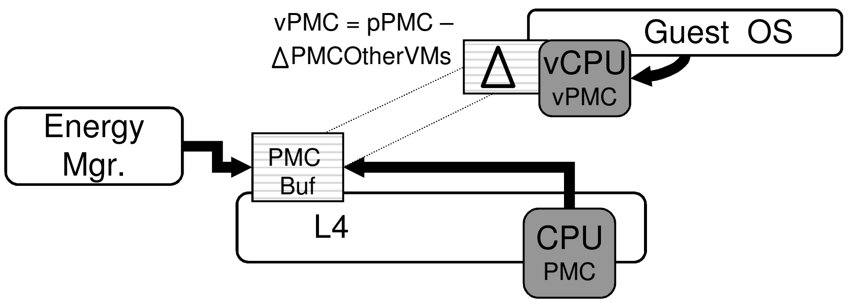

Figure 4.15: Virtualizing performance counters via per–virtual-machine records of the physical counter advances recorded during execution of the virtual machine.

ning (Figure 4.15). The hardware performance counters are made read-accessible to the in-place virtual-machine monitor by setting a control register flag in the physical processors (the guest operating system itself is not granted direct read access; the in-place monitor intercepts and emulates all corresponding instructions on its behalf). The advances of other virtual machines are derived from the performance-counter records collected in the energy manager. To be accessible by the in-place virtual machine monitor, the energy manager maps the corresponding data structures read-only into the address space of the guest operating system.

### Virtual Disk-Energy Accounting

In contrast to the processor, our disk-energy–estimation schemes does not rely on on-line measurements of sensors or counters; rather, it is based on known parameters such as the disk's power consumption in idle and active mode and the time it remains in active mode to handle a request. Directly translating the energy consumption of physical devices from our run-time energy model to the parameter-based model of the respective guest operating system would yield only inaccurate results. The virtual machine would have to calibrate the energy consumption of the devices to calculate the energy parameters of the virtual devices. Furthermore, parameters of shared devices may change with the number of virtual machines, which contradicts the original estimation model. To ensure accuracy in the long run, the guest would have to query the virtual devices regularly for updated parameters. For our current virtual disk-energy model, we therefore use a para-virtual device extension. We expose each disk-energy meter as an extension of the virtual disk device; energy-aware guest operating systems can take advantage of them by customizing the standard device driver appropriately.

## 4.6 Energy-Policy Management

Our framework presently implements energy policies at two different layers in the system: a global host-level energy manager implements processor and disk-energy allocation policies for whole virtual machines and for native L4 applications. An energy-aware guest operating system is responsible for for energy management at the level of individual virtualized applications, within virtual machines. In the following, we explain how we implemented our management polices in the respective components; we begin by detailing the inner workings of the host-level energy manager; we then present our energy-aware guest operating system that redistributes its own virtual machine-wide allotments to its own, subordinate applications.

### 4.6.1 Host-Level Energy Management

Our host-level energy manager currently implements an energy-isolation policy, which enforces device-power budgets for virtual machines and native applications. Energy isolation comprises both processor and disk devices, and relies on the accounting and allocation mechanisms of the microkernel and driver virtual machine described previously. The energy manager is co-located with the external monitor module, which is responsible for global, host-level resource management. As a result, it is also co-located with the top-level scheduler responsible for dispatching virtual machines and native applications. The actual management policy consists of an initialization procedure and a subsequent policy algorithm. During initialization, the manager determines a power limit for each virtual machine or native application and device type, which may not be exceeded during runtime. The processor-power limit reflects the active processor power a virtual machine is allowed to consume directly. The disk-power limit reflects the overall active power consumption the disk-driver virtual machine is allowed to spend in servicing a particular client, including the processor energy spent for processing requests. Note that, nevertheless, the driver's processor and disk energy are accounted separately. Finding an optimal policy for allotment of power budgets was not the focus of our work; at present, the limits are set to static values.

Note that, our work presently focuses on the mechanisms of energy management rather than on policies; as a result, both our processor and our disk-energy management policies are rudimentary, and mainly serve the purpose of validating the principal operativeness of our mechanisms. Future work has to be done to explore the large body of existing algorithms and policies for operating-system energy management, and its applicability to modular operating systems.

### Processor-Energy Management

For processor-energy management, we implemented an energy-based lottery-scheduling policy [Uhlig et al., 2004, Waldspurger and Weihl, 1994] that allots proportional processor-energy shares to virtual processors and applications. Unlike the original algorithm, where the lottery tickets represent the permission to access the processor for a given amount of *time*, the tickets in our algorithm permit access to the processor for a given amount of *energy*. Effectively, the policy algorithm dynamically throttles the energy consumption of virtual processors and L4 applications by computing the energy-ticket allotments accordingly.

A key feature of lottery scheduling is that it does not impose fixed upper bounds on processor utilization: the shares have only relative meaning, and if one virtual processor or application does not fully utilize its share, the scheduler allows other, competing ones to steal the unused remainder. An obvious consequence of dynamic upper bounds is that energy consumption will not be constrained either, at least not with a straight-forward implementation of lottery scheduling.

We solve this problem by creating a distinct *idle application* per physical processor, which is guaranteed to spend all alloted time with issuing halt instructions. With synchronous processor scheduling, the idle application is merely a matter of executing the `hlt` instruction within the root scheduler. With asynchronous scheduling, we create a special idle L4 thread that is assigned its own processor time allotment and executes the `hlt` in a tight loop. The x86 hardware raises an exception whenever user-level software executes the `hlt` instruction transitioning the processor into halted mode. For privileged threads such as the root scheduler or the idle application, L4 translates the exception directly into the real `hlt` afterwards. Initially, no processor share is alloted to the idle application, thus all other virtual processors and applications will be favored on the processor if they require it, yet isolated from each other with different energy allotments.

To cap energy consumption globally, the energy manager can increase the processor-energy shares of the idle application — in other words, execute `hlt` more often in the synchronous scheduling case, or increase the processor time allotment of the idle L4 thread in the asynchronous scheduling case —, which implicitly decreases the shares of real virtual processors and applications. The idle application, in turn, directly translates the remaining processor time into halt cycles. Our approach guarantees that energy limits are effectively imposed; but it still preserves the advantageous processor stealing behavior for all other virtual processors. Since the policy is kept out of the kernel, it can easily be improved with little effort; it can even be exchanged by a more throughput-oriented one for those environments where processor-energy management is not required. Note that our approach of using an idle application is conceptually separate from our general energy-accounting scheme described in Section 4.4: the amount of idle energy (which, according to our estimation model, results from the advancement of the time-stamp counter during `hlt`) is apportioned among the actual virtual machines and L4 applications running on the processor.

Our infrastructure for processor-energy management offers two different methods for accounting and allocation: synchronous and asynchronous. We therefore provide two different host-level policy implementations as well, which we detail in the following:

**Synchronous Processor-Energy Management** With the synchronous accounting and allocation protocol, the policy algorithm is invoked whenever the top-level processor scheduler selects the next virtual processor or native L4 application to be granted the physical processor; this selection process takes place regularly whenever the timer interrupt fires, and on occasion, when the presently running application or virtual processor is blocked, yields the processor voluntarily, or an interrupt occurs. The kernel preemption logic vectors all those preemptions to the top-level scheduler co-located with the energy manager. The policy algorithm uses the energy-accounting records to determine the actual energy consumption per application and virtual machine; it selects the next virtual machine or application to run by executing a lottery according to the definition of lottery scheduling algorithm [Waldspurger and Weihl, 1994]. With the synchronous protocol, the user-level scheduler effectively runs the policy itself; it carries it through by means of re-dispatching the threads selected to run. The user-level scheduler algorithm looks as follows:

```
/*
 * Calculate active processor power:
 * paccess : last access-energy consumption
 * stsc    : cycles passed when client was running
 */
client[cur_id].appower = client[cur_id].paccess /
                         client[cur_id].stsc;
client[cur_id].paccess = 0;
client[cur_id].stsc = 0;

word_t esum = 0;

for (word_t id = 0; id < num_clients; i++)
{

  if (client[id].state != vm_state_runnable)
    continue;
    /*
     * Calculate the energy ticket:
     *   ticket: the client CPU allotement
     */
    client[id].eticket =
       client[id].ticket / client[id].appower;
    esum += client[id].eticket;
}

if (esum)
{
  word_t lottery = rand() % esum;
  word_t winner = 0;

  for (word_t id = 0; i < num_clients; i++)
  {
    if (client[id].state != vm_state_runnable)
      continue;
    winner += client[id].eticket;
    if (lottery < winner)
      return client[id];
  }
}
return NO_RUNNABLE_CLIENT;
```

To calculate the energy-based lottery ticket (`eticket`) for a given client virtual machine or application, we divide the original client allotment (`ticket`) by the client's active power. When determining the active power, we currently consider the last execution periods only; an alternative approach would be to use exponential averages of multiple past execution periods. Note, that we base the calculation of lottery tickets on the *active power consumption*, that is, on the energy consumption of a client divided by the cycles is has consumed. As a result, the scheduler disregards the the overall energy consumption of the client, which may be low despite high active power, if the client does not use the processor very often. An alternative policy taking actual energy into account, is to divide the energy by the total cycles rather than the cycles of the client. Since the scheduler algorithm runs outside the kernel, we can easily pursue changes. For evaluation, however, we use the active power policy, since it better demonstrates the advantages of extensible processor-energy allocation.

**Asynchronous Processor-Energy Management** With the asynchronous protocol, implementation of processor allocation is split among the kernel — where shortcut policies still reside — and the user-level policy manager. Here, the L4 kernel implements a time-based lottery-scheduling policy itself, alloting proportional time shares to threads; however, the kernel also provides an interface that allows (privileged) user-level programs to adjust the lottery tickets. The host-level energy manager leverages the interface and maps an energy-lottery scheduling policy on to the kernel scheduler by adjusting the alloted time tickets dynamically, in a feedback loop. The feedback loop is invoked periodically, every 100 ms. It first obtains the processor-energy consumption of the past interval by querying the accounting records provided by the kernel at times of preemptions. It then compares the actual power consumption with the desired power limits multiplied with the time between subsequent loop invocations. If they do not match for a given virtual machine or application, the manager aligns them by recomputing and setting the kernel-level (time-based) tickets through a kernel interface.

```
for (word_t i = 0; i < num_clients; i++)
{
  /*
   * Calculate client active power from L4 traces
   * paccess: access-energy consumption
   * stsc   : cycles burned
   */
  client[id].paccess = L4_PMC_GetAccessEnergy();
  client[id].stc     = L4_PMC_GetTSC();
  client[id].appower = client[id].paccess /
                       client[cur_id].stsc;

  /*
   * Gracefully approximate L4 ticket based on budget
   */
  u64_t appower    =  client[id].appower;
  u64_t apbudget   =  client[id].apbudget;

  if (appower > apbudget)
    client[id].eticket -=20;
  else
    client[id].eticket +=20;

  L4_SetTicket(client[id], client[id].eticket);

}
```

### Disk-Energy Management

For disk-energy management, we have implemented a feedback loop to recompute throttle factors every 200 ms. It first obtains the disk energy consumption of the past interval by querying the accounting infrastructure. The current consumptions are used to predict future consumptions. For each disk, the loop compares the current energy consumption of each virtual disk client with the desired power limit multiplied by the time between subsequent invocations. If they do not match for a given virtual disk, the manager regulates the device consumption by recomputing and propagating disk throttle factors to the driver virtual machines. If the synchronous interaction protocol is used, the energy manager uses the throttle factors to throttle client disk requests by itself, whenever requests are propagated from the driver. If the asynchronous interaction protocol is used, the manager passes the throttle factors to disk-driver subsystem, which internally uses the factors to compute the request allotments per guest. When computing the disk-throttle factor, the manager takes the past period into consideration, and calculates an offset $\Delta t$ that is added to the current factor to retrieve the new throttle factor. The calculation occurs according to the following formula:

$$\Delta t = \begin{cases} \frac{1}{dta}(t_l - t) + \frac{e_l - e_c}{|e_l - e_c|} & : \begin{cases} e_c > e_l, t > t_l \\ e_c < e_l, t < t_l \end{cases} \\ \frac{dta}{dta-1}(t - t_l) + \frac{e_l - e_c}{|e_l - e_c|} & : \quad else \end{cases}$$

In this formula, $e_c$ denotes the energy consumed, $e_l$ the energy limit per period, and $t$ and $t_l$ and denote the present and past disk throttle factors; finally, $dta$ denotes an adjustment factor determining how much the past difference in throttle factors ($|t_l - t|$) should contribute to the new factor. In practice, we used an adjustment factor of 4 (i.e., 25 per cent). Resulting throttle factors effectively range from 0 to a few thousand:

```
#define DTA   4
static u64_t last = 0;
u64_t tdelta = now - last;

for (word_t i = 0; i < num_dclients; i++)
{

  /*
   * Calculate disk client active disk power:
   *  daccess: access-energy consumption
   *  tdelta : time passed since last invocation
   */
   client[id].daccess = DiskDD_AEnergy();
   dclient[id].adpower =
       (dclient[id].daccess / tdelta);

   client[id].daccess = 0;

    /*
     * Calculate disk throttle factor
     *   dthrottle: current throttle factor
     *  odthrottle: old throttle factor
     */

    u64_t adpower    =  dclient[id].adpower;
    u64_t adbudget   =  dclient[id].adbudget;
    u64_t dthrottle  =  dclient[id].dthrottle;
    u64_t odthrottle =  dclient[id].odthrottle;

    if (adpower > adbudget)
    {
      if (dthrottle >= odthrottle)
        dthrottle -=
          ((dthrottle - odthrottle) / DTA);
      else
        dthrottle -=
          (DTA * (odthrottle - dthrottle) / (DTA-1));
    }
    else
    {
      if (dthrottle <= odthrottle)
        dthrottle +=
          ((odthrottle - dthrottle) / DTA);
      else
        dthrottle +=
          (DTA * (dthrottle - odthrottle) / (DTA-1));
    }

    dclient[id].odthrottle = dclient[id].dthrottle;
    dclient[id].dthrottle = dthrottle;

    DiskDD_SetThrottle(dclient[id], client[id].dthrottle);

}
```

### 4.6.2 Guest-Level Energy Management – An Energy-Aware Guest Operating System

For guest-application specific energy management, we have incorporated the familiar resource-container concept into a standard version of our para-virtualized Linux 2.6 adoption. Our implementation relies on a previous approach to use resource containers in the context of processor-energy management [Bellosa et al., 2003, Weissel and Bellosa, 2004]. We extended the original version with support for disk energy. No further efforts were needed to manage virtual processor energy; we only had to virtualize the performance counters to get the original version to run (Figure 4.16).

Similar to the host-level subsystem, the energy-aware guest operating system performs scheduling based on energy criteria. In contrast to standard schedulers, it uses resource containers as the base abstraction rather than threads or processes. Each application is assigned to a resource container, which then accounts all energy spent on its behalf. To account virtual processor energy, the resource container implementation retrieves the (virtual) performance-counter values on container switches, and charges the resulting energy to the previously active container. A container switch occurs on every context switch between processes residing in different containers.

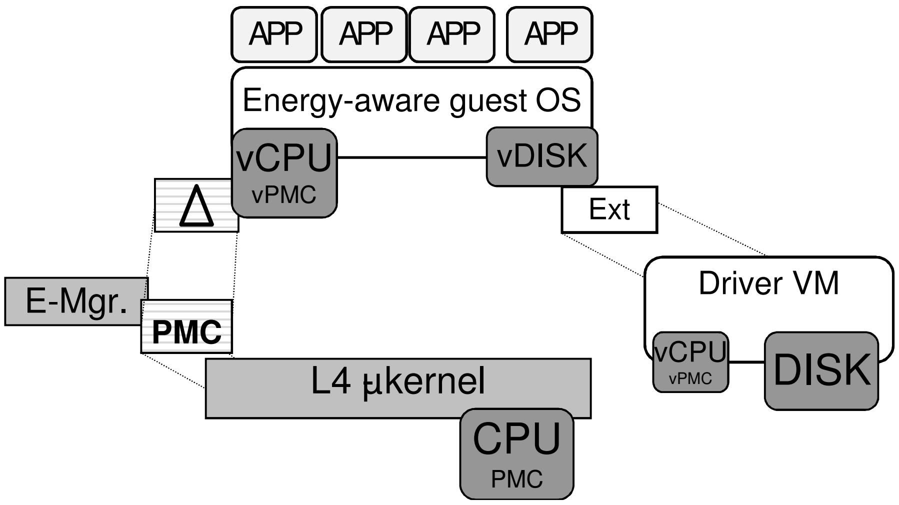

Figure 4.16: Energy-aware guest operating system. The guest OS performs application-specific energy management, based on the virtual processor and disk-energy extensions of our framework. The implementation leverages the resource-container concept, integrated into a standard version of our para-virtualized Linux 2.6 adoption.

To account virtual disk energy, we enhanced the client side of the virtual device driver, which forwards disk requests to the device driver virtual machine. Originally, the custom device driver received single disk requests from the Linux kernel, which contained no information about the user-level application that caused it. We added a pointer to a resource container to every data structure involved in a read or write operation. When an application starts a disk operation, we bind the current resource container to the corresponding page in the page cache. When the kernel writes the pages to the virtual disk, we pass the resource container on to the respective data structures (i.e., buffer heads and `bio` objects). The custom device driver in the client accepts requests in form of `bio` objects and translates them to a request for the device driver virtual machine. When it receives the reply together with the cost for processing the request, it charges the cost to the resource container bound to the `bio` structure. To control the energy consumption of virtual devices, the guest kernel redistributes its own, virtual-machine–wide power limits to subordinate resource containers, and enforces them by means of preemption. Whenever a container exhausts the energy budget of the current period (presently set to 50 ms), it is preempted until a refresh occurs in the next period. A simple user-level application retrieves the virtual machine wide budgets from host-level energy manager and passes them onto the guest kernel via special system calls.

# Chapter 5

# Evaluation

In this chapter, we present experimental results we obtained from our prototype. Our main goal is to demonstrate that our infrastructure provides a viable solution to manage energy in distributed, multi-layered operating systems. We consider two aspects as relevant: the effectiveness of our mechanisms for energy accounting and allocation; and the performance overhead induced by those mechanisms.

The chapter is organized as follows: Section 5.1 presents the experimental setup we used for our evaluation. Section 5.2 presents experiments measuring the effectiveness of distributed energy accounting and allocation at two layers, host level and guest level. Section 5.3 presents experiments quantifying the impact of exposed energy accounting and allocation on the performance of selected benchmarks. Section 5.4 presents a summary and review of our experimental results.

## 5.1 Evaluation Setup

For processor measurements, we used a Pentium D 830 with two cores, each at 3 GHz, running on an Intel 945G motherboard equipped with 2 GBytes RAM. Our implementation is currently limited to single processor systems; we therefore enabled only one core, which always ran at its maximum frequency. When idle, the core consumes about 42 W; under full load, power consumption may be 100 W and more. We performed disk measurements on a Maxtor Diamond Max Plus 9 IDE hard disk with 160GBytes in size, for which we measured an active power of about 6.7 W and idle power of about 4.1 W.

We validated our internal, estimation-based accounting mechanisms by means of an external high-performance data acquisition system (DAQ), which measured the real disk and processor energy consumption. The DAQ system from National Instruments comprises three SCC-AI06 modules, a SC-2345 shielded carrier for the SCC modules, a PCI-6220 data acquisition card, and LabView software in version 7.1.1. We connected the acquisition system with the voltage lines of the processor and disk device, and measured the current by means of precisions resistors, with a sampling rate of 1 kHz.

## 5.2 Effectiveness of Energy Management

The first set of experiments evaluates the effectiveness of accounting and allocating energy consumption, which was one of the essential objectives of energy management. The specific effectiveness goals of our prototype are to enable accurate and comprehensive energy accounting and allocation both for whole virtual machines and for virtualized guest applications. In the following we first present experiments evaluating the accuracy of processor and disk energy accounting; we then present experiments evaluating host-level allocation of processor and disk energy; we finally present experiments on guest-level allocation of processor energy and on faithful legacy processor scheduling.

Note that, although our infrastructure provides two different interaction protocols for processor and disk energy management each, we did not perform all experiments with all combinations, since no additional insights would have emerged from doing so. Rather, we ran each experiment with a selected combination of protocols only. An exception to that rule occurs in the last experiment, where we compare the faithfulness of legacy processor scheduling under different protocol combinations.

### 5.2.1 Processor and Disk-Energy Accounting

To evaluate our approach of distributed energy accounting, we measured the overall energy required for using a virtual disk. For that purpose, we ran a synthetic disk stress test within a Linux guest operating system. The test runs on a virtual hard drive, which is multiplexed on the physical disk by the disk driver virtual machine. The test performs almost no computation, but generates heavy disk load. By opening the virtual disk in *raw* access mode, the test bypasses guest-kernel caches and causes the file I/O to be performed directly to and from user-space buffers. The test permanently reads (writes) consecutive disk blocks of a given size from (to) the disk, until a given total size has been transferred. We performed the test for block sizes from 0.5 KByte up to 64 KByte. We obtained the required energy per block size to run the benchmark from our accounting infrastructure. In the experiment, we used the asynchronous energy-management interaction scheme for both processor and for the disk.

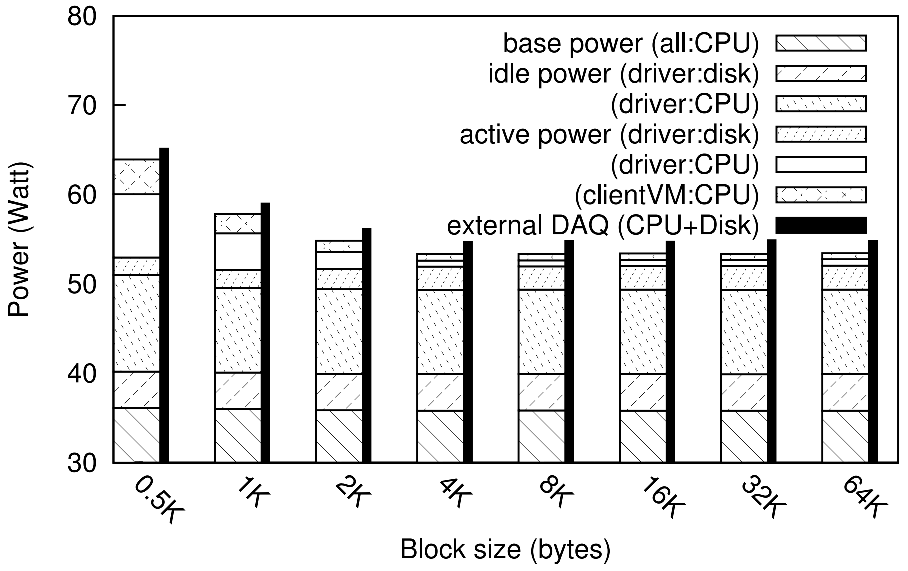

Figure 5.1: Energy distribution for processor and disk during the disk stress test (read case). The thin bars show the real processor and disk power consumption, measured with an external DAQ system.

The results for read and write case are shown in Figure 5.1 and Figure 5.2 respectively. For each size, the figures show the consumption of disk and processor power by the client and the device-driver virtual machine. The lower-most part of each bar shows the base processor-power consumption required by core components such as the microkernel and the user-level virtual-machine monitor parts (36 W); this part is consumed independently of any disk load. The upper parts of

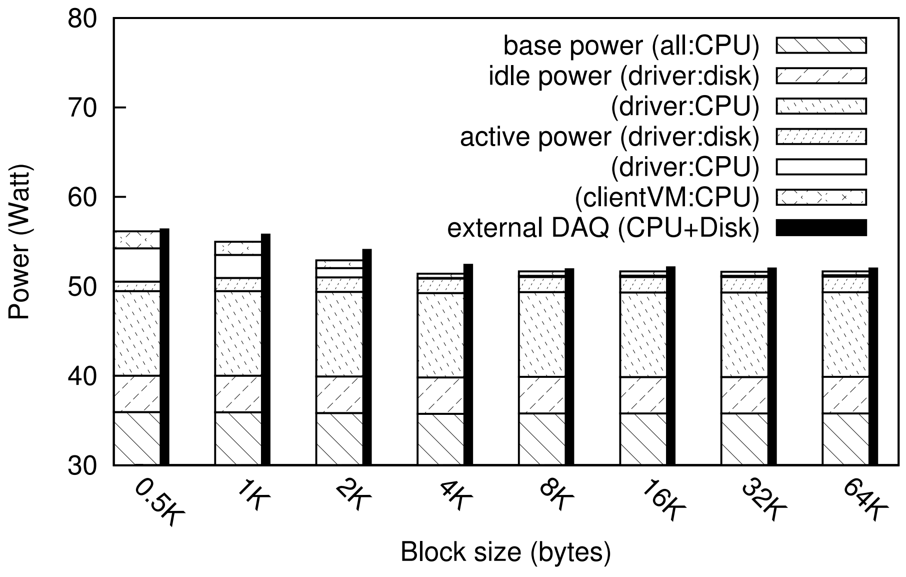

Figure 5.2: Energy distribution for processor and disk during the disk stress test (write case). The thin bars show the real processor and disk power consumption, measured with an external DAQ system.

each bar show the access and idle power consumption caused by the stress test, broken into processor and disk consumption. Since the client virtual machine is the only consumer of the hard disk, it is accounted the complete idle disk power (4.1 W) and processor power (9.4 W) consumed by the driver virtual machine. Since the benchmark saturates the disk, the access disk power consumption of the disk driver mostly stays at its maximum (2.6 W), which is again accounted to the client virtual machine as the only consumer. Access processor power consumption in the driver virtual machine heavily depends on the block size and ranges from 8.4 W for small block sizes down to less than 1 W for large ones; on average, the read case requires more processing (power) than the write case, which is owed to the read throughput being higher than the write throughput, and with it the number of requests per second to process. Note that the overhead for virtualization — both in terms of time and energy — is considerable: in our case, the processor energy costs for processing a virtual disk request may even surpass the costs for handling the request on the physical disk. Finally, access processor power consumption in the client virtual machine varies with the block sizes as well, but at a substantially lower level; the lower level comes unsurprising, as the benchmark bypasses most parts of the disk driver in the client operating system.

The thin bar on the right of each energy histogram shows the real power consumption of the processor and disk, measured with the external DAQ system.

Overall, the results demonstrate that our framework is capable to comprehensively account and attribute the energy required to satisfy disk requests in a modular operating system. The internal accounting records largely correspond with the external measurements.

### 5.2.2 Host-Level Processor-Energy Allocation

To validate our host-level processor-allocation subsystem, we compared the effects of energy-aware scheduling against normal, time based scheduling. For that purpose, we simultaneously ran two Linux virtual machines with distinct energy profiles, and compared their effective power consumption under different scheduling policies. We ran a customized benchmark application in each of the virtual machine, resulting in predictable power consumption: the first benchmark application, `alu`, stresses the processor's ALU, using x86 `bswap` (byte swap), `adc` (add with carry), and `inc` (increment) instructions, which operate on processor registers only. If unconstrained, the `alu` benchmark consumes about 40 W access power. The second benchmark, `pushpop`, stresses both processor registers and memory, by executing series of `push` and `pop` instructions. If unconstrained, `pushpop` consumes about 63 W access power. Mechanics and peculiarities of the benchmarks are unimportant here; we chose them since their power consumption is significantly different.

Figure 5.3 shows the effective power consumption of both virtual machines, as a stacked histogram, for five different scheduling policies. All experiments were done with the synchronous scheduling protocol, that is, with the scheduler running completely outside the kernel. The five different scheduling policies are, from left to right:

**TIME (10/10)** The scheduler operates on the basis of processor time rather than on processor energy, and each virtual machine is alloted a ticket of 10 shares (i.e., both virtual machines are alloted the same amount of time).

**EAS (10/10)** The scheduler operates on the basis of processor energy, and again, each virtual machine is alloted a ticket of 10 shares (i.e., both virtual machines are alloted the same amount of energy).

**TIME (10/5)** The scheduler operates on the basis of processor time, but allots the virtual machine running `pushpop` twice as much time shares as the virtual machine running `alu`.

**EAS (10/5)** The scheduler operates on the basis of processor energy, but allots the virtual machine running `pushpop` twice as much energy shares as the virtual machine running `alu`.

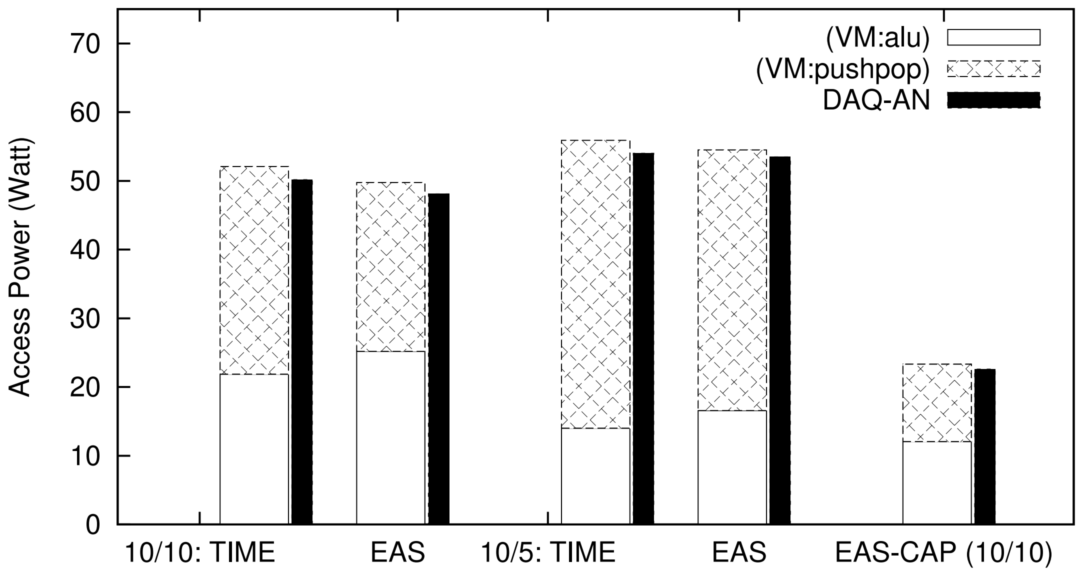

Figure 5.3: Effective power consumption of two virtual machines with different initial power characteristics, for five different energy allocation policies.

**EAS-CAP (10/10)** The scheduler operates on the basis of processor energy, allots both virtual machines the same amount of 10 energy shares, and additionally caps the access power at 25 W.

Note that the power limits are effective limits; strictly spoken, both `alu` and `pushpop` still consume the same amount of power per second *when running*; the scheduler merely reduces each virtual-machine time allotment accordingly, with the result that, over time, the limits are effectively obeyed. The thin bar on the right of each power bar stack shows the real power consumption of the processor, measured with the external DAQ system. The results demonstrate that our framework is capable to accurately apportion the available energy among different virtual machines, and to cap the total energy consumption to user-defined limits while still adhering to virtual-machine specific energy allotments. The scheduling infrastructure is extensible, allowing the policy to be quickly adapted to the actual workload or deployment situation without having to exchange the privileged part of the operating system. Finally, as with the experiments on accounting, the internal accounting and allocation again corresponds with the external measurements.

### 5.2.3 Host-Level Disk-Energy Allocation

To validate the capabilities of disk-energy allocation, we performed a second experiment on a virtual disk, this time with two clients that simultaneously require disk service from the driver. The clients interface with a single disk-driver virtual machine, but operate on distinct hard disk partitions. We set the driver access power limit of client virtual machine 1 to 1W and the limit of client virtual machine 2 to 0.5 W, and periodically obtained driver energy and disk throughput over a period of about 2 minutes. Note that, while those limits concern the disk, they implicitly reduce processor energy as well. For the experiment, we used the asynchronous energy-management interaction protocol schemes for both the processor and disk.

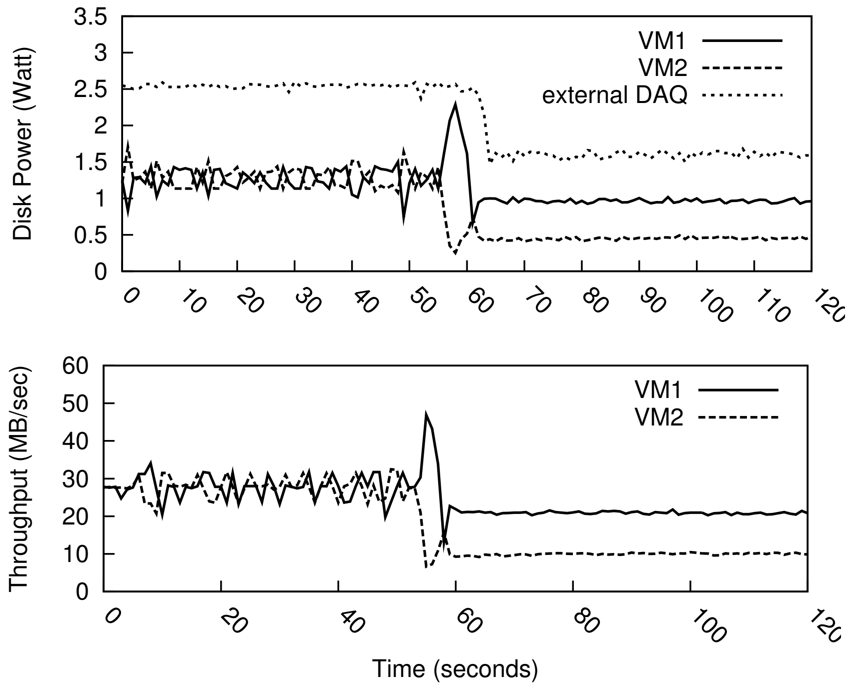

Figure 5.4: Disk-power consumption and throughput of two constrained disk tests simultaneously running in two different guest virtual machines.

Figure 5.4 shows both distributions; we set the limit about 60 seconds after having started the measurements. Note that the implementation causes a short spike of both throughput and energy immediately after having set the respective limits. We attribute the spike to the algorithm's deficiency to level off changing throttle limits more quickly. Overall, however, our experiment demonstrates the driver's capabilities to virtual-machine–specific control over energy consumption. Again, internal accounting and control corresponds with external measurements.

### 5.2.4 Para-Virtualized Processor-Energy Management

In the next experiment, we evaluated the benefits of guest-level energy management; to that end, we compared the effects of enforcing power limits at the host-level against the effects of guest-level enforcement using a para-virtual guest-level energy-management facility. In the first part of the experiment, we ran two instances of the compute-intensive `bzip2` application within an energy-unaware guest operating system. In the unconstrained case, a single `bzip2` instance causes an access processor-power consumption of more than 50 W. The guest, in turn, is alloted an overall processor access power of only 40 W. As the guest is not energy-aware, the limit is enforced by the host-level subsystem. We used the asynchronous protocol variant for host-level processor energy allocation. In the second part, we used an energy-aware guest, which complies with the alloted power itself. It redistributes the budget among the two `bzip2` instances using the resource-container facility. Within the guest, we set the application-level power limits to 10 W for the first, and to 30 W for the second `bzip2` instance. Again, power limits are effective limits and the resource-container implementation only reduces task running times accordingly.

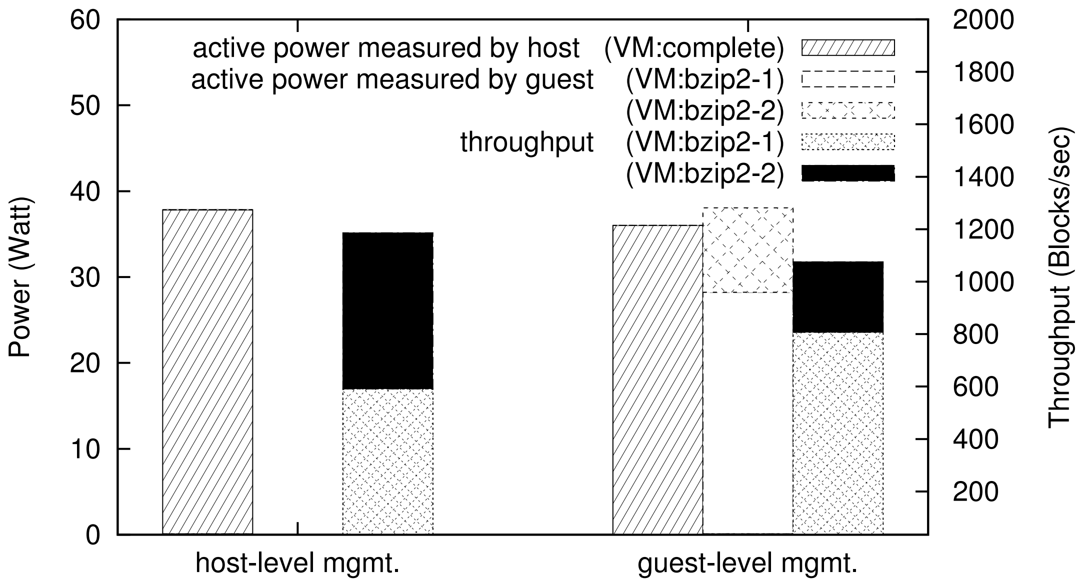

Figure 5.5: Host-level versus guest-level energy redistribution. The figure shows effective power consumption and throughput of two `bzip2` applications, with energy limits enforced either at host-level or at guest-level.

The results are shown in Figure 5.5. For both cases, the figure shows overall access processor power of the guest virtual machine in the leftmost bar, and the throughput broken down to each `bzip2` instance in the rightmost bar. For the guest-level management case on the right side, we additionally obtained the power consumption per `bzip2` instance as seen by the guest's energy-management subsystem itself; it is drawn as the bar in the middle. Note that the guest's view of the power consumption is slightly higher than the view of the host-level energy manager. Hence, the guest imposes somewhat harsher power limits, and causes the overall throughput of both `bzip2` instances to drop compared to host-level control. We do not have a definite answer as to where the drop stems from, but attribute the differences in estimation to the clock drift and inaccurate guest–timer-interrupt delivery to the client.

However, the results are still as expected: host-level control enforces the budgets independent of the guest's particular capabilities — but the enforcement treats all guest's applications as equal and thus reduces the throughput of both `bzip2` instances proportionally. In contrast, guest-level management allows the guest to respect its own user priorities and preferences: it allots a higher power budget to the first `bzip2` instance, resulting in a higher throughput compared to the second instance.

### 5.2.5 Preserving Legacy Processor Scheduling

The next experiment is devoted to validating the effectiveness of our infrastructure to preserve legacy scheduling semantics in guest operating systems. We show how synchronous processor scheduling is able to preserve legacy semantics in paravirtual guest operating systems, whereas both the asynchronous and the original scheduler architectures — at least in a straight-forward implementation —, both trade performance for accurateness.

To determine the implications of user-level scheduling on normal application workloads, we tested how an increasing amount of workload running in distinct virtualized applications — and thus on distinct L4 threads — affects the responsiveness of the overall guest operating system. For that purpose, we generated I/O load on a Linux guest by executing the `netperf` benchmark, which we stressed from an external client through a Gigabit network interface card. At the same time, we also generated processing load *within* the same virtual machine, using `cpuload`, a small user program that loops forever counting bits of a dummy value. We stressed the virtualization of guest–operating-system scheduling logic by executing a growing number of `cpuload` instances in parallel to the `netperf` application. As a result a growing number of Linux guest applications and subsequent L4 threads hosting them become runnable at the same time.

The results are given in Figure 5.6. The figure draws `netperf` over the increasing number of `cpuload` instances, with the synchronous, the asynchronous, and the original scheduling logic. The figure additionally shows the results with

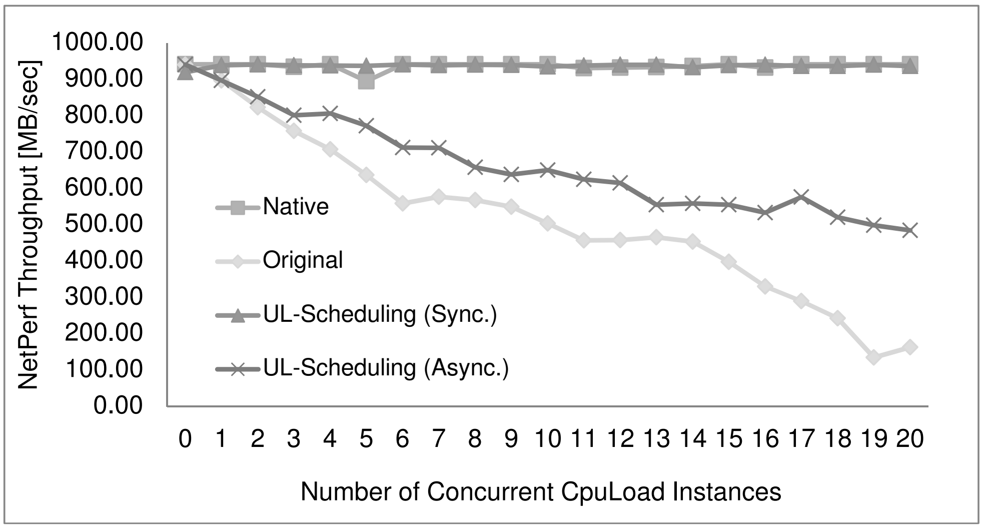

Figure 5.6: Accuracy of legacy processor-scheduling semantics, measured as drop in `netperf` throughput with increasing number of concurrent `cpuload` instances for different interaction protocols, as well as for the original L4 implementation and native Linux.

the same setup running in a native environment, that is, with Linux running on bare hardware. Not shown is the processor utilization, which we measured during all test runs: with no `cpuload` instance running, utilization was 19, 33, 36, and 45 per cent in the native, original, synchronous scheduling, and asynchronous scheduling cases. Whenever more than one `cpuload` was running, utilization rose to 100 per cent. The benefits of synchronous user-level scheduling become evident with the increasing number of `cpuload` instances. With the synchronous protocol, `netperf` throughput stays constant, a behavior that is conforming to the outcome of the native installation, where `netperf` throughput remains undisturbed from additional `cpuload` instances as well. However, with both the asynchronous and the original versions, `netperf` throughput drops with increasing numbers of `cpuload` instances. We can attribute that behavior to the L4 kernel scheduler counteracting the intentions of the Linux guest scheduler. The synchronous scheduling architecture allows the virtual-machine monitor to accurately map Linux scheduling and dispatching decisions to the underlying L4 threads, since the kernel does not perform any implicit scheduling decisions. In contrast, both the asynchronous and the original L4 scheduling architecture *do* perform scheduling decisions at timer interrupts — the asynchronous version based on its time-based stride scheduling policy, and the original version based on its round-robin scheduler. The asynchronous version still improves on the original version, in that it allows L4 scheduling decisions to be tracked and the resulting distribution of processor time and energy to be accounted; also it allows the virtual machine scheduler to allocate processor time to whole virtual machines (thread groups) rather than to individual threads only. Still both the asynchronous user-level and the original scheduler architecture may lead to situations where L4 scheduler decisions contradict Linux scheduler decisions.

Note that the purpose of the experiment is not to show that using a kernel-level scheduler inherently leads to poor network performance. More accurate scheduling (and better throughput results) can also be achieved with the asynchronous and the original L4 scheduling architecture, by means of user code that prevents the L4 kernel scheduling policy from selecting the "wrong" thread for dispatching. Such a scheme could be either implemented by modifying the guest–operating-system scheduler to become aware of and coordinate with the L4 kernel scheduler, or by adding functionality to the virtual-machine monitor that makes sure all threads except the one Linux designates to run are not runnable. Both solutions are not generic, however, in that they still leave a global policy in the host kernel.

## 5.3 Performance of Energy Management

The second set of experiments strives to quantify the performance overhead associated with our accounting and allocation infrastructure. In the following, we first evaluate the costs of adding the user-controlled processor-energy management logic to the microkernel. We then evaluate the impact of combined processor and disk management in terms of disk throughput and of processor utilization.

### 5.3.1 Performance of Processor Management

To evaluate the performance impact of our processor-scheduling architecture, we deemed two questions as relevant: first, we wanted to evaluate how expensive our scheduling architecture is in terms of the basic absolute overhead. Second, we wanted to find out how those basic costs are reflected in overall application performance.

To quantify the basic overhead of adding user-level scheduling logic to the kernel, we compared the performance of IPC on the original L4 version against the same operation on our new L4 version. We used a ping-pong IPC microbenchmark that creates a pair of communicating threads that perform message transfers back and forth. The benchmark measures round-trip time for different message sizes; since transfer times are short, the benchmark conveys messages repeatedly. Figures 5.8 and 5.7 list the results in $\mu$s, for messages within address spaces (intra-AS) and across address spaces (inter-AS).

With the original scheduling architecture, IPC costs are about 0.18 $\mu$s for intra-AS and about 0.5 $\mu$s for inter-AS IPCs. With the asynchronous protocol, IPC costs

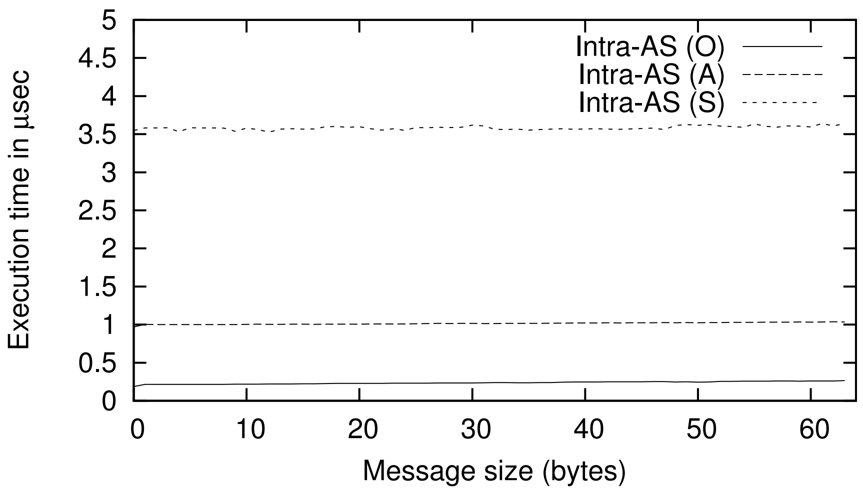

Figure 5.7: Intra-address space IPC costs of L4 with support for user-level scheduling in the synchronous (S) and asynchronous (A) version, compared to the original (O) version.

increase to about 1.0 $\mu$s and 1.4 $\mu$s. A large portion of these costs stems from the hardware costs for reading out performance counters: the `rdpmc` instruction requires about 0.042 $\mu$s, thus reading out 8 performance counters alone yields an overhead of about 0.34 $\mu$s. We attribute the rest of performance degradation for the asynchronous scheme to the changed kernel scheduling protocol and implementation. For the synchronous scheduling protocol, IPC costs increase dramatically to 10.5 $\mu$s and 11 $\mu$s respectively. This comes unsurprising, as each IPC is interposed by the user-level scheduler, requiring extra transitions between threads and from kernel to user level.

Our micro-benchmark tests give us detailed insights on a specific hot spot of our architecture and protocols; however, they are not necessarily a predictor for overall application performance. To quantify those effects at application level, we ran another set of benchmarks as within a virtual machine, and compared their results across the different scheduler protocols. We ran three different benchmarks within a Linux guest operating system instance: i) a compile of the Linux kernel 2.4.20, with the source code served out of a ram disk in memory to avoid I/O operations, ii) the `netperf` benchmark, which we stressed from the local client via the loop-back device, and iii) `netperf`, this time stressed from an external client through a Gigabit network interface card. Since this benchmark is I/O-bound, we additionally measured processor utilization during the run (all other benchmarks in this experiment showed a processor utilization of 100 per cent).

The results are depicted in Table 5.9. For the sake of completeness, the fig-

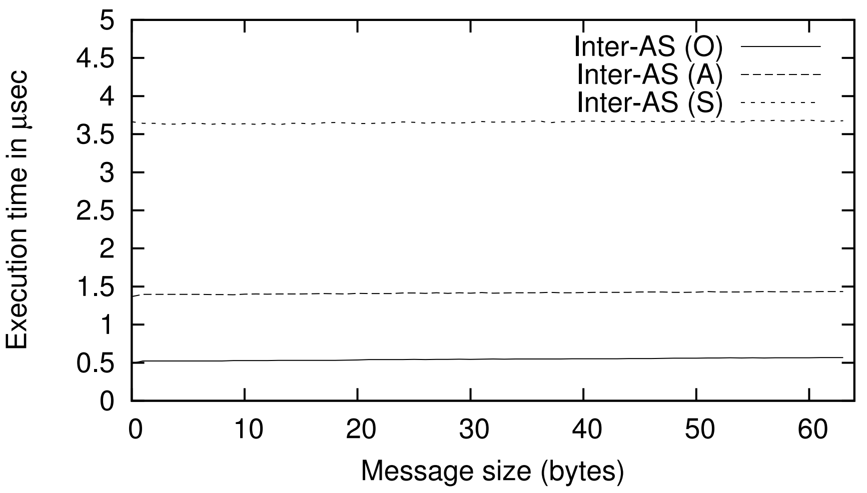

Figure 5.8: Inter-address space IPC costs of L4 with support for user-level scheduling in the synchronous (S) and asynchronous (A) version, compared to the original (O) version.

| **Workload** | **Native** | **Original** | **ASYNC** | **SYNC** |
|---|---|---|---|---|
| Kernel build [min] | 1:47 | 1:58 | 1:58 | 2:07 |
| Netperf Local [MB/sec] | 5738 | 5463 | 5326 | 4348 |
| Netperf Remote [MB/sec] | 942 | 941 | 941 | 926 |
| Netperf Remote CPU util. [per cent] | 19 | 33 | 36 | 45 |

Table 5.9: Kernel-build time [min] and Netperf throughput T [$\frac{MB}{sec}$] of our asynchronous and synchronous scheduling protocols, compared to the original infrastructure.

ure additionally shows the results with the same setup running in a native environment, that is, with Linux running on bare hardware. With asynchronous scheduling, kernel-build performance stays on par with the original version, while `netperf` performance loss is 2.5 per cent in throughput in the local case, and 9 per cent in processor utilization in the remote case. With synchronous scheduling, the performance loss relative to original is 7.5 per cent for kernel build, 20.4 per cent in throughput for local `netperf`, and 1.6 per cent in throughput and 36 per cent in processor utilization for remote `netperf`.

### 5.3.2 Performance of Combined Disk and Processor Management

Finally, we wanted to evaluate the performance of our energy-management infrastructure in a more complex scenario. For that purpose, we evaluated the performance impact of our management infrastructure on the throughput of our disk-driver subsystem. We used the same disk benchmark we already used for evaluating disk accounting; again, we ran the benchmark in its own guest virtual machine, on a virtual disk provided by a driver virtual machine. We disabled all processor and disk budgets and let the benchmark run unconstrained, but with the infrastructure for energy accounting and allocation still in place. We then compared the setup against an original version of the virtualization environment lacking any energy-management infrastructure. We measured performance for all protocol combinations, that is, for synchronous and asynchronous processor scheduling, and for synchronous and asynchronous disk scheduling. Note that we compare the performance of two virtualization environments; a performance comparison of the original virtual disk driver against native drivers is not of our interest here; it can be found in [LeVasseur et al., 2004].

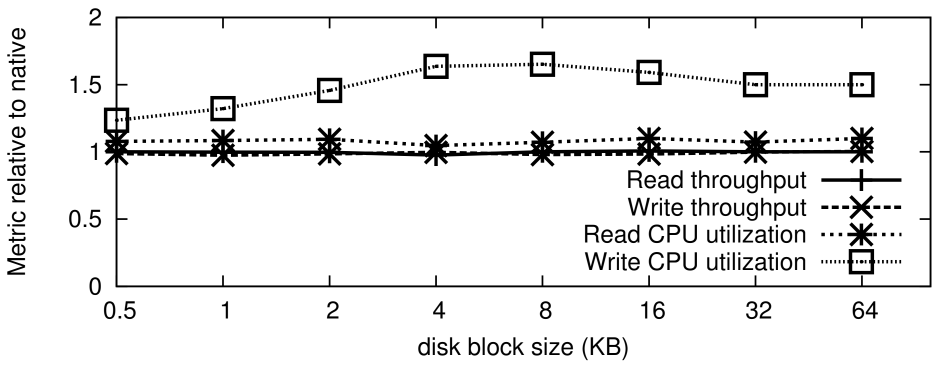

Figure 5.10: Throughput and processor utilization of our management infrastructure (asynchronous processor scheduling and disk scheduling) for disk streaming read and write, relative to native virtualization for disk streaming read and write.

The results are depicted in Figures 5.10, to 5.13. Disk throughput is largely identical to the original version, independent whether protocols are asynchronous or synchronous. With synchronous processor scheduling, but asynchronous disk scheduling, throughput is higher than original, but only slightly, increasing from 46 to 48 MBytes/sec on average. Processor utilization, in turn, increases in all versions, asynchronous and synchronous ones. The increase occurs at a low level in absolute terms, but is still significant relatively: with asynchronous processor and disk scheduling, utilization increases from 5.1 to 6.1 per cent on average. With

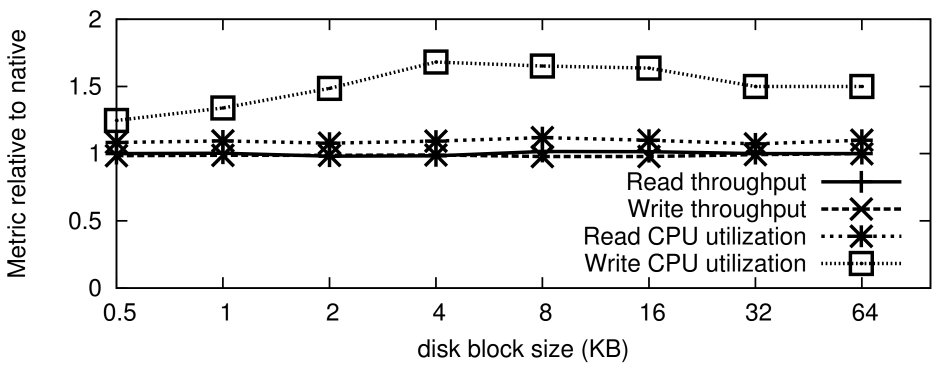

Figure 5.11: Throughput and processor utilization of our management infrastructure (asynchronous processor scheduling, synchronous disk scheduling) for disk streaming read and write, relative to native virtualization.

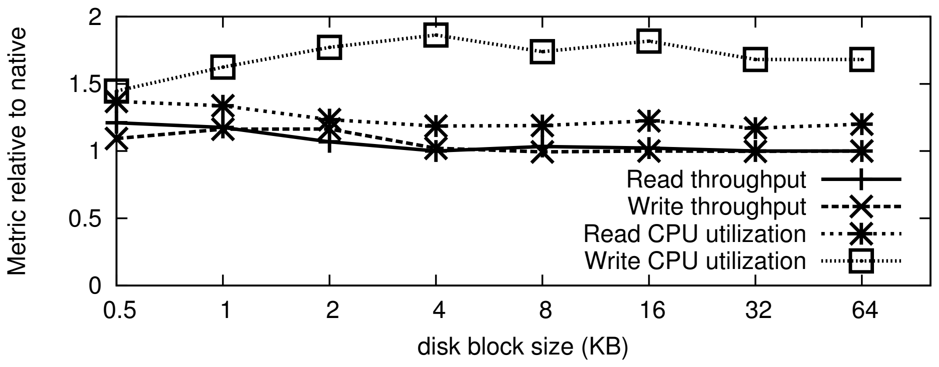

Figure 5.12: Throughput and processor utilization of our management infrastructure (synchronous processor scheduling, asynchronous disk scheduling) for disk streaming read and write, relative to native virtualization.

asynchronous processor scheduling but synchronous disk scheduling, utilization rises to 6.2 per cent on average. With synchronous processor scheduling but asynchronous disk scheduling, utilization rises to 7.3 per cent on average, and with synchronous processor and disk scheduling, utilization rises to 7.6 per cent on average. We credit the differences in throughput and processor utilization to the different mechanics of scheduling and interrupt handling in the different versions.

Note that, in general, the performance overhead is lower relatively with smaller block sizes. We believe that this effect stems from a high absolute overhead of our energy-management infrastructure, which, however, has less impact with small block sizes since there, the absolute *virtualization* overhead is quite high as well. Altogether, however, we do take the results as a general indicator of serious per-

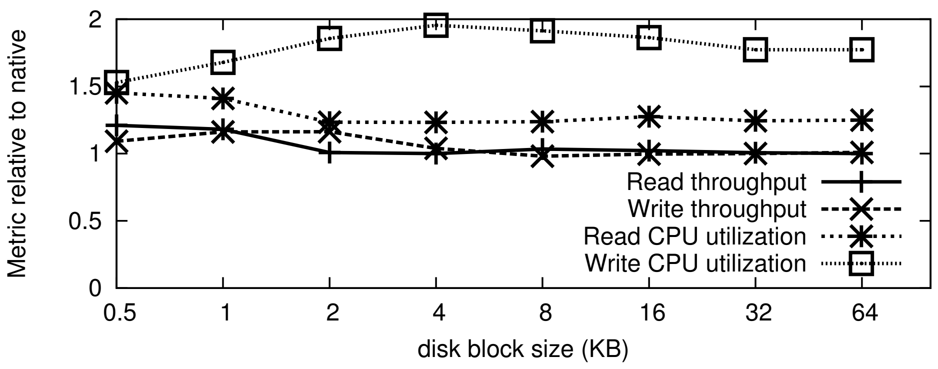

Figure 5.13: Throughput and processor utilization of our management infrastructure (synchronous processor scheduling and disk scheduling) for disk streaming read and write, relative to native virtualization.

formance drawbacks with synchronous scheduling in the current version of the prototype. Future work needs to be done to evaluate how this overhead can be mitigated further, for instance, by using module isolation concepts that are less costly than the address space we are currently limited to in our prototype.

## 5.4 Summary

In summary, we draw the following conclusions from our experiments:

**Effectiveness of energy management** Our experiments show that our prototype is capable of accurately accounting processor and disk energy consumption to different notions of applications at runtime. Results show that our framework deals with layering and interdependencies, as exemplified by our virtualized disk service, for which we track down energy consumption to both the physical processor and the physical disk. Internal accounting records furthermore correspond with external measurements of processor and disk energy consumption. Our experiments also show that our prototype is capable of accurately allocating processor and disk energy to different notions of applications. Results show that our framework stipulates the energy consumption of individual hardware devices, for complete virtual machines at the host-level, and for individual virtualized applications running within energy-aware guest operating systems. Our allocation mechanisms are extensible in terms of energy-management implementation and policies. They allow, for instance, scheduling to be quickly changed from energy-oriented to throughput-oriented. Finally, our synchronous processor scheduling scheme enables better and more accurate user-controlled scheduling than existing microkernel architectures.

**Performance of energy management** Our experiments comparing our prototype against vanilla versions of the L4 microkernel and the virtualization environment running atop yields mixed results and do not allow us to draw a uniform or definitive conclusion: overheads for our processor energy management may be dramatic (up to factor 10x) for selected micro-benchmarks scenarios, but range from 0 to 9 per cent respectively 7.5 to 36 per cent at application layer, depending on the desired accuracy of scheduling. Overheads for combined processor and disk management are zero with respect to disk throughput, and limited to an increase in processor utilization, from 5.1 to at most 7.6 per cent, that is, by at most 49 per cent. Altogether, the performance results indicate that there is a trade-off between accuracy and performance in energy management, and therefore support our belief that it should be up to policy developer to choose the point in the trade-off space — which our infrastructure enables by offering different management protocols. Note again that, at present, we leave the question open how and when the particular protocol should be chosen and leave it up to future work to investigate an appropriate (and possibly runtime) configuration mechanism.

# Chapter 6

# Conclusion

This chapter concludes the thesis. We begin with summarizing its contributions in Section 6.1; we then state suggestions for future work in Section 6.2.

## 6.1 Contributions of the Thesis

This thesis addresses the problem of how operating-system energy management can be facilitated in presence of operating-system structures becoming increasingly modularized. We have developed a framework for system-level support for managing energy in distributed, multi-layered operating-system environments, as they are becoming common in today's computer systems. Our framework strives to enable energy-awareness and energy management if the resource-management subsystem is distributed and scattered among operating-system modules, rather than being centralized and monolithic. The thesis makes the following research contributions:

**A Model for Modularization-Aware Energy Management** Instead of treating operating-system energy management as centralized and privileged, we model it as a feedback loop involving resources, activities, and energy-policy–management modules in different compartments. We furthermore propose to solely rely on the notion of energy as the base abstraction, as it quantifies energy effects a distributable and partitionable way.

**Exposed and Distributed Energy Accounting and Allocation** We propose distributed and exposed energy-accounting and energy-allocation mechanisms for extensible energy management between policy managers and resource providers. Exposed accounting thereby allows policy managers to track how the energy is spent in the system, from originating activities down to the hardware devices. Exposed allocation enables them to dynamically and remotely control energy consumption and other effects for both software activities and hardware devices.

**Energy Management Interaction Protocols** Exposing energy accounting and allocation requires propagating accounting state and resource-allocation decisions between involved modules. Different communication protocols are thinkable; we postulate that the optimal protocol is largely defined by two opposing factors, the timeliness requirements of the energy policies, and the protocol performance overhead induced by module isolation. We explore two different protocols, a synchronous and an asynchronous variant.

**Virtualization of Energy Effects** Virtualization layers have become very popular in operating systems, and a de-facto standard to solve legacy-compatibility problems; we therefore support a notion of virtualized energy effects, to provide the additional benefit of a development path towards fine-grain, energy-aware resource management for virtualized applications.

**Prototype Implementation** To demonstrate our framework design we have built a prototype for a microkernel-based component operating system. We use an instance of the L4 microkernel, and, atop, a platform-virtualization layer providing compatibility to Linux applications. Our prototype supports processor and disk energy management both at the physical and at the virtual layer. To that end, it features distributed and exposed mechanisms for accounting and allocation of processor and disk energy both to complete virtual machines and to individual virtualized applications. We have validated our approach in two aspects, in the effectiveness and in the performance of our energy accounting and allocation mechanisms.

**Validation of Effectiveness** Our experimental results show that our prototype is capable of accurately accounting and allocating processor and disk energy consumption to different notions of applications at runtime. Our framework deals with layering and interdependencies, as exemplified by a virtualized disk service, for which we track down energy consumption on both the processor and the physical disk. Internal accounting records cumulatively correspond with external measurements of processor and disk energy consumption. Our allocation mechanisms enable extensible policies, which can be quickly changed from energy-oriented to throughput-oriented.

**Validation of Performance** Our experimental results further highlight the interplay between accuracy and performance of energy management: overheads for processor energy management may be dramatic for micro-benchmarks, but percolate to application level at a more moderate level, where they range

range from 0 to 36 for different application benchmarks. Overheads for combined processor and disk energy management are zero in terms of disk throughput and limited to an increase in processor utilization, from 5.1 to at most 7.6 per cent, all depending on the desired accuracy of energy management. We draw the conclusion it should be up to developers of energy management policies, which point in the trade-off space between accuracy and performance to choose; our energy management infrastructure strives to enable those different decisions to be put in effect, by offering different management protocols.

## 6.2 Suggestions for Future Work

We see our work as a support infrastructure to develop and evaluate energy management strategies for modular operating systems. We consider three areas to be important for future work:

First, and foremost, our work is currently focused on energy-management mechanisms rather than on policies; while we do evaluate our mechanisms with a small set of policies, future work has to be done to explore the large body of existing algorithms and policies for operating-system energy management, and its applicability to modular operating systems. First steps towards this direction could be to investigate more complex processor or disk-energy–management policies such as thermal management [Merkel and Bellosa, 2006], or energy-efficient caching and pre-fetching [Papathanasiou and Scott, 2004]

Second, while our work focused on mechanisms for modular energy management, there is still a large opportunity for improving our proposed mechanisms. For example, we have put a strong emphasis on software mechanisms, since we expect that hardware mechanisms such as sleep states can be integrated in rather straight-forward fashion; still, future work has to be done to thoroughly explore the interplay of different hardware power states on the one side, and distributed energy accounting and modular policy management on the other side. As another example, at present, we offer two protocols for energy management interaction, but leave the question open how and when the particular protocol should be chosen; future work must investigate how the selection of protocols can be facilitated in a flexible way, for example, by offering a dynamic configuration system, or by allowing different scheduling domains to use different protocols at runtime. Also, future work needs to be done to investigate if the performance overheads of the present mechanisms can be mitigated. Particularly the synchronous protocol schemes presently add serious performance (and subsequently energy) overhead, which have to be offset from potential energy savings achieved through our infrastructure. A potential performance optimization would be to put policy management modules into more light-weight protection domains than address spaces, in order to lower the overhead of transitioning between policy modules and device drivers.

Third, our implementation and particularly the evaluation focused on single-node, x86-based stationary computer system. Future work needs to be done to investigate how our approach proves effective in other application scenarios such as servers with multiple processors or devices, or the embedded and mobile space. First steps in this direction could be the extension of our distributed processor accounting and allocations schemes towards multi-processor systems, which could enable energy-aware load migration or thermal balancing; or the development of a uniform accounting and allocation policy and scheme for managing battery lifetime in mobile, modular operating systems.

# Acknowledgements

First, and foremost, I would like to thank my advisor Frank Bellosa. Without his assistance and encouragement this work would not have been possible. I would also like to thank Gernot Heiser, my secondary reviewer, for spending his precious time for reading my thesis and attending my defense. During my graduate program, I had the pleasure to supervise and work with several thesis students, notably Max Laier, Michael Schilli, Sebastian Reichelt, Sebastian Biemüller, Christian Lang, and Marcus Reinhardt. With their projects, insights, and experiences they contributed quite some bits and pieces to my own PhD thesis project.

I would like to thank the members of the System Architecture Group at University of Karlsruhe for providing a collegial work environment. I would notably like to thank my roommate Andreas Merkel for the discussions and idea bouncing, and James McCuller for providing a most professional IT environment and for being a colleague I enjoyed chatting and sharing thoughts with. I am also thankful to the present and former members of the L4 crowd at Karlsruhe, notably Uwe Dannowski, Joshua LeVasseur, Espen Skoglund, and Volkmar Uhlig, with whom I enjoyed working on L4Ka microkernel technology and many other fun projects. Special thanks go to Volkmar Uhlig personally, for the many interesting and inspiring discussions I had with him (not only about work matters), and for encouraging and helping me to get an internship position at IBM Research for half a year during my PhD program.

Credits go to my kids Hannah and Jonas, for showing me day to day that life has more to offer than just profession and career. Finally I want to thank my beloved Katrin for untertaking with me both graduate studies in particular and life in general, and for sharing and helping to alleviate the many situations of self-doubt and loneliness that are inevitably part of the PhD endeavor.

# Bibliography

[Accetta et al., 1986] Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., and Young, M. (1986). Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, pages 93–112, Atlanta, GA, USA.

[Adams and Agesen, 2006] Adams, K. and Agesen, O. (2006). A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, San Jose, CA, USA.

[Ainsworth et al., 2008] Ainsworth, P., Echenique, M., Padzieski, B., Villalobos, C., Walters, P., and Landon, D. (2008). Going green with IBM Systems Director. IBM Red Book.

[Allalouf et al., 2009] Allalouf, M., Arbitman, Y., Factor, M., Kat, R., Meth, K., and Naor, D. (2009). Storage modeling for power estimation. In *Proceedings of the The Israeli Experimental Systems Conference*, pages 107–121, Haifa, Israel.

[Amsden et al., 2006] Amsden, Z., Arai, D., Hecht, D., Holler, A., and Subrahmanyam, P. (2006). VMI: An interface for paravirtualization. In *Proceedings of the 2006 Ottawa Linux Symposium*, pages 363–378, Ottawa, Canada.

[Anand et al., 2004] Anand, M., Nightingale, E. B., and Flinn, J. (2004). Ghosts in the machine: Interfaces for better power management. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*, pages 3–23, Boston, MA, USA.

[Anderson et al., 1991] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. (1991). Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 95–109, Pacific Grove, CA, USA.

[Appavoo et al., 2002a] Appavoo, J., Auslander, M., DaSilva, D., Edelsohn, D., Krieger, O., Ostrowski, M., Rosenburg, B., Wisniewski, R., and Xenidis, J. (2002a). Scheduling in K42. White Paper, IBM Research.

[Appavoo et al., 2002b] Appavoo, J., Auslander, M., DaSilva, D., Edelsohn, D., Krieger, O., Rosenburg, M. O. B., Wisniewski, R., and Xenidis, J. (2002b). Utilizing Linux kernel components in K42. Technical report, IBM Watson Research.

[Aron et al., 2001] Aron, M., Deller, L., Elphinstone, K., Jaeger, T., Liedtke, J., and Park, Y. (2001). The SawMill framework for virtual memory diversity. In *Proceedings of the 6th Asia-Pacific Computer Systems Architecture Conference*, pages 3–10, Bond University, Gold Coast, QLD, Australia.

[Banga et al., 1999] Banga, G., Druschel, P., and Mogul, J. C. (1999). Resource containers: A new facility for resource management in server systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 45–58, Berkeley, CA, USA.

[Barham et al., 2003] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating System Principles*, pages 164–177, Bolton Landing, NY, USA.

[Bellosa, 2000] Bellosa, F. (2000). The benefits of event-driven energy accounting in power-sensitive systems. In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 37–42, Kolding, Denmark.

[Bellosa et al., 2003] Bellosa, F., Weissel, A., Waitz, M., and Kellner, S. (2003). Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, pages 37–42, New Orleans, LA, USA.

[Benini et al., 2000] Benini, L., Bogliolo, A., and Micheli, G. D. (2000). A survey of design techniques for system-level dynamic power management. *IEEE Transactions on Very Large Scale Integration Systems*, 8(3):299–316.

[Bershad et al., 1995] Bershad, B. N., Chambers, C., Becker, D., Sirer, E. G., Fiuczynski, M., Savage, S., and Eggers, S. (1995). Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 267–284, Copper Mountain, CO, USA.

[Bianchini and Rajamony, 2004] Bianchini, R. and Rajamony, R. (2004). Power and energy management for server systems. *IEEE Computer*, 37(11):68–76.

[Biemueller and Dannowski, 2007] Biemueller, S. and Dannowski, U. (2007). L4-based real virtual machines – an API proposal. In *Proceedings of the 1st International Workshop on MicroKernels for Embedded Systems*, pages 36–42, Sydney, Australia.

[Brill, 2007] Brill, K. G. (2007). The invisible crisis in the data center: The economic meltdown of moore's law. White Paper, The Uptime Institute.

[Carrera et al., 2003] Carrera, E. V., Pinheiro, E., and Bianchini, R. (2003). Conserving disk energy in network servers. In *Proceedings of the 2003 International Conference on Supercomputing*, pages 86–97, San Francisco, CA, USA.

[Chase et al., 2001] Chase, J., Anderson, D., Thakur, P., and Vahdat, A. (2001). Managing energy and server resources in hosting centers. In *Proceedings of the 18th Symposium on Operating System Principles*, pages 103–116, Banff, Canada.

[Chou et al., 2001] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. (2001). An empirical study of operating system errors. In *Proceedings of the 18th Symposium on Operating System Principles*, pages 73–88, Banff, Canada.

[Citrix Systems Corporation, 2009] Citrix Systems Corporation (2009). Citrix XenServer. `http://www.citrix.com/xenserver`.

[Colarelli and Grunwald, 2002] Colarelli, D. and Grunwald, D. (2002). Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 International Conference on Supercomputing*, pages 1–11, Baltimore, MD, USA.

[Delaluz et al., 2001] Delaluz, V., Kandemir, M., Vijaykrishnan, N., Sivasubramaniam, A., and Irwin, M. J. (2001). Hardware and software techniques for controlling dram power modes. *IEEE Transactions on Computers*, 50(11):1154–1173.

[Delaluz et al., 2002] Delaluz, V., Sivasubramaniam, A., Kandemir, M. T., Vijaykrishnan, N., and Irwin, M. J. (2002). Scheduler-based DRAM energy management. In *Proceedings of the 39th Design Automation Conference*, pages 697–702, New Orleans, LA, USA.

[des Places et al., 1996] des Places, F. B., Stephen, N., and Reynolds, F. D. (1996). Linux on the OSF Mach3 microkernel. In *Proceedings of the Conference on Freely Distributable Software*, pages 33–46, Boston, MA, USA.

[Druschel et al., 1997] Druschel, P., Pai, V. S., and Zwaenepoel, W. (1997). Extensible kernels are leading OS research astray. In *Proceedings of 6th Workshop on Hot Topics in Operating Systems*, pages 38–42, Cape Cod, MA, USA.

[Economou et al., 2006] Economou, D., Rivoire, S., Kozyrakis, C., and Ranganathan, P. (2006). Full-system power analysis and modeling for server environments. In *Proceedings of the Workshop on Modeling, Benchmarking, and Simulation*, Boston, MA, USA.

[Elnozahy et al., 2002] Elnozahy, M., Kistler, M., and Rajamony, R. (2002). Energy-efficient server clusters. In *Proceedings of the 2nd Workshop on Power-Aware Computing Systems*, pages 179–196, Cambridge, MA, USA.

[Elnozahy et al., 2003] Elnozahy, M., Kistler, M., and Rajamony, R. (2003). Energy conservation policies for web servers. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, pages 59–67, Seattle, WA, USA.

[Elphinstone et al., 2007] Elphinstone, K., Greenaway, D., and Ruocco, S. (2007). Lazy scheduling and direct process switch — merit or myths? In *Proceedings of the 3rd Workshop on Operating System Platforms for Embedded Real-Time Applications*, pages 69–77, Pisa, Italy.

[Engler et al., 1995] Engler, D. R., Kaashoek, M. F., and O'Toole, J. (1995). Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 251–266, Copper Mountain, CO, USA.

[Ethier, 2004] Ethier, S. (2004). Application-driven power management. In *Proceedings of the SDR 04 Technical Conference and Product Exposition.*, Phoenix, AZ, USA.

[Fan et al., 2001] Fan, X., Ellis, C. S., and Lebeck, A. R. (2001). Memory controller policies for DRAM power management. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 129–134, Huntington Beach, CA, USA.

[Fan et al., 2007] Fan, X., Weber, W.-D., and Barroso, L. A. (2007). Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 13–23, San Diego, CA, USA.

[Felter et al., 2005] Felter, W., Rajamani, K., Keller, T., and Rusu, C. (2005). A performance-conserving approach for reducing peak power consumption in server systems. In *Proceedings of the 2005 International Conference on Supercomputing*, pages 293–302, Cambridge, MA, USA.

[Femal and Freeh, 2005] Femal, M. E. and Freeh, V. W. (2005). Boosting data center performance through non-uniform power allocation. In *Proceedings of the 2nd International Conference on Autonomic Computing*, pages 250–261, Seattle, WA, USA.

[Filani et al., 2008] Filani, D., He, J., Gao, S., Rajappa, M., Kumar, A., Shah, P., and Nagappan, R. (2008). Dynamic data center power management. Trends, issues, and solutions. *Intel Technology Journal*, 12(1):59–68.

[Flautner and Mudge, 2002] Flautner, K. and Mudge, T. N. (2002). Vertigo: Automatic performance-setting for Linux. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 105–116, Boston, MA, USA.

[Flinn and Satyanarayanan, 1999] Flinn, J. and Satyanarayanan, M. (1999). Energy-aware adaptation for mobile applications. In *Proceedings of the 17th Symposium on Operating System Principles*, pages 48–63, Charleston, SC, USA.

[Ford and Susarla, 1996] Ford, B. and Susarla, S. R. (1996). CPU inheritance scheduling. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 91–105, Berkeley, CA, USA.

[Forest, 2008] Forest, W. (2008). Revolutionizing data center efficiency. The Uptime Institute Green Enterprise Computing Symposium.

[Forin et al., 1991] Forin, A., Golub, D., and Bershad, B. (1991). An I/O system for Mach 3.0. In *Proceedings of the 2nd USENIX Mach Symposium*, pages 163–176, Monterey, CA, USA.

[Fraser et al., 2004] Fraser, K., Hand, S., Neugebauer, R., Pratt, I., Warfield, A., and Williamson, M. (2004). Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, Boston, MA, USA.

[Gefflaut et al., 2000] Gefflaut, A., Jaeger, T., Park, Y., Liedtke, J., Elphinstone, K., Uhlig, V., Tidswell, J. E., Deller, L., and Reuther, L. (2000). The SawMill multiserver approach. In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 109–114, Kolding, Denmark.

[Goel and Duchamp, 1996] Goel, S. and Duchamp, D. (1996). Linux device driver emulation in Mach. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 65–74, San Diego, CA, USA.

[Goldberg, 1972] Goldberg, R. P. (1972). *Architectural Principles for Virtual Computer Systems*. PhD thesis, Division of Engineering and Applied Physics, Harvard University, Cambridge, MA, USA.

[Gomaa et al., 2004] Gomaa, M., Powell, M. D., and Vijaykumar, T. N. (2004). Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 260–270, Boston, MA, USA.

[Govidan et al., 2009] Govidan, M. S. S., Lefurgy, C., and Dholakia, A. (2009). Using on-line power modeling for server power capping. In *Proceedings of the 2009 Workshop on Energy Efficient Design*, Austin, TX, USA.

[Govil et al., 1995] Govil, K., Chan, E., and Wasserman, H. (1995). Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking*, pages 13–25, Berkeley, CA, USA.

[Goyal et al., 1996] Goyal, P., Guo, X., and Vin, H. M. (1996). A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 107–121, Berkeley, CA, USA.

[Green Hills Software, 2009] Green Hills Software (2009). INTEGRITY real-time operating system. `http://www.ghs.com/products/rtos/integrity.html`.

[Gurumurthi et al., 2003] Gurumurthi, S., Sivasubramaniam, A., Kandemir, M., and Franke, H. (2003). DRPM: Dynamic speed control for power management in server class disks. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 169–179, New York, NY, USA.

[Haeberlen et al., 2000] Haeberlen, A., Liedtke, J., Park, Y., Reuther, L., and Uhlig, V. (2000). Stub-code performance is becoming important. In *Proceedings of the 1st Workshop on Industrial Experiences with Systems Software*, pages 31–38, San Diego, CA, USA.

[Hand et al., 2005] Hand, S., Warfield, A., Fraser, K., Kotsovinos, E., and Magenheimer, D. (2005). Are virtual machine monitors microkernels done right?

In *Proceedings of 10th Workshop on Hot Topics in Operating Systems*, pages 95–99, Santa Fe, NM, USA.

[Hansen, 1970] Hansen, P. B. (1970). The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241.

[Hanson and Krogh, 1995] Hanson, R. and Krogh, F. (1995). Solving nonlinear least squares and nonlinear equations. `http://www.netlib.org/opt/dqed.f`.

[Härtig et al., 2005] Härtig, H., Hohmuth, M., Feske, N., Helmuth, C., Lackorzynski, A., Mehnert, F., and Peter, M. (2005). The Nizza secure-system architecture. In *Proceedings the 1st International Conference on Collaborative Computing: Networking, Applications and Worksharing*, San Jose, CA, USA.

[Härtig et al., 1997] Härtig, H., Hohmuth, M., Liedtke, J., and Schönberg, S. (1997). The performance of $\mu$-kernel based systems. In *Proceedings of the 16th Symposium on Operating System Principles*, pages 66–77, Saint Malo, France.

[Heath et al., 2006] Heath, T., Centeno, A. P., George, P., Ramos, L., Jaluria, Y., and Bianchini, R. (2006). Mercury and Freon: temperature emulation and management in server systems. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 106–116, San Jose, CA, USA.

[Heath et al., 2002] Heath, T., Pinheiro, E., Hom, J., Kremer, U., and Bianchini, R. (2002). Application transformations for energy and performance-aware device management. In *Proceedings of the 11th Conference on Parallel Architectures and Compilation Techniques*, pages 121–130, Charlottesville, VA, USA.

[Heiser, 2005] Heiser, G. (2005). Secure embedded systems need microkernels. *;login: the USENIX Association newsletter*, 30(6):9–13.

[Heiser, 2008] Heiser, G. (2008). The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, pages 11–16, Glasgow, Scotland.

[Heiser, 2009] Heiser, G. (2009). Hypervisors for consumer electronics. In *Proceedings of the 6th IEEE Consumer Communications and Networking Conference*, pages 614–618, Las Vegas, NV, USA.

[Heiser et al., 2007] Heiser, G., Elphinstone, K., Kuz, I., Klein, G., and Petters, S. M. (2007). Towards trustworthy computing systems: Taking microkernels to the next level. *ACM Operating Systems Review*, 41(4):3–11.

[Heiser et al., 2006] Heiser, G., Uhlig, V., and LeVasseur, J. (2006). Are virtual-machine monitors microkernels done right? *ACM Operating Systems Review*, 40(1):95–96.

[Herder et al., 2006] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. (2006). MINIX 3: a highly reliable, self-repairing operating system. *ACM Operating Systems Review*, 40(3):80–89.

[Hewlett-Packard Development Company, 2008] Hewlett-Packard Development Company (2008). Dynamic Power Capping TCO and Best Practices. White Paper.

[Hildebrand, 1992] Hildebrand, D. (1992). An architectural overview of QNX. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, USA.

[Huang et al., 2003] Huang, H., Pillai, P., and Shin, K. G. (2003). Design and implementation of power-aware virtual memory. In *Proceedings of the USENIX 2003 Annual Technical Conference*, pages 57–70, San Antonio, TX, USA.

[Hunt and Larus, 2007] Hunt, G. C. and Larus, J. R. (2007). Singularity: rethinking the software stack. *ACM Operating Systems Review*, 41(2):37–49.

[Intel, 2010] Intel (1999-2010). *Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*. Intel Corporation.

[Isci et al., 2006] Isci, C., Buyuktosunoglu, A., Cher, C.-Y., Bose, P., and Martonosi, M. (2006). An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 347–358, Washington, DC, USA.

[Jaeger et al., 1999] Jaeger, T., Elphinstone, K., Liedtke, J., Panteleenko, V., and Park, Y. (1999). Flexible access control using IPC redirection. In *Proceedings of 7th Workshop on Hot Topics in Operating Systems*, pages 191–196, Rio Rico, AZ, USA.

[Jones et al., 1997] Jones, M. B., Roşu, D., and Roşu, M.-C. (1997). CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the 16th Symposium on Operating System Principles*, pages 198–211, New York, NY, USA.

[Joseph and Martonosi, 2001] Joseph, R. and Martonosi, M. (2001). Run-time power estimation in high performance microprocessors. In *Proceedings of the*

*2001 International Symposium on Low Power Electronics and Design*, pages 135–140, Huntington Beach, CA, USA.

[Kansal and Zhao, 2008] Kansal, A. and Zhao, F. (2008). Fine-grained energy profiling for power-aware application design. In *Proceedings of the 1st Workshop on Hot Topics in Measurement and Modeling of Computer Systems*, pages 26–31, Annapolis, MD, USA.

[Kellner, 2003] Kellner, S. (2003). Event-driver temperature control in operating systems. Undergraduate thesis, Department of Computer Science, Distributed and Operating Systems Group, University of Erlangen-Nürnberg, Germany.

[Kerby, 2007] Kerby, B. (2007). *Managing Data Center Power and Cooling*. Dell, Inc.

[Kivity et al., 2007] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. (2007). kvm: the Linux virtual machine monitor. In *Proceedings of the Ottawa Linux Symposium 2007*, pages 225–230, Ottawa, Canada.

[Klein et al., 2009] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. (2009). seL4: Formal verification of an OS kernel. In *Proceedings of the 22th Symposium on Operating System Principles*, pages 207–220, Big Sky, MT, USA.

[Krieger et al., 2006] Krieger, O., Auslander, M., Rosenburg, B., Wisniewski, R. W., Xenidis, J., Silva, D. D., Ostrowski, M., Appavoo, J., Butrico, M., Mergen, M., Waterland, A., and Uhlig, V. (2006). K42: Building a complete operating system. In *Proceedings of the 1st ACM SIGOPS EuroSys conference*, pages 133–145, Leuven, Belgium.

[Kumar et al., 2003] Kumar, R., Farkas, K. I., Jouppi, N. P., Ranganathan, P., and Tullsen, D. M. (2003). Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, San Diego, CA, USA.

[L4 Development Team, 2009a] L4 Development Team (2009a). *L4 X.2 Reference Manual*. University of Karlsruhe, Germany.

[L4 Development Team, 2009b] L4 Development Team (2009b). L4Ka::Pistachio. `http://l4ka.org/projects/pistachio`.

[Lawitzky et al., 2008] Lawitzky, M. P., Snowdon, D. C., and Petters, S. M. (2008). Integrating real time and power management in a real system. In *Proceedings of the 4th Workshop on Operating System Platforms for Embedded Real-Time Applications*, pages 35–44, Prague, Czech Republic.

[Lebeck et al., 2000] Lebeck, A., Fan, X., Zeng, H., and Ellis, C. (2000). Power-aware page allocation. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–116, Cambridge, MA, USA.

[Lefurgy et al., 2003] Lefurgy, C., Rajamani, K., III, F. L. R., Felter, W. M., Kistler, M., and Keller, T. W. (2003). Energy management for commercial servers. *IEEE Computer*, 36(12):39–48.

[Leslie et al., 1996] Leslie, I. M., McAuley, D., Black, R., Roscoe, T., Barham, P. T., Evers, D., Fairbairns, R., and Hyden, E. (1996). The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):184–197.

[LeVasseur, 2009] LeVasseur, J. (2009). *Device-driver reuse via virtual machines*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia.

[LeVasseur et al., 2004] LeVasseur, J., Uhlig, V., Stoess, J., and Götz, S. (2004). Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 17–30, San Fransisco, CA, USA.

[LeVasseur et al., 2008] LeVasseur, J., Uhlig, V., Yang, Y., Chapman, M., Chubb, P., Leslie, B., and Heiser, G. (2008). Pre-virtualization: soft layering for virtual machines. In *Proceedings of the 13th Asia-Pacific Computer Systems Architecture Conference*, pages 1–9, Hsinchu, Taiwan.

[Levin et al., 1975] Levin, R., Cohen, E., Corwin, W., Pollack, F., and Wulf, W. (1975). Policy/mechanism separation in HYDRA. In *Proceedings of the 5th Symposium on Operating System Principles*, pages 132–140, Austin, TX, USA.

[Levine and Roth, 1997] Levine, F. E. and Roth, C. P. (1997). A programmer's view of performance monitoring in the PowerPC microprocessor. *IBM Journal of Research and Development*, 41(3):345–356.

[Li et al., 1994] Li, K., Kumpf, R., Horton, P., and Anderson, T. (1994). A quantitative analysis of disk drive power management in portable computers. In

*Proceedings of the USENIX Winter 1994 Technical Conference*, pages 279–292, San Francisco, CA, USA.

[Liedtke, 1992] Liedtke, J. (1992). Clans & Chiefs. In *Proceedings of the 12th GI/ITG Fachtagung Architektur von Rechensystemen*, pages 294–305, Kiel, Germany.

[Liedtke, 1993] Liedtke, J. (1993). Improving IPC by kernel design. In *Proceedings of the 14th Symposium on Operating System Principles*, pages 175–188, Asheville, NC, USA.

[Liedtke, 1995] Liedtke, J. (1995). On $\mu$-Kernel construction. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 237–250, Copper Mountain, CO, USA.

[Liedtke et al., 1991] Liedtke, J., Bartling, U., Beyer, U., Heinrichs, D., Ruland, R., and Szalay, G. (1991). Two years of experience with a $\mu$-kernel based OS. *ACM SIGOPS Operating Systems Review*, 25(2):51–62.

[Lu et al., 2000] Lu, Y.-H., Chung, E.-Y., Simunic, T., Benini, L., and Micheli, G. D. (2000). Quantitative comparison of power management algorithms. In *Proceedings of the Conference on Design Automation and Test in Europe*, pages 20–26, Paris, France.

[Lu and Micheli, 2001] Lu, Y.-H. and Micheli, G. D. (2001). Comparing system-level power management policies. *IEEE Design & Test of Computers*, 18(2):10–19.

[Malone and Belady, 2006] Malone, C. and Belady, C. (2006). Metrics to characterize data center & IT equipment energy use. In *Digital Power Forum*, Richardson, TX, USA.

[Mandagere et al., 2007] Mandagere, N., Diehl, J., and Du, D. H.-C. (2007). Greenstor: Application-aided energy-efficient storage. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 16–29, San Diego, CA, USA.

[Maren, 1999] Maren, K. T. V. (1999). The Fluke device driver framework. Master's thesis, Department of Computer Science, University of Utah, Salt Lake City, UT.

[Marsh et al., 1991] Marsh, B. D., Scott, M. L., LeBlanc, T. J., and Markatos, E. P. (1991). First-class user-level threads. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 110–21, Pacific Grove, CA, USA.

[Merkel and Bellosa, 2006] Merkel, A. and Bellosa, F. (2006). Balancing power consumption in multiprocessor systems. In *Proceedings of the 1st ACM SIGOPS EuroSys conference*, pages 403–414, Leuven, Belgium.

[Microsoft Corporation, 2009] Microsoft Corporation (2009). Hyper-V. `http://www.microsoft.com/hyper-v-server/`.

[Microsoft Corporation and Intel Corporation, 2009] Microsoft Corporation and Intel Corporation (2009). *A Dynamic Approach to Power Budgeting*. Microsoft Corporation, Intel Corporation.

[Moore et al., 2005] Moore, J. D., Chase, J. S., Ranganathan, P., and Sharma, R. K. (2005). Making scheduling ”cool”: Temperature-aware workload placement in data centers. In *Proceedings of the USENIX 2005 Annual Technical Conference*, pages 61–75, Anaheim, CA, USA.

[Mudge, 2001] Mudge, T. (2001). Power: A first-class architectural design constraint. *IEEE Computer*, 34(4):52–58.

[Narayanan et al., 2008] Narayanan, D., Donnelly, A., and Rowstron, A. I. T. (2008). Write off-loading: Practical power management for enterprise storage. In *Proceedings of the 6th Conference on File and Storage Technologies*, pages 253–267, San Jose, CA, USA.

[Narayanan et al., 2009] Narayanan, D., Thereska, E., Donnelly, A., Elnikety, S., and Rowstron, A. (2009). Migrating server storage to SSDs: analysis of tradeoffs. In *Proceedings of the 4th ACM SIGOPS EuroSys conference*, pages 145–158, Nürnberg, Germany.

[Nathuji, 2008] Nathuji, R. (2008). *Mechanisms for coordinated power management with application to cooperative distributed systems*. PhD thesis, School of Computer Science and Engineering, Georgia Institute of Technology, GA, USA.

[Nathuji et al., 2009] Nathuji, R., England, P., Sharma, P., and Singh, A. (2009). Feedback driven QoS-aware power budgeting for virtualized servers. In *In Proceedings of the International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks*, pages 13–28, Paris, France.

[Nathuji and Schwan, 2007] Nathuji, R. and Schwan, K. (2007). VirtualPower: Coordinated power management in virtualized enterprise systems. In *Proceedings of the 21th Symposium on Operating System Principles*, pages 265–278, Stevenson, WA, USA.

[Nathuji and Schwan, 2008] Nathuji, R. and Schwan, K. (2008). VPM tokens: virtual machine-aware power budgeting in datacenters. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 119–128, Boston, MA, USA.

[Nathuji et al., 2008] Nathuji, R., Somani, A., Schwan, K., and Yogendra, J. (2008). CoolIT: Coordinating facility and IT management for efficient datacenters. In *Proceedings of the 1st Workshhp on Power-Aware Computing and Systems*, San Diego, CA, USA.

[National Academy of Sciences of Ukraine, Cybernetics Institute, 2009] National Academy of Sciences of Ukraine, Cybernetics Institute (2009). Openopt Framework. `http://openopt.org`.

[Naveh et al., 2006] Naveh, A., Rotem, E., Mendelson, A., Gochman, S., Chabukswar, R., Krishnan, K., and Kumar, A. (2006). Power and thermal management in the Intel Core Duo processor. *Intel Technology Journal*, 10(2):109–122.

[Neugebauer and McAuley, 2001] Neugebauer, R. and McAuley, D. (2001). Energy is just another resource: energy accounting and energy pricing in the nemesis OS. In *Proceedings of 8th Workshop on Hot Topics in Operating Systems*, pages 67–72, Schloß Elmau, Oberbayern, Germany.

[Ng and Chen, 1999] Ng, W. T. and Chen, P. M. (1999). The systematic improvement of fault tolerance in the rio file cache. In *Proceedings of the 29th IEEE Annual International Symposium on Fault-Tolerant Computing*, pages 76–83, Washington, DC, USA.

[Open Kernel Labs, 2009] Open Kernel Labs (2009). OKL4. `http://www.oklabs.com`.

[Papathanasiou and Scott, 2004] Papathanasiou, A. E. and Scott, M. L. (2004). Energy efficient prefetching and caching. In *Proceedings of the USENIX 2004 Annual Technical Conference*, pages 255–268, Boston, MA, USA.

[QNX Corporation, 2009] QNX Corporation (2009). QNX power management. `http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/sys_arch/power.html`.

[Raghavendra et al., 2008] Raghavendra, R., Ranganathan, P., Talwar, V., Wang, Z., and Zhu, X. (2008). No "power" struggles: coordinated multi-level power management for the data center. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 48–59, Seattle, WA, USA.

[Ranganathan et al., 2006] Ranganathan, P., Leech, P., Irwin, D. E., and Chase, J. S. (2006). Ensemble-level power management for dense blade servers. In *Proceedings of the 33th Annual International Symposium on Computer Architecture*, pages 66–77, Boston, MA, USA.

[Rao, 1996] Rao, S. S. (1996). *Engineering Optimization: Theory and Practice.* John Wiley & Sons Inc.

[Rasmussen, 2006] Rasmussen, N. (2006). Cooling strategies for ultra-high density racks and blade servers. White Paper, American Power Conversion Corporation.

[Raymond, 2004] Raymond, E. S. (2004). *The Art of UNIX Programming*. Addison-Wesley.

[Rohou and Smith, 1999] Rohou, E. and Smith, M. D. (1999). Dynamically managing processor temperature and power. In *Proceedings of the 2nd Workshop on Feedback-Directed Optimization*.

[Rosenblum and Garfinkel, 2005] Rosenblum, M. and Garfinkel, T. (2005). Virtual machine monitors: Current technology and future trends. *IEEE Computer*, 38(5):39–47.

[Sachs et al., 2004] Sachs, D. G., Yuan, W., Hughes, C. J., Harris, A., Adve, S. V., Jones, D. L., Kravets, R. H., and Nahrstedt, K. (2004). Grace: A hierarchical adaptation framework for saving energy. Technical Report UIUCDCS-R-2004-2409, Department of Computer Science, University of Illinois, Urbana-Champaign, IL, USA.

[Sales, 2005] Sales, J. (2005). *Symbian OS Internals: Real-time kernel programming*. John Wiley & Sons Inc.

[Schulz, 2007] Schulz, G. (2007). Storage power and cooling issues heat up. Online Article, enterprisestorageforum.com.

[Seltzer et al., 1996] Seltzer, M. I., Endo, Y., Small, S., and Smith, K. A. (1996). Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, Berkeley, CA, USA.

[Skadron et al., 2003a] Skadron, K., Stan, M. R., Huang, W., Velusamy, S., Sankaranarayanan, K., and Tarjan, D. (2003a). Temperature-aware computer systems: Opportunities and challenges. *IEEE Micro*, 23(6):52–61.

[Skadron et al., 2003b] Skadron, K., Stan, M. R., Huang, W., Velusamy, S., Sankaranarayanan, K., and Tarjan, D. (2003b). Temperature-aware microarchitecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 2–13, New York, NY, USA.

[Snowdon, 2009] Snowdon, D. (2009). *Operating System Directed Power Management.* PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia.

[Snowdon et al., 2007] Snowdon, D. C., Petters, S. M., and Heiser, G. (2007). Accurate on-line prediction of processor and memory energy usage under voltage scaling. In *Proceedings of the 7th ACM International Conference on Embedded Software*, pages 84–93, Salzburg, Austria.

[Snowdon et al., 2009] Snowdon, D. C., Sueur, E. L., Petters, S. M., and Heiser, G. (2009). Koala: A platform for OS-level power management. In *Proceedings of the 4th ACM SIGOPS EuroSys conference*, pages 289–302, Nürnberg, Germany.

[Stemm and Katz, 1997] Stemm, M. and Katz, R. H. (1997). Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Transactions on Communications*, E80-B(8):1125–31.

[Stoess, 2002] Stoess, J. (2002). I/O-FlexPages on the x86-architecture. Undergraduate thesis, System Architecture Group, University of Karlsruhe, Germany.

[Stoess and Uhlig, 2006] Stoess, J. and Uhlig, V. (2006). Flexible, low-overhead event logging to support resource scheduling. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 115–120, Minneapolis, MN, USA.

[Sugerman et al., 2001] Sugerman, J., Venkitachalam, G., and Lim, B.-H. (2001). Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the USENIX 2001 Annual Technical Conference*, pages 1–14, Boston, MA, USA.

[Sullivan, 2000] Sullivan, R. F. (2000). Alternating cold and hot aisles provides more reliable cooling for server farms. White Paper, The Uptime Institute.

[Sun Microsystems Corporation, 2009] Sun Microsystems Corporation (2009). VirtualBox. `http://www.virtualbox.org`.

[Swift et al., 2003] Swift, M. M., Bershad, B. N., and Levy, H. M. (2003). Improving the reliability of commodity operating systems. In *Proceedings of the 19th Symposium on Operating System Principles*, pages 77–110, Bolton Landing, NY, USA.

[Tanenbaum et al., 2006] Tanenbaum, A. S., Herder, J. N., and Bos, H. (2006). Can We Make Operating Systems Reliable and Secure? *IEEE Computer*, 39(5):44–51.

[Tolia et al., 2009] Tolia, N., Wang, Z., Ranganathan, P., Bash, C., and Marwah, M. (2009). Unified thermal and power management in server enclosures. In *Proceedings of the ASME/Pacific Rim Technical Conference and Exhibition on Packaging and Integration of Electronic and Photonic Systems, MEMS, and NEMS*, San Francisco, CA, USA.

[Tucker and Gupta, 1989] Tucker, A. and Gupta, A. (1989). Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th Symposium on Operating System Principles*, pages 159–166, Litchfield Park, AZ, USA.

[Uhlig, 2005] Uhlig, V. (2005). *Scalability of Microkernel-Based Systems*. PhD thesis, System Architecture Group, University of Karlsruhe, Germany.

[Uhlig et al., 2004] Uhlig, V., LeVasseur, J., Skoglund, E., and Dannowski, U. (2004). Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 43–56, San Jose, CA, USA.

[Verma et al., 2008] Verma, A., Ahuja, P., and Neogi, A. (2008). pMapper: Power and migration cost aware application placement in virtualized systems. In *Proceedings of the ACM/IFIP/USENIX 9th International Middleware Conference*, pages 243–264, Leuven, Belgium.

[Verma et al., 2009] Verma, A., Dasgupta, G., Nayak, T. K., De, P., and Kothari, R. (2009). Server workload analysis for power minimization using consolidation. In *Proceedings of the USENIX 2009 Annual Technical Conference*, pages 39–48, San Diego, CA, USA.

[VMware Inc., 2007] VMware Inc. (2007). VMware Distributed Power Management concepts and use. White Paper, VMware Inc.

[VMware Inc., 2009a] VMware Inc. (2009a). Vmware ESX. `http://www.vmware.com/products/esx`.

[VMware Inc., 2009b] VMware Inc. (2009b). VMware Mobile Virtualization Platform. `http://www.vmware.com/products/mobile`.

[VMware Inc., 2009c] VMware Inc. (2009c). VMware Workstation. `http://www.vmware.com/products/workstation/`.

[Waldspurger and Weihl, 1994] Waldspurger, C. A. and Weihl, W. E. (1994). Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 1–11, Monterey, CA, USA.

[Wang and Wang, 2009] Wang, X. and Wang, Y. (2009). Co-con: Coordinated control of power and application performance for virtualized server clusters. In *Proceedings of the 17th IEEE International Workshop on Quality of Service*, pages 1–9, Charleston, SC, USA.

[Weiser et al., 1994] Weiser, M., Welch, B. B., Demers, A. J., and Shenker, S. (1994). Scheduling for reduced CPU energy. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 13–23, Monterey, CA, USA.

[Weissel and Bellosa, 2002] Weissel, A. and Bellosa, F. (2002). Process cruise control – event-driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 238–246, Grenoble, France.

[Weissel and Bellosa, 2004] Weissel, A. and Bellosa, F. (2004). Dynamic thermal management for distributed systems. In *Proceedings of the 1st Workshop on Temperature-Aware Computer Systems*, München, Germany.

[Weissel et al., 2002] Weissel, A., Beutel, B., and Bellosa, F. (2002). Cooperative IO - a novel IO semantics for energy-aware applications. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 117–130, Boston, MA, USA.

[Welch, 1995] Welch, G. F. (1995). A survey of power management techniques in mobile computing operating systems. *ACM Operating Systems Review*, 29(4):47–56.

[Wichmann, 1968] Wichmann, B. (1968). A modular operating system. In *In Proceedings of Information Processing 68, North-Holland Publishing Company*, pages 548–556, Amsterdam, Netherlands.

[Xen.org, 2009] Xen.org (2009). Xen credit-based CPU scheduler. `http://wiki.xensource.com/xenwiki/CreditScheduler`.

[Zedlewski et al., 2003] Zedlewski, J., Sobti, S., Garg, N., Zheng, F., Krishnamurthy, A., and Wang, R. (2003). Modeling hard-disk power consumption. In *Proceedings of the 2nd Conference on File and Storage Technologies*, pages 217–230, San Francisco, CA, USA.

[Zeng et al., 2005] Zeng, H., Ellis, C. S., and Lebeck, A. R. (2005). Experiences in managing energy with ECOSystem. *IEEE Pervasive Computing*, 4(1):62–68.

[Zeng et al., 2002] Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A. (2002). ECOSystem: managing energy as a first class operating system resource. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–132, San Jose, CA, USA.

[Zeng et al., 2003] Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A. (2003). Currentcy: unifying policies for resource management. In *Proceedings of the USENIX 2003 Annual Technical Conference*, pages 43–56, San Antonio, TX, USA.

[Zhu et al., 2005] Zhu, Q., Chen, Z., Tan, L., Zhou, Y., Keeton, K., and Wilkes, J. (2005). Hibernator: Helping disk arrays sleep through the winter. In *Proceedings of the 20th Symposium on Operating System Principles*, pages 177–190, Brighton, UK, USA.

[Zhu et al., 2004] Zhu, Q., David, F. M., Devaraj, C. F., Li, Z., Zhou, Y., and Cao, P. (2004). Reducing energy consumption of disk storage using power-aware cache management. In *Proceedings of the 10th International Conference on High-Performance Computer Architecture*, pages 118–129, Madrid, Spain.