Energiesparende Programmierung

Sharon Friedrich

Seminar: Grüne Informationstechnik
Prof. Dr. Thomas Ludwig

30. Juni 2008

Gliederung

1. Einleitung
2. Eingebettete Systeme
3. Ansätze für energiesparende Programmierung
   1. Compileroptimierung
   2. Software Washing Machine
   3. Energiesparende Kryptographie und Datenkomprimierung
   4. Ausnutzen von Speicherhierarchien
   5. Minimalistische Software
   6. Software Engineering
4. Zusammenfassung und Ausblick

1. Einleitung

Mooresche Gesetz

- Hardware immer schneller
- Exponentielles Wachstum

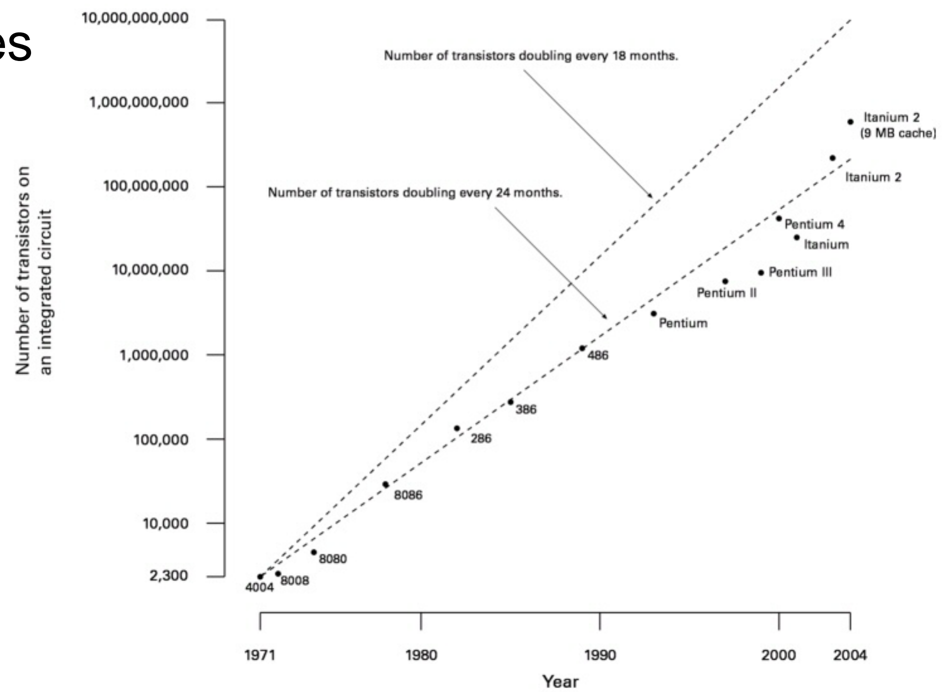
Wirthsches Gesetz

- „Software gets slower – faster than hardware gets faster“

Schneller Fortschritt und Verbesserung in der Hardware (exponentiell, Mooresches Gesetz)
Software wird immer komplexer → immer höhere Anforderungen an Software der Kunden → langsamer (Wirthsches Gesetz [1])
→Obwohl Hardware immer schneller wird, keine schnellere Abarbeitung von Aufgaben, da durch langsamere Software Geschwindigkeitsvorteil aufgehoben
→Software wird schneller langsamer als Hardware schneller wird

1. Einleitung

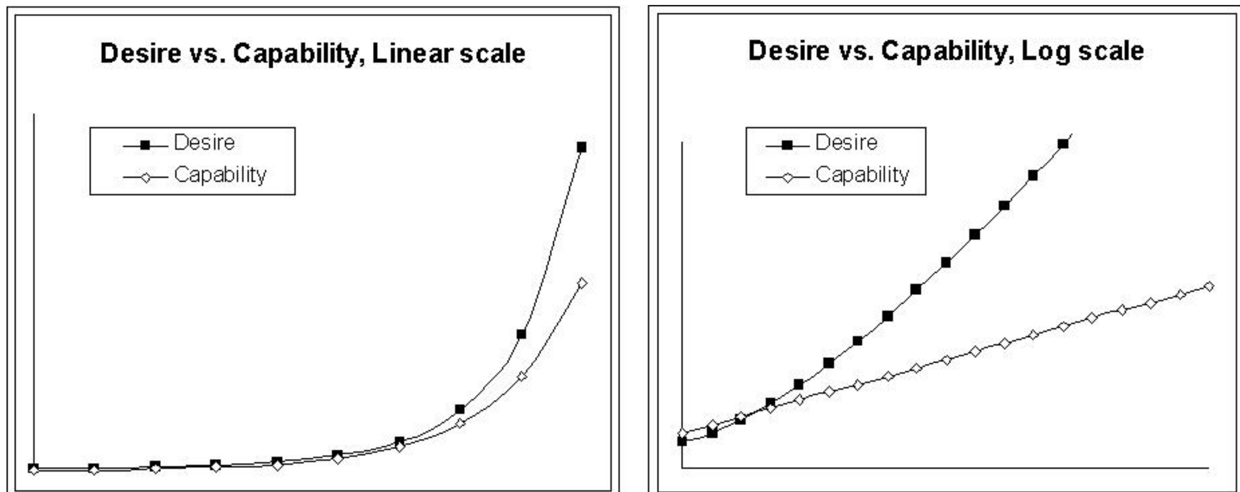
Mooresches Gesetz



Beispiel Prozessor für wachsenden Technikfortschritt

1. Einleitung

Wirthsches Gesetz



Grafik zeigt die Wünsche der Kunden, was die Software erfüllen muss und zum Vergleich, was die Hardware erfüllen kann → soll verdeutlichen, dass Software schneller komplexer wird, als Hardware sich verbessert [2]

1. Einleitung

- Beispiel:
  - 1970: Speicherbedarf eines Texteditors 8000 Bytes
  - Heute: mindestens das 1000-fache

- Ineffizienz von Programmen → erhöhter Energieverbrauch
  - z. B. lange Laufzeiten, hoher Speicherverbrauch

Beispiel aus [2]

1. Einleitung

- Ziel: Energieverbrauch von Programmen reduzieren
  - Wichtig für Systeme mit beschränkten Energieressourcen
    - z. B. mobile Geräte
  - Optimierung des Programmcodes
    - Implementierung anfangs oft ineffizient
      - z. B. automatische Code-Generierung

Gliederung

1. Einleitung
2. Eingebettete Systeme
3. Ansätze für energiesparende Programmierung
   1. Compileroptimierung
   2. Software Washing Machine
   3. Energiesparende Kryptographie und Datenkomprimierung
   4. Ausnutzen von Speicherhierarchien
   5. Minimalistische Software
   6. Software Engineering
4. Zusammenfassung und Ausblick

2. Eingebettete Systeme

- Hard- und Softwaresysteme, die in einem größeren System integriert sind
- Aufgabe: Steuerung, Regelung, Überwachung
- Beispiel: Antiblockiersystem im Auto, mobile Systeme wie Handys

[3,4]

2. Eingebettete Systeme

- Eigenschaften:
  - Zweckbestimmtheit
  - Verlässlichkeit
  - Echtzeitanwendung
  - Transparenz für den Benutzer
  - Häufig ohne Festplatte, Betriebssystem, Tastatur oder Maus
- „Power is considered as the most important constraint in embedded systems“

  L. Eggermont
- Primitive Hardware (Prozessor, Speicher)

Zweckbestimmtheit:
→für einen bestimmten Zweck entworfen
→z. B. Prozessor für bestimmte Software → nicht vorgesehen, andere Software wie z. B. Spiele zu unterstützen
→Grund: mehr Verlässlichkeit, Ressource vollständig verbraucht

Verlässlichkeit:
→ Meisten eingebetteten Systeme sicherheitskritisch (z. B. Autos)
→ Benutzer muss sich drauf verlassen lassen können, dass das System nicht versagt
→ System muss wartbar sein
→ System muss stets verfügbar sein
→ ein versagendes System darf nicht schaden
→ vertrauliche Daten müssen vertraulich bleiben

Transparenz:
→ Benutzer ist Anwesenheit eines eingebetteten Systems oftmals nicht bewusst, da auch UI zweckbestimmt ist (z. B. Pedale im Auto)

Effizienz: Energie, Kosten, Gewicht, Echtzeit, Code-Größe
→ Energie
→Kosten: Einsatz primitiver Hardware → z. B. langsamerer Prozessor, keine Festplatte → auch weniger Energieverbrauch

[3,4]

Gliederung

1. Einleitung
2. Eingebettete Systeme
3. Ansätze für energiesparende Programmierung
   1. Compileroptimierung
   2. Software Washing Machine
   3. Energiesparende Kryptographie und Datenkomprimierung
   4. Ausnutzen von Speicherhierarchien
   5. Minimalistische Software
   6. Software Engineering
4. Zusammenfassung und Ausblick

3.1 Compileroptimierung

- Ziel: Verbesserung der Laufzeit des Programms / Minimierung des Speicherbedarfs
- Abhängig von verfügbarer Hardware
- Teilweise sehr aufwändig

Compileroptimierung: schon vielfach eingesetzt, ABER: bringen wenig Energieeffizienz [3,4,5]

3.1 Compileroptimierung

Optimierungsmöglichkeiten

- Einsparung von Maschinenbefehlen
- Entfernung von totem Code
- Entfernung von unbenutzten Variablen
- Schleifenoptimierung
- Inlining

3.1 Compileroptimierung

```
int a[1000];
c = a;
for (i = 1; i < 100; i++) {
 b += *c;
 b += *(c+7);
 c += 1;
}
```

```
LDR r3, [r2, #0]
ADD r3,r0,r3
MOV r0,#28
LDR r0, [r2, r0]
ADD r0,r3,r0
ADD r2,r2,#4
ADD r1,r1,#1
CMP r1,#100
BLT LL3
```

2096 cycles
19.92 µJ

```
ADD r3,r0,r2
MOV r0,#28
MOV r2,r12
MOV r12,r11
MOV r11,rr10
MOV r0,r9
MOV r9,r8
MOV r8,r1
LDR r1, [r4, r0]
ADD r0,r3,r1
ADD r4,r4,#4
ADD r5,r5,#1
CMP r5,#100
BLT LL3
```

2231 cycles
16.47 µJ

Erhöhung der Performanz bedeutet nicht Reduzierung des Energieverbrauchs (siehe Abbildung)
→ Reduzierung des Energieverbrauchs durch Ausnutzung von Registern [5]

3.1 Compileroptimierung

- Energieverbrauch als Kostenfunktion in Compilern
  - Energy-aware Instruction Scheduling
  - Energy-aware Instruction Selection

	Intel 486DX2				Fujitsu SPARClite '934			
No.	Instruction	Current (mA)	Cycles	Energy ($10^{-8}J$)	Instruction	Current (mA)	Cycles	Energy ($10^{-8}J$)
1	nop	276	1	2.27	nop	198	1	3.26
2	mov dx,[bx]	428	1	3.53	ld [%10],%i0	213	1	3.51
3	mov dx,bx	302	1	2.49	or %g0,%i0,%10	198	1	3.26
4	mov [bx],dx	522	1	4.30	st %i0,[%10]	346	2	11.4
5	add dx,bx	314	1	2.59	add %i0,%o0,%10	199	1	3.28
6	add dx,[bx]	400	2	6.60	mul %g0,%r29,%r27	198	1	3.26
7	jmp	373	3	9.23	srl %i0,1,%10	197	1	3.25

[3, 4, 5, 6]
2 Konzepte zur Reduzierung des Energieverbrauchs durch Compiler:
- Energy-aware Instruction Scheduling
-Energy-aware Instruction Selection

→ Tabelle [6] zeigt, dass die einzelnen Assemblerbefehle einen unterschiedlichen Energieverbrauch haben und deshalb die oberen Konzepte möglich sind → allerdings eventuell veraltet, da von 1996

3.1 Compileroptimierung

Energy-aware Instruction Scheduling

- Veränderung der Reihenfolge von Maschinenbefehlen
- Liste mit Instruktion → gerichteter azyklischer Graph für Datenabhängigkeiten
- 2 Typen Energieverbrauch:
  - Ausführung der Instruktion
  - Ausführung nach einer anderen Instruktion
- Bis zu 30% Reduzierung des Energieverbrauchs

-Reihenfolge von Maschinenbefehlen kann in vielen Fällen verändert werden, ohne dass sich die Bedeutung des Programms ändert
→ Optimierungstechnik wird auf den durch den Compiler erzeugten Output angewendet
- Datenabhängigkeiten: Ausführungszeit, Wartezeiten
-z. B. Top-Down Traversal durch den Graph: diejenige Instruktion wird gewählt, die maximale Zeitverzögerung zum Ende des betrachteten Blocks hat
-Als Kostenfunktion kann durchschnittlicher Energieverbrauch bei Ausführung der betrachteten Instruktion oder durchschnittlicher Energieverbrauch bei Ausführung nach einer bestimmten anderen Instruktion (abhängig von Switching Aktivitäten) gewählt werden

[3,4, 11]

3.1 Compileroptimierung

Energy-aware Instruction Selection

- Analyse von äquivalenten Befehlssequenzen
- Wahl der besten Befehlssequenz

Profiler

- Softwaretool
  - Analyse von Software zur Laufzeit
  - Messung von Eigenschaften
  - Identifikation von zu optimierenden Programmabschnitten

[3,4,5]
Existieren mehrere Befehlssequenzen, die die gleiche Aufgabe haben und zum selben Ziel führen, d. h. äquivalent sind → diejenige wählen mit geringstem Energieverbrauch

Wie kann Energieverbrauch ermittelt werden?
→Softwaretool Profiler
→Eigenschaften: z. B. Anzahl der Funktionsaufrufe, Energieverbrauch, Laufzeitverhalten

Gliederung

1. Einleitung
2. Eingebettete Systeme
3. Ansätze für energiesparende Programmierung
   1. Compileroptimierung
   2. Software Washing Machine
   3. Energiesparende Kryptographie und Datenkomprimierung
   4. Ausnutzen von Speicherhierarchien
   5. Minimalistische Software
   6. Software Engineering
4. Zusammenfassung und Ausblick

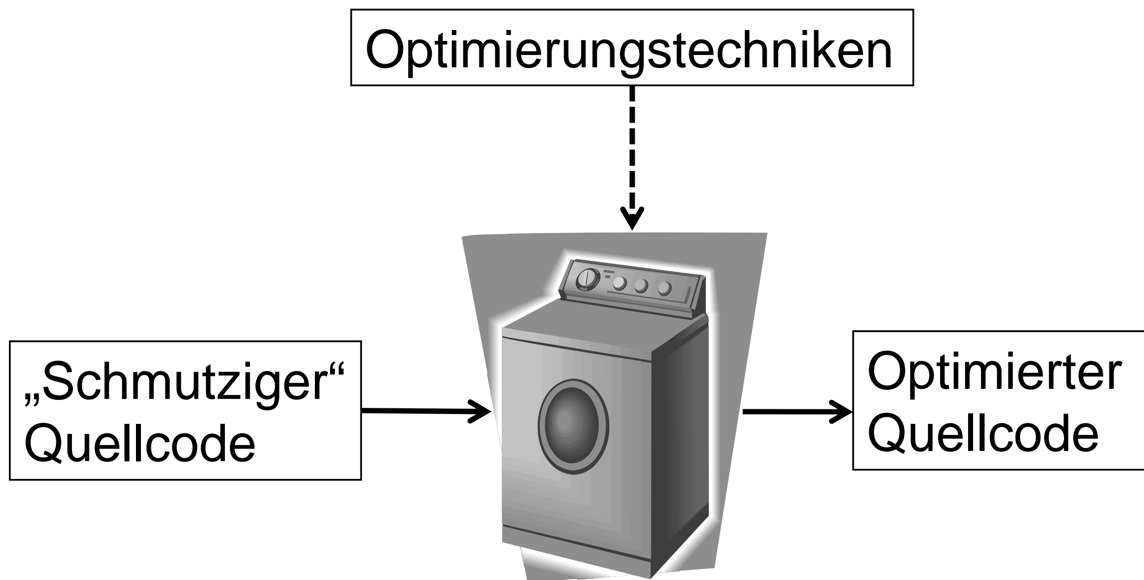
3.2 Software Washing Machine

Software Washing Machine

- Konzept zur Optimierung von ineffizienten Codeteilen
- Entwickelt am IMEC (Interuniversity Microelectronics Centre) im Jahr 2002
  - Speziell für eingebettete Systeme
- Auf Quellcodeebene
- Ergänzt Compileroptimierungen

[4,7,8,9, 15]
Durch das Konzept der Software Washing Machine sollen Transformationen direkt am Quellcode vorgenommen werden, um den Energieverbrauch zu reduzieren

3.2 Software Washing Machine



Prinzip der Software Washing Machine: Schmutziger Code, d. h. ineffizienter Code wird mithilfe von Optimierungstechniken (siehe nächsten Folien) in optimierten/energieeffizienten Code umgewandelt

z. B. wurde Code automatisch aus Spezifikationen generiert oder wurde zur Lesbarkeit geschrieben → ineffizient

3.2 Software Washing Machine

Was ist „schmutziger“ Quellcode?

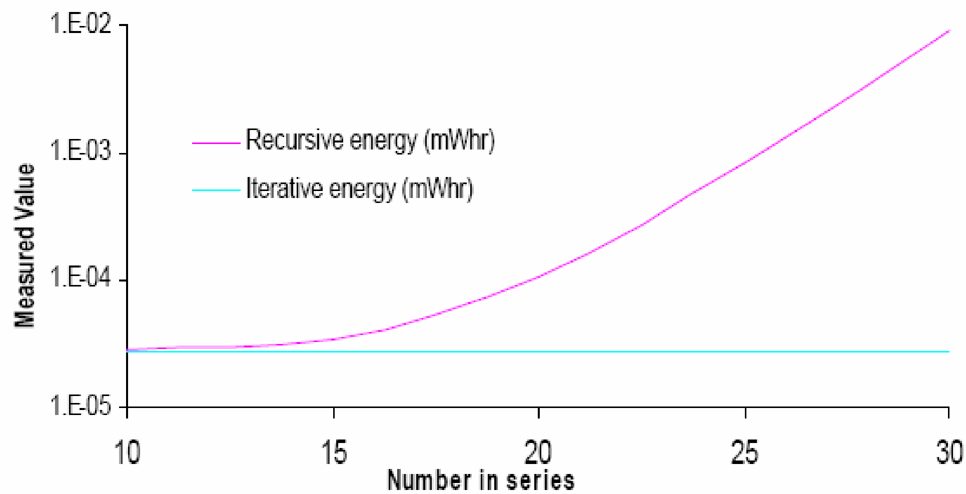
- Nicht optimiert → durch Compiler nicht ausreichend
- Sorgt für ineffiziente Programme
- Mögliche Ursachen:
  - Algorithmenwahl (z. B. rekursiv vs. iterativ)
  - Datentypwahl
  - Arrays (teure Zugriffe)
  - Schleifen + Verzweigungen

Optimierung durch Compiler zur Erreichung von Energieeffizienz nicht ausreichend
Ineffizienter, nicht optimierter Quellcode resultiert in ineffizienten Programmen → hoher Energieverbrauch

Datentypwahl: z. B. Integer vs. Float
Zugriffe auf Elemente eines großen Arrays verursachen, wenn nicht optimiert, einen hohen Energieverbrauch

3.2 Software Washing Machine

- Berechnung der Fibonacci-Zahlen



[10]
Beispiel für Unterschied im Energieverbrauch bei der Wahl zwischen iterativem und rekursivem Algorithmus
→ Es gibt auch den umgekehrten Fall, dass der iterative Algorithmus mehr Energie benötigt, als der rekursive (z. B. Türme des Hanoi)

3.2 Software Washing Machine

High-Level Optimierungstechniken:

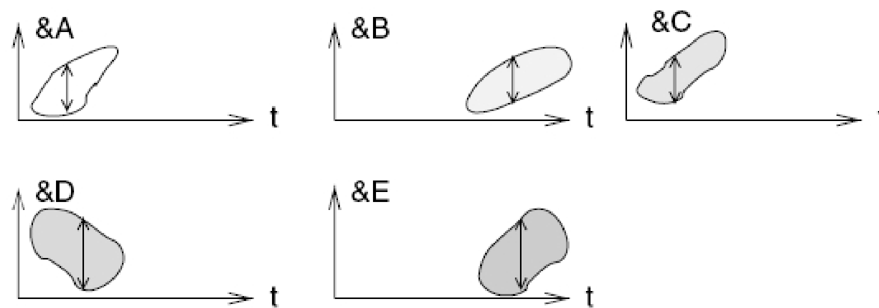
- Transformationen am Quellcode vor Kompilierung
- Integration in Compiler
- Methoden: Array Folding und Loop Splitting

[3,4,7]
Prinzip der Software Washing Machine: Einsatz von High-Level Optimierungstechniken, direkt auf den Quellcode angewandt
→Kann in den Compiler integriert werden, wobei sie vor der Kompilierung durchgeführt werden
→Beispiele für High-Level Optimierungstechniken: Array Folding und Loop Splitting (siehe nächsten Folien)

3.2 Software Washing Machine

1. Array Folding

- Oft nur kleiner Bereich eines Arrays zu einem Zugriffszeitpunkt relevant
  - Address Reference Window
    - Maximale Anzahl an relevanten Elementen



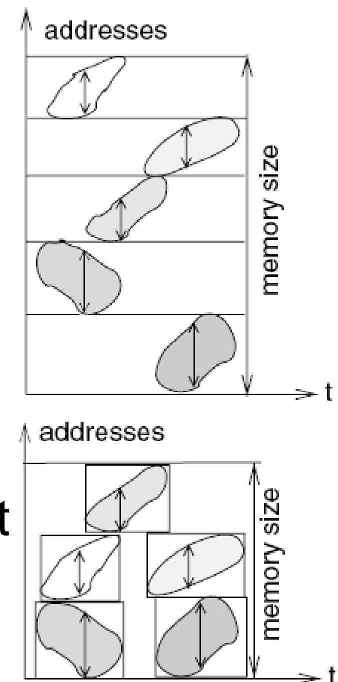
Hintergrund: zu einem bestimmten Zeitpunkt wird meist nur auf einen kleinen Bereich eines Arrays zugegriffen → nicht alle Elemente relevant
Address Reference Window: maximale Distanz zwischen den Adressen von zwei Arrayelementen zu einem Zugriffszeitpunkt (siehe Doppelpfeil in der Abbildung)

Abbildung zeigt das Address Reference Window verschiedener Arrays zu einem bestimmten Zeitpunkt t

[3, 4]

3.2 Software Washing Machine

- Typische Speicherallokierung
  - Maximaler Speicherplatz für gesamte Laufzeit
- Inter-Array Folding
  - Gleicher Speicherplatz bei nicht gleichzeitiger Ausführung (verschiedene Arrays)

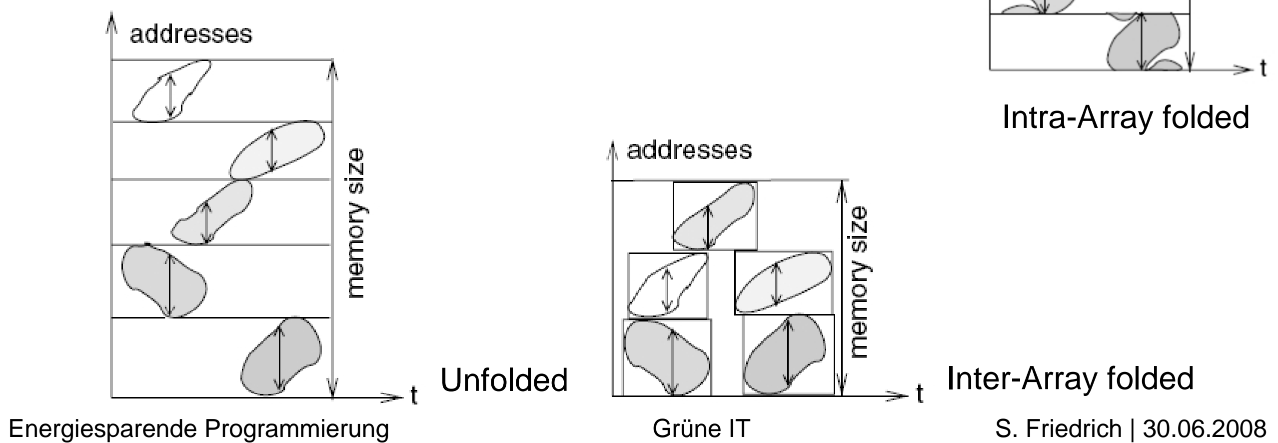


Hintergrund: Globale Arrays nehmen während der gesamten Laufzeit den maximal benötigten Speicherplatz ein (obere Abbildung) → unnötig, da nicht alle Elemente immer gleichzeitig benötigt werden → teure Zugriffe + unnötiger Speicherverbrauch → hoher Energieverbrauch → Optimierung durch Array Folding
Zwei Typen von Array Folding:
1.) Inter-Array Folding
2.) Intra-Array Folding

Zu 1.) Verschiedene Arrays, die nicht gleichzeitig benötigt werden, allokieren den gleichen Speicherplatz (untere Abbildung)
Zu 2.) Verschiedene Elemente eines Arrays, die nicht gleichzeitig benötigt werden, allokieren den gleichen Speicherplatz (siehe nächste Folie)

3.2 Software Washing Machine

- Intra-Array Folding
  - Gleicher Speicherplatz bei nicht gleichzeitiger Ausführung (Elemente eines Arrays)



→ z. B. MPEG-2 verwendet viel größere Arrays als benötigt

3.2 Software Washing Machine

2. Loop Splitting

- Hardwareschleifen (Zero-Overhead Loop)
  - Steuerung der Schleife durch Hardware
  - Keine Rechenzeit für Schleifenzähler
  - Keine Prozessorzyklen benötigt
  - Von einigen Prozessoren für Schleifen limitierter Größe angeboten

[3,4]
Hardwareschleifen haben geringeren Energieverbrauch als normale Schleifen (keine Rechenzeit, keine Prozessorzyklen)

3.2 Software Washing Machine

- Ausnutzen von Hardwareschleifen → Aufsplitten von großen Schleifen
- Parallelisierung einer Schleife
- Mit oder ohne Veränderung des Schleifenaufbaus
  - Spezialfall: Loop Nest Splitting

[12]
Loop Splitting:
→Mehrere Schleifen aus einer
→Verteilung der Anweisungen
→Ziel: Parallelisierung der Schleifen, da weniger Datenabhängigkeiten [12]

3.2 Software Washing Machine

Loop Nest Splitting

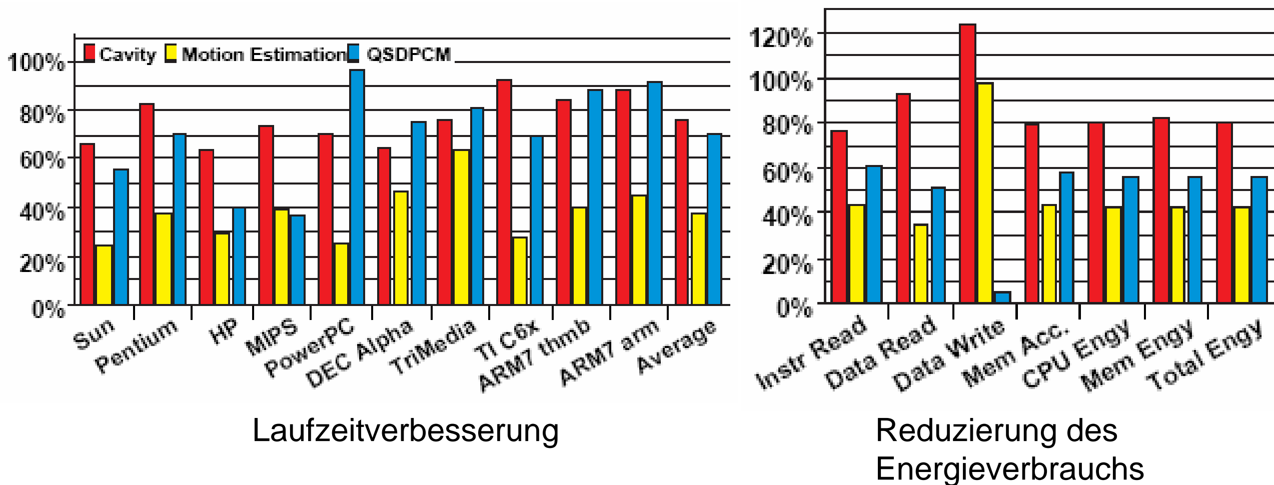
- Viele ineinander verschachtelte Schleifen
- Schleifen mit if-Anweisungen
  - Reduzierung der Anzahl der ausgeführten if-Anweisungen
- Beispiel: Bildverarbeitende Algorithmen

Viele if-Anweisungen zur Überprüfung der Pixel auf Randpixel	=	Keine Überprüfung, umfasst fast alle Pixel	+	Randüberprüfung, Wenige Pixel

[3, 4, 12, 19]
- Viele ineinander verschachtelte Schleifen mit if-Anweisungen durch Loop Nest Splitting optimieren
- if-Anweisungen in den Schleifen für Normal- und Sonderfälle → benötigen viele Instruktionen → hoher Energieverbrauch
- vor allem Multimediaanwendungen verarbeiten große Datenmengen mithilfe mehrfach verschachtelter Schleifen
- Reduzierung der Anzahl der ausgeführten if-Anweisungen
  → z. B. Iterationen bestimmen, in denen if-Anweisungen erfüllt
  → Schleifen so aufsplitten, dass für einen möglichst großen Teil der Iterationen if-Anweisungen entfallen
  → Aufdeckung von Bedingungen, die keinen Einfluss auf den Kontrollfluss haben → können entfernt werden
- Beispiel Bildverarbeitende Algorithmen:
  → verwenden Informationen der benachbarten Pixel des betrachteten Pixels zur Verbesserung der Qualität eines Bildes
  → Algorithmus für Randpixel so nicht möglich → modifizierte Berechnungen → if-Anweisungen in den Schleifen
  → Aufsplitten der Schleife → eine Schleife für den typischen Fall, der für fast alle Pixel gilt und eine Schleife, die den Sonderfall bearbeitet

Reduktion der Anzahl der ausgeführten if-Anweisungen: Bestimmung von Iterationen, in denen if-Anweisungen nachweislich erfüllt sind
Aufteilung des Iterationsraums der verschachtelten Schleife, so dass für möglichst großen Teil der Iterationen if-Anweisungen entfallen
Aufdeckung von Bedingungen, die keinen Einfluss auf Kontrollfluss haben → können entfernt werden

3.2 Software Washing Machine

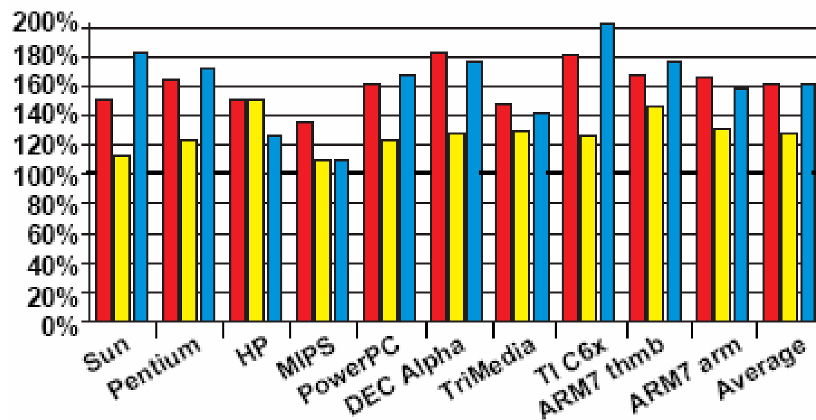


Laufzeitverbesserung

Reduzierung des Energieverbrauchs

Loop Nest Splitting wurde auf drei Multimediaprogramme bzw. –algorithmen angewendet
1.) Cavity:
→ medizinisches Bildverarbeitungsprogramm
→ unterstützt Erkennung von Gehirntumoren
→ auf Code wurden bereits andere Transformationen vor Benchmark durchgeführt → Transformationen erzeugten Overhead
→ deshalb Loop Nest Splitting
2.) MPEG-4 Full Search Motion Estimation:
→ Algorithmus für Bewegungsvorhersage
→ beinhaltet 6-fach ineinander verschachtelte Schleifen
3.) QSDPCM Algorithmus
→ Quadtree structured pulse code modulation
→ Algorithmus für Videokomprimierung
→ ähnlich wie Motion Estimation
→ 7-fach ineinander verschachtelte Schleifen
Abbildungen zeigen Benchmark Ergebnisse
Laufzeit verbessert sich durch Loop Nest Splitting ebenso wird der Energieverbrauch reduziert

3.2 Software Washing Machine



Erhöhung der Codegröße

Allerdings wird der Code durch Loop Nest Splitting größer, da Code absichtlich wiederholt wird.
→ mehrere hundert Instruktionen mehr
→ trotzdem kein großer Nachteil des Algorithmus‘, da der Energieverbrauch, um die zusätzlichen Instruktionen zu speichern, nicht die Einsparungen, die man durch den Algorithmus erhält (siehe vorherige Folie) übertrifft

3.2 Software Washing Machine

Weitere Source Code Optimierungen

- Boolesche Ausdrücke
- Register
- Variablen und Funktionen

- Energiereduktion für Prozessor StrongARM gemessen

[10, 13, 18]

3.2 Software Washing Machine

- Boolesche Ausdrücke als konditionale arithmetische Ausdrücke
  - Energiereduktion 10,6 %

```
bool PointInRect1(Point p, Rectangle *r) {
        return ( p.x >= r->xmin &&
                 p.x < r->xmax &&
                 p.y >= r->ymin &&
                 p.y < r->ymax );
}
```

```
bool PointInRect2(Point p, Rectangle *r) {
        return ( (unsigned) (p.x - r->xmin) < r->xmax &&
        (unsigned) (p.y - r->ymin) < r->ymax);
}
```

3.2 Software Washing Machine

- Register
  - Lokale Variablen statt Parameter
    - Kann in Register abgelegt werden
    - 9,54 % Energiereduktion
  - Lokale Kopie einer globalen Variable
    - 6,42 % Energiereduktion
  - Lokale Kopie einer Pointer-Kette
    - z. B. p->pos->x
    - 34 % Energiereduktion

Wert der übergebenen Parameter in lokale Variable kopieren → kann in Registern abgelegt werden → Zugriff verbraucht weniger Energie
→Ebenso bei globalen Variablen

Pointer-Kette p->pos->x
→Kopie von p->pos in eine lokale Variable
→Keine mehrfachen Überprüfungen des Wertes

3.2 Software Washing Machine

- Variablen und Funktionen
  - Integer statt char und short
    - 18,2 % bzw. 0,39 % Energiereduktion
  - <= 4 Parameter
    - Verwendung von Registern
    - 90 % Energiereduktion
  - Inlinefunktionen
    - 17 % Energiereduktion
  - Platzierung von Funktionen
    - 24 % Energiereduktion

-Bei weniger als vier Parametern Register können zur Abarbeitung der Funktion verwendet werden.
-Funktion als inline deklarieren → Code der Funktion wird direkt an der Stelle des Funktionsaufrufs eingesetzt
-Reihenfolge von Funktionen im Quellcode hat Auswirkungen auf Energieverbrauch

Beispiel:

```
int f(int n) {
   return square(n) + square(n);
}
int square(int x){
   return x * x;
}
```

benötigt mehr Energie als

```
int square(int x){
   return x * x;
}
int f(int n) {
   return square(n) + square(n);
}
```

Gliederung

1. Einleitung
2. Eingebettete Systeme
3. Ansätze für energiesparende Programmierung
   1. Compileroptimierung
   2. Software Washing Machine
   3. Energiesparende Kryptographie und Datenkomprimierung
   4. Ausnutzen von Speicherhierarchien
   5. Minimalistische Software
   6. Software Engineering
4. Zusammenfassung und Ausblick

3.3 Energiesparende Kryptographie und Datenkomprimierung

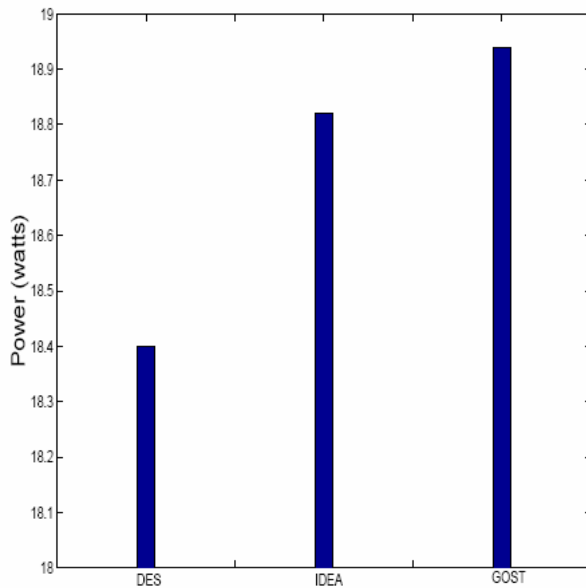
Energiesparende Kryptographie

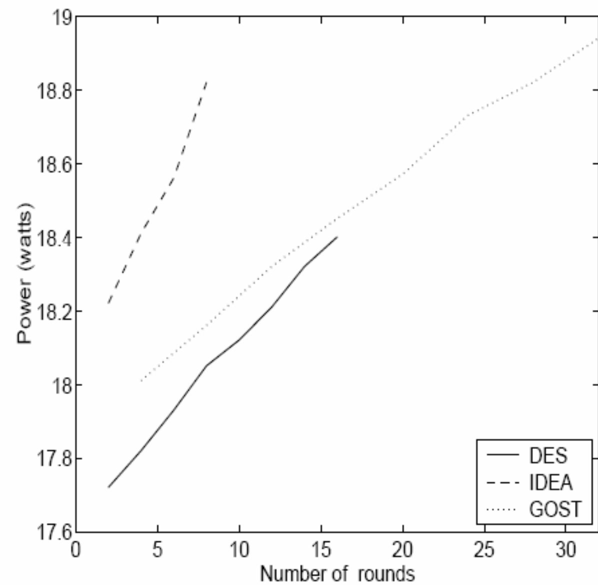
- Wichtige Sicherheitsmaßnahme bei Übertragung von Daten
- Algorithmen sehr rechen- und speicherintensiv
- Abhängig von Schlüssellänge + Anzahl der Verschlüsselungsrunden

[20, 23]
Bei Verschlüsselung Einsatz komplexer mathematischer Operationen
Energieverbrauch ist abhängig von der Länge des verwendeten Schlüssels und der Anzahl der Verschlüsselungsrunden (siehe Abbildungen auf folgenden Folien)
→ Generell asymmetrische Verschlüsselungsverfahren benötigen mehr Energie als symmetrische Verfahren

3.3 Energiesparende Krypto-graphie und Datenkomprimierung

Energiesparende Kryptographie





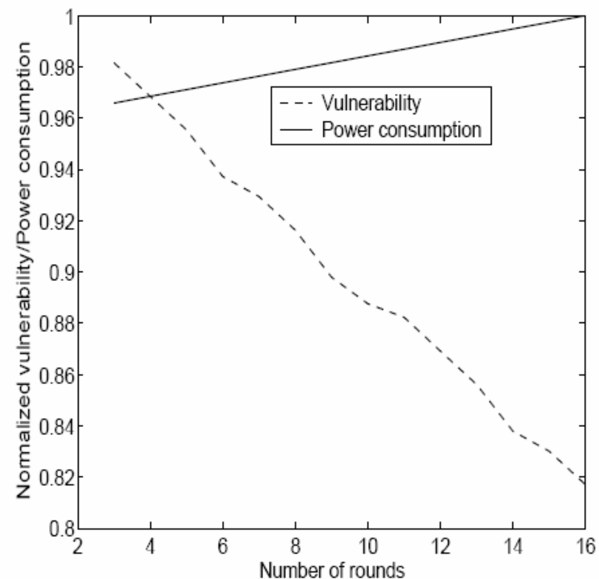
DES (Data Encryption Standard): Schlüssellänge 56 Bit, 16 Verschlüsselungsrunden
IDEA (International Data Encryption Algorithm): Schlüssellänge 128 Bit, 8 Verschlüsselungsrunden
GOST (Gossudarstwenny Standart): Schlüssellänge 256 Bit, 32 Verschlüsselungsrunden

Benchmarks ausgeführt auf einem Sony Vaio 700 Mhz Pentium 3 mit 128 MB RAM und Red Hat Linux 2.4.8 Betriebssystem

3.3 Energiesparende Kryptographie und Datenkomprimierung

Energiesparende Kryptographie

Ansatz: Nicht-kritische Informationen mit weniger Sicherheit übertragen



-Herausforderung: Energieverbrauch reduzieren, aber keine Einbußungen in der Sicherheit haben
- Längere Schlüssel + mehr Verschlüsselungsrunden → hohe Sicherheit, allerdings hoher Energieverbrauch
- Kleinere Schlüssel + weniger Verschlüsselungsrunden → geringer Energieverbrauch, allerdings geringe Sicherheit
- Trade-Off zwischen Sicherheit des Verschlüsselungsalgorithmus und dessen Energieverbrauch (siehe Abbildung)
- Schwierig Sicherheit eines Algorithmus zu messen: Einsatz von Kryptoanalysemethoden und der Brute-Force-Attacke
- Schwachstelle (Vulnerability): Durchschnitt der maximalen Anzahl an Klartexten, die von der Brute-Force-Attacke benötigt werden und der Anzahl an Klartexten, die benötigt werden, um mit Kryptoanalysemethoden den kryptografischen Schlüssel zu finden

Ansatz: Informationen, die nicht sicherheitskritisch sind können mit weniger Sicherheit übertragen werden → z. B. kleinere Schlüssellänge, weniger Verschlüsselungsrunden → es müssen Algorithmen entwickelt werden, die solche Pakete bzw. Informationen identifizieren

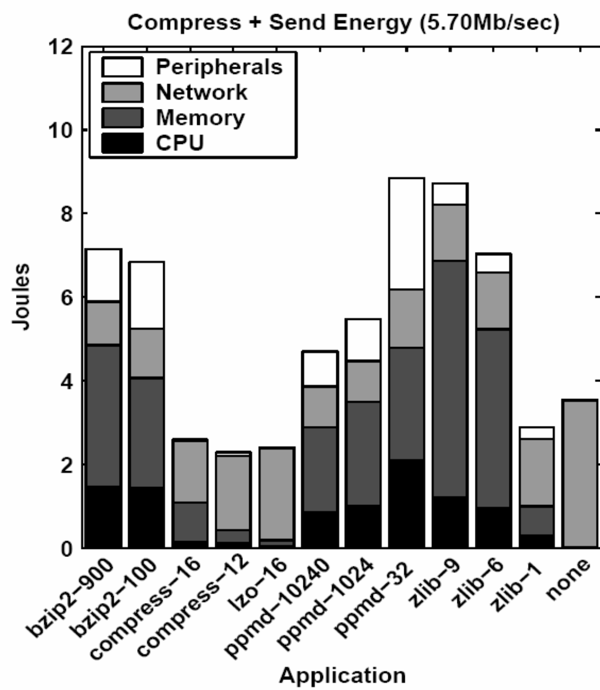
3.3 Energiesparende Krypto-graphie und Datenkomprimierung

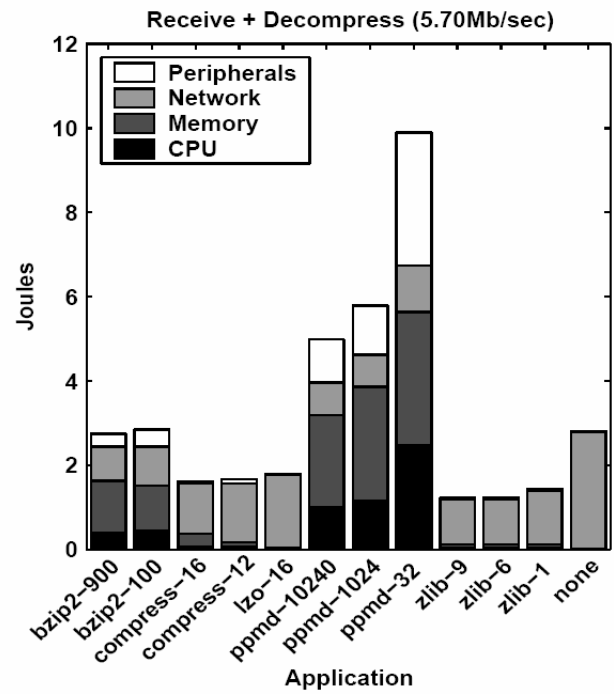
Energiesparende Datenkomprimierung

- Energie zur Übertragung eines Bits 1000mal mehr Energie als 32-Bit Berechnung
  - ABER: Speicherzugriff 200mal mehr Energie als Berechnungen
- Nutzen abhängig von Hardware- und Softwarefaktoren

[24]
- Datenkomprimierung kann den Energieverbrauch reduzieren, aber es kann auch sein, dass die Speicherzugriffe, die während den Berechnungen zur Datenkomprimierung durchgeführt werden, dominieren, so dass der Verbrauch zum Versenden der Bits weniger wäre als derjenige zum Komprimieren und Versenden der Bits
- Abwägung durch Betrachtung von Hardware- und Softwarefaktoren
- Hardwarefaktoren: CPU-, Speicher-, Netzwerkenergieverbrauch
- Softwarefaktoren: Komprimierungsverhältnis, Speicherzugriffsstatistik
- Faktoren können sich mit der Zeit ändern → muss regelmäßig neu ausgewertet werden

3.3 Energiesparende Krypto-graphie und Datenkomprimierung





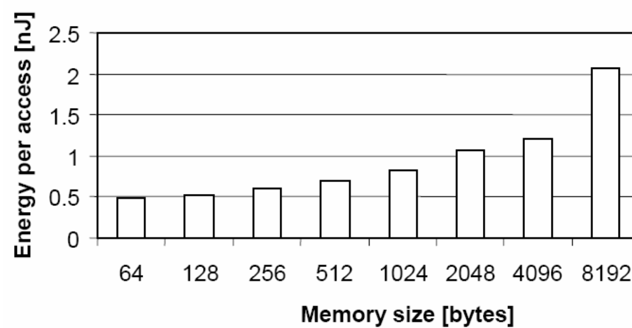
Benchmarks für CPU-, Speicher-, Netzwerkenergieverbrauch → Nutzen nur, wenn durch Kompressionsverfahren weniger Gesamtverbrauch als ohne (ganz rechts im Diagramm)

Gliederung

1. Einleitung
2. Eingebettete Systeme
3. Ansätze für energiesparende Programmierung
    1. Compileroptimierung
    2. Software Washing Machine
    3. Energiesparende Kryptographie und Datenkomprimierung
    4. Ausnutzen von Speicherhierarchien
    5. Minimalistische Software
    6. Software Engineering
4. Zusammenfassung und Ausblick

3.4 Speicherhierarchien

- Schnellere Zugriffe durch kleinere Speicher
  - Geringerer Energieverbrauch
- Verwendung von Speicherhierarchien
  - Verbesserung der Speicherzugriffszeiten
  - Reduzierung des Energieverbrauchs



[3, 4, 5, 25, 26, 27]
Speichersystem Hauptursache für hohen Energieverbrauch

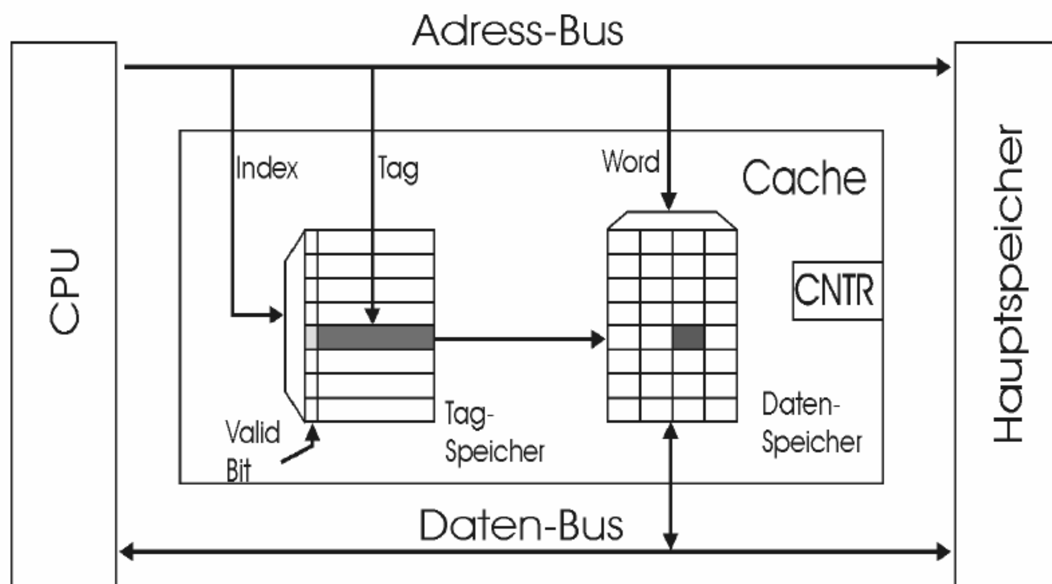
3.4 Speicherhierarchien

- Caches
  - kleinerer Speicher → Verringerung des Energieverbrauchs
  - Tag-Speicher + Daten-Speicher + Hardwarekontrolllogik
  - nicht immer optimal
  - Überprüfung, ob Daten im Cache
    - Energieverbrauch durch Zugriffe auf Tags
    - Steigt mit Anzahl der Sätze

Überprüfung, ob Daten im Cache kann zu Cache Misses führen, d. h. Daten sind nicht im Cache → unnötiger Energieverbrauch
Zugriffe auf Tags benötigen Energie → je mehr Sätze, desto höher der Energieverbrauch → am höchsten für vollassoziativen Cache

3.4 Speicherhierarchien

- Caches



Architektur eines Caches

3.4 Speicherhierarchien

- Scratch-Pad Memories (SPM)
  - On-Chip Speicher
  - Schnell und klein
  - Keine Tag-Speicher + Hardwarekontrolllogik
    - Zugriff durch Verwendung der Adresse
    - Compiler übernimmt Kontrolle
    - Weniger Wartezyklen im Prozessor
    - Geringerer Energieverbrauch

Bei Caches ist nur der L1-Cache ein On-Chip Speicher (L2-Cache i.d.R. Off-Chip Speicher)
→Energieverbrauch pro Zugriff auf On-Chip Speicher viel geringer als auf Off-Chip Speicher

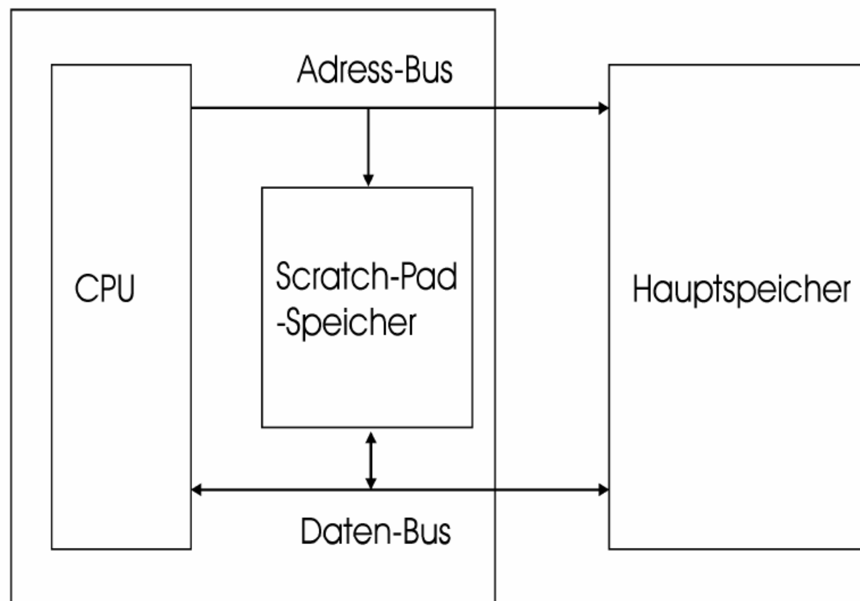
Bei SPM sind Programmierer bzw. der Compiler für die Steuerung verantwortlich, da keine Hardwarekontrolllogik existiert
→ Keine Misses möglich, da durch Steuerung des Programmierers bzw. durch den Compiler immer bekannt, welcher Inhalt sich im SPM befindet
→ Keine energieverbrauchenden Vergleiche der Cache-Adresse mit der aktuell gültigen Adresse
→ häufig ausgeführte Programmteile in den SPM kopiert
→ Ausführung von Instruktionen energieeffizienter als bei Caches
→ Allerdings: das Kopieren der Programmteile ins SPM verbraucht mehr Energie als beim Cache → trotzdem insgesamt energieeffizienter

Zum Kopieren werden Kopierfunktionen in die Anwendung eingefügt zur Kompilierungszeit
→Algorithmus:

1.) Code analysieren, um Programmteile zu identifizieren
2.) mögliche Programmteile und Orte für die Kopierfunktionen im Kontrollfluss der Software identifizieren
3.) Auswahl der besten Menge an Programmteilen, die zusammen mithilfe der Kopierfunktion ins SPM kopiert werden um in Zukunft von dort ausgeführt zu werden
→ z. B. Analyse der Größe und der Anzahl der Ausführungen eines Programmteils

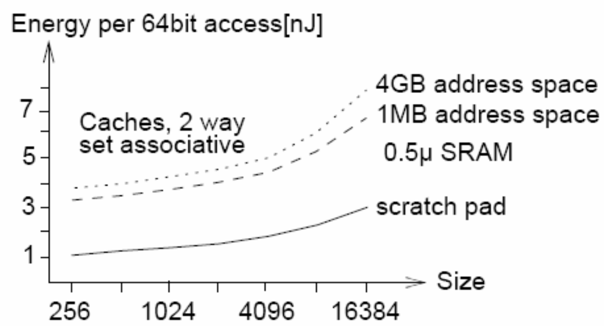
3.4 Speicherhierarchien

- Scratch-Pad Memories (SPM)

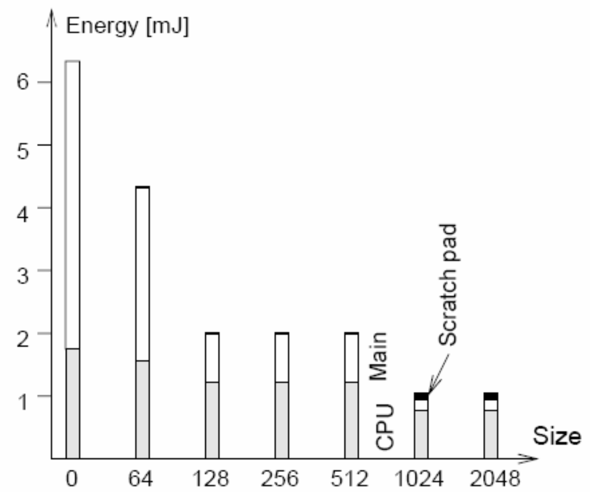


Architektur eines SPMs

3.4 Speicherhierarchien



Energieverbrauch für Cache und SPM



Energieeinsparungen durch Verwendung von SPM

Verbesserung des Energieverbrauchs um durchschnittlich 29,9 % gegenüber Caches
Abbildungen aus [4]

Gliederung

1. Einleitung
2. Eingebettete Systeme
3. Ansätze für energiesparende Programmierung
    1. Compileroptimierung
    2. Software Washing Machine
    3. Energiesparende Kryptographie und Datenkomprimierung
    4. Ausnutzen von Speicherhierarchien
    5. Minimalistische Software
    6. Software Engineering
4. Zusammenfassung und Ausblick

3.5 Minimalistische Software

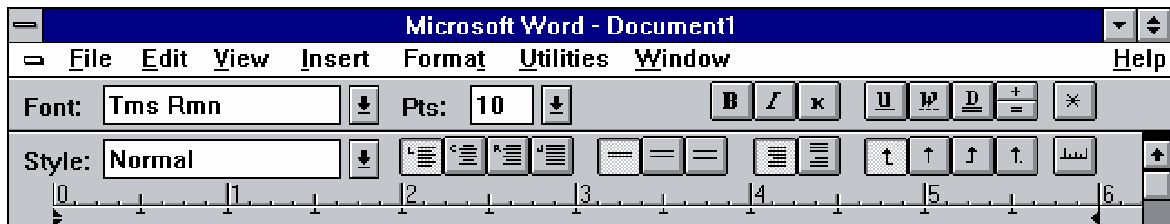
- Beschränkung auf wesentliche Funktionalität
  - z. B. Entfernung von Desktopsymbolen, Toolbars, Verschönerungen
- Reduzierung von Ressourcenverbrauch
  - Geringerer Speicherverbrauch
  - Schnellere Laufzeiten
- Heutzutage viele Abandonware als minimalistisch angesehen, z. B. Microsoft Word 1.0

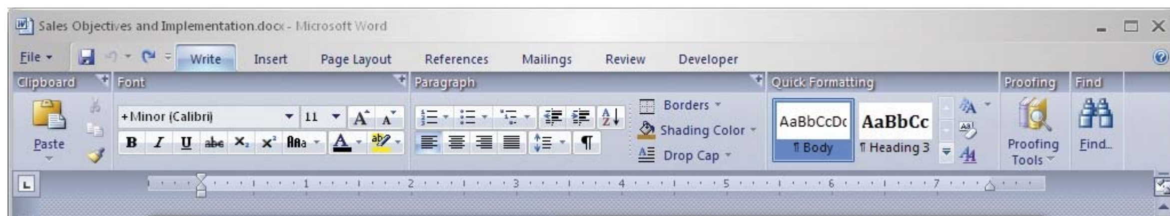
Softwareentwicklungen führen oft zu Implementierungen von Features, die von den Kernanforderungen an die Software abweichen → unnötiger Ressourcenverbrauch → höherer Energieverbrauch → Entwicklung von minimalistischer Software als Ansatz für die Reduzierung des Energieverbrauchs
In minimalistischer Software: sehr einfaches User Interface, Graphical User Interface und sehr einfache Interaktionen zwischen Benutzer und System
Beispiele: Oberon als BS, reine Textbrowser, Paint als Bildbearbeitungsprogramm

Abandonware: alte Software, die nicht mehr vermarktet, unterstützt und verwendet wird

Minimalismus auch bei Programmiersprachen (z. B. Pascal, C, Assembler, Python) und Programmbibliotheken (z. B. C-Standardbibliotheken: uClibc, dietlibc anstatt Glibc)

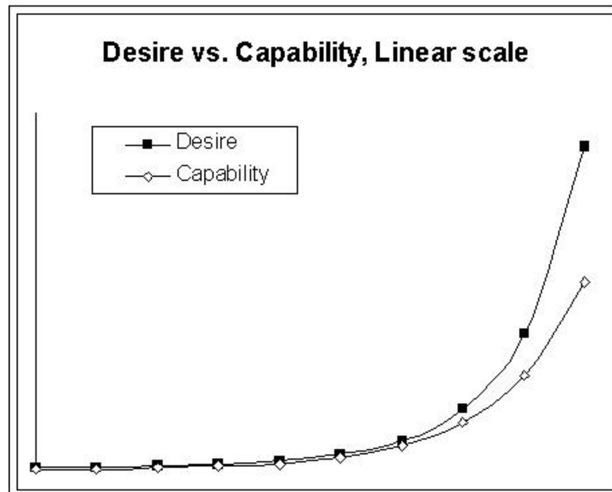
3.5 Minimalistische Software

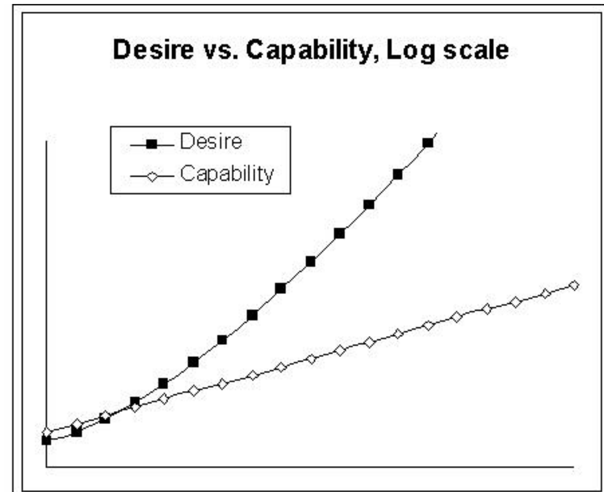




3.5 Minimalistische Software

- Problem: Ansprüche von Benutzern steigt stetig





Entwicklung minimalistischer Software zur Energiereduzierung kaum möglich → Benutzer der Software haben hohe Ansprüche, die stetig steigen (siehe Abbildung) → viele Benutzer wollen schöne graphische Benutzerschnittstellen und Software mit vielen Features

Gliederung

1. Einleitung
2. Eingebettete Systeme
3. Ansätze für energiesparende Programmierung
   1. Compileroptimierung
   2. Software Washing Machine
   3. Kryptographie und Datenkomprimierung
   4. Energiesparende Ausnutzen von Speicherhierarchien
   5. Minimalistische Software
   6. Software Engineering
4. Zusammenfassung und Ausblick

3.6 Software Engineering

- Entwicklung + Weiterentwicklung von Software oft durch verschiedene Personen
- Dokumentation wichtig
- Nachvollziehbarkeit von Entscheidungen
- Verständnis des Codes wesentlich
  - Codeoptimierungen
  - Wahl eines Algorithmus
  - Verhinderung von Code Clones, Bloating (ineffizienter Code)

Da verschiedene Personen bei der Entwicklung bzw. Weiterentwicklung von Software beteiligt sind, ist Dokumentation wichtig, insbesondere Kommentare im Quellcode, um nachvollziehen zu können, warum bestimmte Algorithmen gewählt wurden, warum Funktionalitäten auf diese Weise implementiert wurden usw.
→ z. B. kann eine umständlichere Implementierung vorliegen, um den Energieverbrauch zu reduzieren
→Wenn beteiligte Personen den Code der anderen nicht verstehen (eventuell sind diese Personen auch nicht mehr erreichbar, um nachzufragen), dann besteht die Gefahr, dass Code Clones (durch Copy & Paste erzeugter Code) oder Bloating (Erweiterung des Codes ohne zusätzliche Funktionalität, z. B. Neuschreiben einer bereits existierenden Funktionalität, da der Code nicht verstanden wurde) im Code auftauchen
→Folge: ineffizienter Code; Umschreiben von Code, der eigentlich zur Energiereduzierung derart geschrieben wurde

Gliederung

1. Einleitung
2. Eingebettete Systeme
3. Ansätze für energiesparende Programmierung
    1. Compileroptimierung
    2. Software Washing Machine
    3. Energiesparende Kryptographie und Datenkomprimierung
    4. Ausnutzen von Speicherhierarchien
    5. Minimalistische Software
    6. Software Engineering
4. Zusammenfassung und Ausblick

4. Zusammenfassung und Ausblick

- Effiziente Programme reduzieren Energieverbrauch
- Anpassung der Codeoptimierungen durch Compiler
- Konzept der Software Washing Machine noch nicht durchgesetzt
  - Mehrere Ansätze → wenig konkrete Umsetzungen

4. Zusammenfassung und Ausblick

- Optimierung von Speicherzugriffen
    - Verwendung von Speicherhierarchien
- Minimalisierung von Software
    - Effizienzsteigerung durch Beschränkung auf das Wesentliche
    - ABER: Widerspricht Wünschen von Benutzern

4. Zusammenfassung und Ausblick

- Methoden des Software Engineerings erhöhen Programmverständnis
  - Verhindern Bloating, Code Clones
- Energiesparende Programmierung bisher nur für eingebettete Systeme in Betracht gezogen
  - Auch für normale Anwendungssoftware sinnvoll

Literatur

[1] WIRTH, Niklaus: A Plea for Lean Software. – URL http://cr.yp.to/bib/1995/wirth.pdf. Zugriffsdatum: 20.07.2008

[2]COCKBURN, Alistair A. R.: Growth of human factors in application development. – URL http://alistair.cockburn.us/index.php/Growth_of_human_factors_in_application_development. - Zugriffsdatum: 20.07.2008

[3] MARWEDEL, Peter: Embedded system design. 2. Berlin: Springer, 2005

[4] MARWEDEL, Peter: Embedded Software: How To Make It Efficient? – URL http://ls12-www.cs.tu-dortmund.de//publications/papers/2002-euromicro.pdf. - Zugriffsdatum: 20.07.2008

[5] MARWEDEL, Peter: Compilation for Embedded Processors. – URL http://ls12-www.cs.uni-dortmund.de/%7Emarwedel/kluwer-es-book/es-marw-5d-scratchpad.pdf. - Zugriffsdatum: 20.07.2008

[6] TIWARI, Vivek; MALIK, Sharad; WOLFE, Andrew; LEE, Mike T.: Instruction Level Power Analysis and Optimization of Software. – URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.4245&rep=rep1&type=pdf. – Zugriffsdatum: 20.07.2008

[7] MARWEDEL, Peter: Generation of efficient software code for embedded systems on a chip. – URL http://medea.it.uc3m.es/2003/6.1_.pdf. Zugriffsdatum: 20.07.2008

[8] DE MAN, Hugo: On Nanoscale Integration and Gigascale Complexity in the Post.com world. – URL http://www.date-conference.com/conference/2002/keynotes/deman/deman_slides.pdf. Zugriffsdatum: 20.07.2008

[9] ISS RWTH Aachen: Software Washing Machine for Embedded Code. – URL http://www.iss.rwth-aachen.de/Projekte/SSS/washingmachine.html. Zugriffsdatum: 20.07.2008

[10] SIMUNIC, Tajana; BENINI, Luca; DE MICHELI, Giovanni: Präsentation: Energy-Efficient Design of Battery-Powered Embedded Systems – URL http://akebono.stanford.edu/users/tajana/presentations/islped99_slides.pdf. Zugriffsdatum: 20.07.2008

Literatur

[11] MARWEDEL, Peter: Energy-Aware Computing – Lecture 11: Software-level techniques. URL – http://www.inf.ed.ac.uk/teaching/courses/eac/L11.pdf. Zugriffsdatum: 20.07.2008

[12] FALK, Heiko; MARWEDEL, Peter: Control Flow driven Splitting of Loop Nests at the Source Code Level. URL - http://ls12-www.cs.tu-dortmund.de/publications/papers/2003-date.pdf. Zugriffsdatum: 20.07.2008

[13] SIMUNIC, Tajana; BENINI, Luca; DE MICHELI, Giovanni: Energy-Efficient Design of Battery-Powered Embedded Systems. – URL http://akebono.stanford.edu/users/tajana/papers/islped99_paper.ps. Zugriffsdatum: 20.07.2008

[14] JONES, Nigel: Efficient C Code for Eight-Bit MCUs. URL – http://www.ece.cmu.edu/~ece348/reading/jones98_efficient_c_for_8bit_mcu.pdf. Zugriffsdatum: 20.07.2008

[15] CGO: Optimizations for DSP and Embedded Systems (ODES-2). URL – http://www.ece.vill.edu/~deepu/odes/odes-2_digest.pdf. Zugriffsdatum: 20.07.2008

[16] CHUNG, Eui-Young; BENINI, Luca; DE MICHELI, Giovanni: Source Code Transformation based on Software Cost Analysis. URL – http://si2.epfl.ch/~demichel/publications/archive/2001/ISSSconf01pg153.pdf. Zugriffsdatum: 20.07.2008

[17] SINHA, Amit: Energy Aware Software. URL – http://mtlweb.mit.edu/researchgroups/icsystems/pubs/theses/sinha_sm_1999.pdf. Zugriffsdatum: 20.07.2008

[18] SIMUNIC, Tajana: Energy Efficient System Design And Utilization. Stanford University, Dissertation, 2001, URL – http://akebono.stanford.edu/users/tajana/papers/thesis.pdf. Zugriffsdatum: 20.07.2008

[19] SCHWARZER, Martin: Untersuchung des Einflusses von Compiler-Optimierungen auf die maximale Programm-Laufzeit (WCET). Universität Dortmund, Diplomarbeit, 2007, URL – http://ls12-www.cs.tu-dortmund.de/publications/theses/schwarzer.pdf.gz. Zugriffsdatum: 20.07.2008

[20] CHANDRAMOULI, R.; BAPATLA, S.; SUBBALAKSHMI, K. P.; UMA, R. N.: Battery Power-aware Encryption. URL – http://www.ece.stevens-tech.edu/~mouli/jbatenc.pdf. Zugriffsdatum: 20.07.2008

Literatur

[21] FLINN, Jason; SATYANARAYANAN, M.: Managing Battery Lifetime with Energy-Aware Adaptation. URL – http://www.cs.cmu.edu/~satya/docdir/p137-flinn.pdf. Zugriffsdatum: 20.07.2008

[22] SIMUNIC, Tajana; BENINI, Luca; DE MICHELI, Giovanni; HANS, Mat: Source Code Optimization and Profiling of Energy Consumption in Embedded Systems. URL – http://akebono.stanford.edu/users/tajana/papers/isss00_paper.ps. Zugriffsdatum: 20.07.2008

[23] KRISHNAMURTHY, P.; RUANGCHAIJATUPON, N.: Encryption and Power Consumption in Wireless LANs. URL – http://www.wlan01.wpi.edu/proceedings/wlan08d.pdf. Zugriffsdatum: 20.07.2008

[24] BARR, Kenneth; ASANOVIC, Krste: Energy Aware Lossless Data Compression. URL – http://www.cag.csail.mit.edu/scale/papers/compression-mobisys2003.pdf. Zugriffsdatum: 20.07.2008

[25] STEINKE, Stefan; GRUNWALD, Nils; WEHMEYER, Lars; BANAKAR, Rajeshwari; BALAKRISHNAN, M.; MARWEDEL, Peter: Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory. URL – http://ls12-www.cs.tu-dortmund.de/publications/papers/2002-isss.pdf. Zugriffsdatum: 20.07. 2008

[26] VERMA, Manish; WEHMEYER, Lars; PYKA, Robert; MARWEDEL, Peter; BENINI, Luca: Compilation and Simulation Tool Chain for Memory Aware Energy Optimizations. URL - http://ls12-www.cs.tu-dortmund.de//publications/papers/2006-samos.pdf. Zugriffsdatum: 20.07.2008

[27] LEE, Bo-Sik: Vergleich des Energieverbrauchs von Cache- und Scratch-Pad-Speichern für den ARM7-Prozessor. Universität Dortmund, Diplomarbeit, 2001, URL – http://ls12-www.cs.tu-dortmund.de/publications/theses/lee.ps.gz. Zugriffsdatum: 20.07.2008